Proof Assistant

to Aid Proof Conversion and Verification

Shreyas Nagarajappa (u5834888)

Supervisor: Dr. John Slaney

Advanced Computing Project (COMP4560)

Australian National University

May 2017
Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this report are original and have not been submitted in whole or in part for consideration for any other course in this, or any other university. This report is my own work and contains nothing which is the outcome of work done in collaboration with others.

Shreyas Nagarajappa (u5834888)
May 2017
Acknowledgements

I would like to express my gratitude to John Slaney for offering me the project, and guiding my way through thick and thin. I would also like to thank Weifa Liang and Peter Strazdins for augmenting my learning experience during the duration of this course. An honorable mention to Australian National University and the College of Engineering and Computer Science for putting together and running the Advanced Computing Project program.
Abstract

Students who take an introductory logic course tend to find it difficult to gain an abstract view of natural deduction proofs and instead approach proofs with an algorithmic approach, which is undesirable. The aim of the project was to create a software artifact, called ProofAssistant, that helps students understand the essence of natural deduction. To help understand natural deduction, proofs will be represented in two different notations and these proofs can be converted from one notation to the other. The process of conversion involves the creation of an abstract representation of proofs in the backend of ProofAssistant. When looking at proofs in two different notations, the students are expected to understand natural deduction at an abstract level by separating the characteristics of an individual notation from the characteristics of the proof. Doing so would enable these students to excel in areas like machine learning that are reliant on natural deduction.
# Table of contents

List of figures xi

## 1 Introduction 1
1.1 Background of Logic Resources 1
1.2 Motivation for Building a Proof Assistant 1
1.3 Objectives of the Project 2
1.4 Delimitation 3

## 2 Logic Preliminaries 5
2.1 Terminology 5
2.2 Modified Fitch Notation 7
2.3 Modified Lemmon Notation 8
2.4 Differences between Fitch and Lemmon Rules 9

## 3 Software Functional Specification 11
3.1 Overview 11
3.2 Screen by Screen Specification 11
3.3 Scenarios 14
3.4 Non-Goals 14

## 4 Software Technical Implementation 17
4.1 Overview 17
4.2 Requirements 17
4.3 Architecture 18
4.4 Backend 19
4.5 Frontend: Web Client 20
4.6 Using Mixed Technologies 22
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Proof Conversion and Verification Methodology</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview</td>
<td>25</td>
</tr>
<tr>
<td>5.2</td>
<td>Parsing</td>
<td>25</td>
</tr>
<tr>
<td>5.3</td>
<td>Proof Verification</td>
<td>26</td>
</tr>
<tr>
<td>5.4</td>
<td>Proof Conversion</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>Testing and Evaluation</td>
<td>33</td>
</tr>
<tr>
<td>6.1</td>
<td>Overview</td>
<td>33</td>
</tr>
<tr>
<td>6.2</td>
<td>Performance Evaluation</td>
<td>33</td>
</tr>
<tr>
<td>6.3</td>
<td>Testing for Correctness</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>Limitations and Future Work</td>
<td>37</td>
</tr>
<tr>
<td>7.1</td>
<td>Overview</td>
<td>37</td>
</tr>
<tr>
<td>7.2</td>
<td>Support Quantifiers</td>
<td>37</td>
</tr>
<tr>
<td>7.3</td>
<td>Investigate Real Sources of Error</td>
<td>37</td>
</tr>
<tr>
<td>7.4</td>
<td>Use Logged Data</td>
<td>38</td>
</tr>
<tr>
<td>7.5</td>
<td>Migration from .Net to .NET Core</td>
<td>38</td>
</tr>
</tbody>
</table>

References

Appendix A  Fitch Notation Rules

Appendix B  Lemmon Notation Rules

Appendix C  Independent Study Contract

Appendix D  README.md from Code Repository
List of figures

3.1 ProofAssistant Home Screen ........................................... 12
3.2 Proof Input Screen .................................................... 12
3.3 ProofAssistant Instructions Screen ................................. 13

4.1 UML Diagram ............................................................. 18
4.2 Asynchronous DB ........................................................ 20
4.3 Database Network User ............................................... 20
4.4 UI Forms ................................................................. 21
4.5 Ajax Post ................................................................. 22

5.1 And Elimination Rule .................................................... 27
5.2 Proof Tree ............................................................... 28
5.3 Duplicate Formula in Fitch .......................................... 31

6.1 Resources Used ........................................................... 34
6.2 Logic Notes Exercises ............................................... 35
Chapter 1

Introduction

1.1 Background of Logic Resources

The Australian National University (ANU), under the supervision of John Slaney\(^1\), is hosting an ecosystem of resources pertaining to learning and practicing logic. These resources range from a basic guide to logic in the form of *The Logic Notes* website\(^2\) to challenging puzzles for more advanced logicians through the *Logic for Fun* website\(^3\).

The *Logic Notes* resource provides a theoretical treatment of logic for beginners and it emphasizes the use of proofs to express logic. The users of this resource usually practice the exercise problems by hand. This introduces the difficulty of providing users feedback on their work. While the Logical for Fun resource does have a solver built into it, the style of entering proofs is different from that denoted in the Logic Notes, and it exposes much more complexity than that is required by a beginner. So, these tools do not provide a comprehensive coverage of logic.

The *ProofAssistant* project was conceived to build a software tool that will complement this ecosystem of resources. The software tool is a web based proof assistant for first order logic. It is a tool that beginners can use to practice proofs in the style dictated by the Logic Notes to reinforce the preliminary concepts in the reading.

1.2 Motivation for Building a Proof Assistant

Logic proofs are a common approach to teaching beginners the foundations of logic. To an outsider, while proofs may appear to be mere symbol manipulation, proofs are in fact aimed

\(^{3}\)https://www.l4f.cecs.anu.edu.au/
at teaching general methods and concepts that are useful independently of the semantics of the proof. Proofs help teach deductive inferences, where one uses a collection of statements to reason about another statement. Deductive reasoning and inferences finds a vast number of applications in the formal branches of disciplines like computer science, linguistics and mathematics. [2]

Unfortunately, the significance of proofs seems to be lost on the students of logic. In his experience of teaching logic at the ANU, John Slaney noticed that students try to memorize proofs or take an algorithmic approach to symbol manipulation when presented with proofs to solve in exams and assignments. Some students do not seek to understand why things work the way they do and rather prefer to memorize a lot of disconnected facts. The ProofAssistant resource is aimed at providing tools to help students understand the fundamentals of natural deduction.

While industrial strength proof assistants like Isabella and Coq do exist in the wild, they are unfit for helping beginners. They are designed to be used with higher order logic for large scale projects. They also define their own structured proof language which detracts from their ease of use for beginners. [19]

1.3 Objectives of the Project

The primary objective of the project is to stop students who learn logic for the first time from learning proofs by rote. This in turn is expected to prepare them to excel in advanced topics like machine learning and algorithm design whose foundations are based in deductive reasoning [17].

From a functional standpoint, the software will allow users to verify their proofs for correctness and will allow users to enter proofs in different notations, and allows the conversion of proofs between these notations.

Conversion of proofs between two notations will help students not get lost in the details of any one proof notation. They can view proofs in an abstract sense and can discern between the characteristics of the proof and intricacies of the notation. For example, in my introductory logic course at the ANU (COMP2600), I did not understand the nature of assumptions made in sub-proofs in the Fitch notation (see Chapter 2 Section 2) where these assumptions seemed to disappear by the end of the proof. When I started writing proofs in the Lemmon notation (see Chapter 2 Section 2), discharging assumptions in Lemmon style helped me understand that the conclusion was derived from a small set of assumptions despite a larger assumption set playing a part in the proof.
From a technical standpoint, the focus has been on writing the code in a maintainable and readable fashion. The code is modular and the project is open to extension without rewriting existing code. This will help future developers add features to the project that will in turn help students understand logic further.

1.4 Delimitation

In 2016, the College of Engineering and Computer Science at the ANU ran two introductory logic courses.[21] They were COMP2600 (Formal Methods for Software Engineering), where first order logic proofs were thought using the Fitch notation, and COMP2620 (Logic), which used the Lemmon notation. Students at the ANU usually studied one of these courses. These students were to be the primary audience for ProofAssistant.

When the ProofAssistant project was conceived in spring 2016, the aim was to help these students enter proofs in the systems they were comfortable with, while enabling them to also learn the system from the other course. Due to a subsequent change in curriculum, the syllabus of COMP1600 (replacement for COMP2600) was not set in stone when the project started. For the purposes of this project, I assume that COMP2600 and COMP2620 will run as they did in 2016.

In my opinion, it was a good risk mitigation decision given the inflexible time frame of the project. Waiting for more information might not have given the project enough time to meet its goals. This was the best path to take in order to keep true to the aim of helping students learn logic.
Chapter 2

Logic Preliminaries

2.1 Terminology

This chapter contains the preliminary knowledge required to understand the remainder of this report. It is a collection of definitions and ideas that pertain to logic, and their significance.

Logic

Logic is a formal discipline that is part philosophy and part mathematics. It is the system study of the form of arguments and reasoning about their validity. Logic consists of a formal language designed to enable things to be said clearly and unambiguously, together with a deductive system used to capture and codify arguments that are valid in the given language. Of late, logic has played an integral part in computer science.[24][23]

Natural Deduction

The concept of natural deduction is a generalization of the concept of a formal proof. A formal proof or derivation is a finite sequence of sentences whose last sentence is the conclusion of the sequence (also called a theorem) in a formal system. Each sentence of the sequence is either an axiom, an assumption, or follows from the preceding sentences in the sequence by a rule of inference.[1]

Natural deduction is a family of proof systems in which deductive reasoning is expressed by inference rules closely related to the “natural” way of reasoning[9]. Natural deduction proofs are naturally axiomless[1]. Proofs of theorems within a natural deduction system than those in systems that rely heavily on axioms[1].
Proof Assistant

A proof assistant is a software tool to assist with the development of formal proofs by human-machine collaboration. This involves some sort of interactive proof editor, or other interface, with which a human can guide the search for proofs.

Formal proofs can be checked by computers effectively, but, finding proofs can be a difficult job. In fact this problem reduces to boolean satisfiability, which is NP complete [3]. Proof assistants are not to be confused with interactive theorem provers, which involve automated reasoning. Common proof assistants like Isabelle and Coq are used to solve complex mathematical problems, but, they do not give users proofs. Automated theorem provers give proofs, but, are limited by their complexity.[22]

First Order Logic

First order logic is a collection of formal systems in logic. First-order logic uses quantified variables over non-logical objects and allows the use of sentences that contain variables, so that rather than propositions such as Socrates is a man one can have expressions in the form "there exists X such that X is Socrates and X is a man" where there exists is a quantifier and X is a variable.[6] This distinguishes it from propositional logic, which does not use quantifiers or relations.[7]

The software is meant to handle both propositional logic and first order logic as both find themselves in introductory logic courses.

Proposition and Formulas

A proposition is a declarative sentence that is either true (denoted either T or 1) or false (denoted either F or 0). Variables, like p, q, and r, are usually used to represent propositions.[11]

In propositional logic, a propositional formula is a proposition that is well formed and is either true or false. If values of all variables are known, then the unique truth value of the proposition is known. An example of such a formula would be:

\[ \text{NOT}(p \text{ AND } q) \text{ IMPLIES (NOT } p \text{ OR NOT } q) \]

Sequent

A sequent is a conditional assertion like the one shown below.

\[ A_1, ..., A_i \vdash B_1, ..., B_j \]
2.2 Modified Fitch Notation

If a sequent may have i number of conditions represented above by A1 to Ai (antecedents), and j number of assertions represented above by B1 to Bj (consequents). If all the antecedents are true, then at least one of the consequents is true.\[10\]

**Rules**

In logic, a rule of inference, is a logical form consisting of a function which takes premises, analyzes their syntax, and returns one or more conclusions. Most rules come in one of two flavors: introduction or elimination rules. Introduction rules introduce the use of a logical operator, and elimination rules eliminate it.

**Standard Notation**

A notational system is a technical system of symbols used to represent special things. Therefore, a notation is a collection of related symbols that are each given an arbitrary meaning, created to facilitate structured communication within a domain knowledge or field of study. Standard notations refer to general agreements in the way things are written or denoted.

Both the Fitch and Lemmon notations are standard notations. However, slightly modified versions of these standard notations are used in COMP2600 and COMP2620 respectively, which are described in the following sections.

### 2.2 Modified Fitch Notation

This section describes the Fitch notation for natural deduction proofs as used in COMP2600. Each row in a Fitch-style proof is either:

- an assumption or subproof assumption or

- a sentence justified by the citation of

1. a rule of inference and
2. the prior line or lines of the proof that license that rule.

Introducing a new assumption increases the level of indentation, and begins a new vertical "scope" bar that continues to indent subsequent lines until the assumption is discharged. This mechanism immediately conveys which assumptions are active for any given line in the
proof, without the assumptions needing to be rewritten on every line (as with Lemmon style proofs).[5]

The following example displays the main features of Fitch notation:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>Assumption</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>Assumption</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>A</td>
<td>Repeat (1)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>B → A</td>
<td>→-Intro (3-4)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>A → (B → A)</td>
<td>→-Intro (2-5)</td>
<td></td>
</tr>
</tbody>
</table>

1. The line represents a null assumption, i.e., we are proving a tautology

2. The high level subproof: we start by assuming A

3. The nested subproof: we start by assuming B

4. Repeat line 1: we can repeat statements from an outer scope

5. Introduce an implication by saying that we can conclude B from A

6. Introduce an implication by saying that we can conclude B implies A from A

The standard set of rules used in COMP2600 can be found in Appendix A.

2.3 Modified Lemmon Notation

The Lemmon system is a natural deductive logic developed by E. J. Lemmon.[1] It represents natural deduction proofs as sequences of justified steps. In this system, a proof has a definition with the following conditions:

- has a finite sequence of well-formed formulas
- each line of it is justified by a rule of the system
- the last line of the proof is what is intended, and this last line of the proof uses only the premises that were given, if any
If no premise is given, the formula is called theorem. Therefore, the definition of a theorem in the Lemmon system is a formula that can be proved in this system, using an empty set of assumptions.[10]

An example of the proof of a sequent \((P \rightarrow Q) \rightarrow (P \rightarrow Q) \rightarrow Q\) follows

1. (1) \((P \rightarrow Q) \rightarrow P\) Assumption
2. (2) \(P \rightarrow Q\) Assumption
1, 2 (3) \(P\) \rightarrow-Elim 1, 2
1, 2 (4) \(Q\) \rightarrow-Elim 2, 3
1, 2 (5) \((P \rightarrow Q) \rightarrow Q\) \rightarrow-Intro 4[2]

1. The line represents an assumption
2. The line represents an assumption
3. Eliminate the second implication on line 1, because we have assumed the left side of the implication on line 2
4. Eliminate the implication on line 2, because we have derived the left side of the implication on line 3
5. Introduce an implication by saying that we can conclude \(Q\) from \(P\) implies \(Q\)

### 2.4 Differences between Fitch and Lemmon Rules

Lemmon, in his book *Beginning Logic*, argues that the predicate calculus rules used in the system he proposes can be found in Fitch.[10] This implies that when representing the same proof in the two notations, the information presented by both notations is identical. But the two systems do have some non-trivial differences that have to be dealt with by *ProofAssistant* that are highlighted in this section.

By looking at the sample proofs presented in the previous sections, it is readily visible that there exist differences in the graphical representation of the proofs. The first noticeable difference is that the Fitch notation allows the visualization of subproofs and nested subproofs. Whereas the Lemmon notation is slightly more abstract which allows the introduction of a theorem at any point in a proof.[20] In fact, subproofs and nested subproofs can be interleaved in this notation. This allows more flexibility in terms of the order in which formulas can
appear in a Lemmon proof. The second graphical difference is that the proof lines are indented in the Fitch system to indicate the assumptions the proof line depends on. In the Lemmon system, these dependencies are listed explicitly to the left of each proof line.

There also exist rules and symbols that are unique to each notation. The first of these is the representation of a contradiction. In the Lemmon system, a contradiction is represented using the $\bot$ symbol and calls the rule that introduces this symbol a negation elimination rule. In the Fitch system, a contraction is implied by the conjunction of the two contradicting formulas. While the negation introduction rules behave similarly in the two notations, the negation introduction rule in the Fitch system has no direct replacement in the Lemmon system. The Lemmon system similarly specifies two double negation rules which do not exist in the Fitch system. These rules can be found in Appendices A and B to be compared.
Chapter 3

Software Functional Specification

3.1 Overview

The main artifact produced by this project is a proof assistant. Users interact with the proof assistant through a website called ProofAssistant.web. This chapter is a list of functional specifications that will not discuss the algorithms used by the proof engine, which will be discussed in Chapter 5. It simply discusses what the user sees when they interact with ProofAssistant.web. ProofAssistant.web is a website that helps students understand first order logic proofs more abstractly.

3.2 Screen by Screen Specification

ProofAssistant.web consists of three screens, each of which are described in this section. The screens are always accessible through their respective links on top of the web pages.

Home Screen

The Home screen (see Fig. 3.1) tells the users about the goals of the website and describes its audience as beginners eager to learn logic. It briefly introduces the capabilities of the website. It also tells the users about the Fitch and Lemmon systems and presents links to resources that will help users learn about the systems they are not familiar with.

Enter Proof Screen

The Enter Proof screen (see Fig. 3.2) consists of two text areas which allow users to enter proofs in the Fitch and Lemmon styles. The Verify button under each text area is used to
Fig. 3.1 Home screen of ProofAssistant

Fig. 3.2 Screen to enter proofs for conversion and verification
verify the respective proof. The » and « buttons are used to convert proofs in the direction indicated by the angular brackets. The buttons with symbols on top of the text areas are helpful for users to input commonly used Unicode characters in logic proofs.

The Fitch text area has separate text boxes to enter the formula and the rule used to derive the formula, or identify the formula as an assumption. Tabs are used to represent indentation, which users can be input using the Tab key.

Similarly the Lemmon text area has text boxes where users can enter the formula, the rules, and the line numbers containing the assumptions that were required to derive the formula. The Tab key does not input any character when users are working in the Lemmon text area.

Fig. 3.3 Instructions screen of ProofAssistant

Instructions Screen

The Instructions screen (see Fig. 3.3) contains the instructions that help users enter proofs. It contains three collapsible sections.

The first section describes the individual text boxes, the process of entering symbols and indentation, and the process of constructing well formed proofs. The second and third sections have rule templates for the Fitch and Lemmon systems respectively.
3.3 Scenarios

In designing the website, it helped to imagine a few real-life stories of how stereotypical students would use it. Two of these scenarios have been outlined below.

Scenario 1: Dave

Dave is an undergraduate student enrolled in COMP2600 at the ANU, or a similar offering elsewhere. He is an exceptional student with a High Distinction average and has a habit of going above and beyond the expectations of the course. When Dave learns about the existence of other natural deduction proof notations, he is keen to use them to expand his knowledge of proofs, and other proof systems.

Dave visits ProofAssistant.web and enters proofs in the Fitch system he knows from class. He converts proof to the Lemmon system. By using two systems to represent the same proof, Dave is capable of distinguishing the unique aspects of each system, and common attributes of a proof. He is then better able to see a proof as an abstract representation of natural deduction.

Scenario 2: Jess

Jess is an undergraduate student enrolled in COMP2620 at the ANU, or a similar offering elsewhere. Unfortunately for Jess, she finds it hard to understand some concepts from class, and needs extra help. She needs to complete a natural deduction assignment.

Jess visits ProofAssistant.web and uses the error checking feature for the Lemmon system. When Jess tries to discharge a line that is not an assumption, the beginner friendly software, which checks for common mistakes made by students, provides a useful error message. Jess learns that she cannot discharge lines that are not assumptions and starts to comprehend the significance of assumptions in a natural deduction proof.

3.4 Non-Goals

When working in the context of a single semester project, it was clear early on that the software would not be able to incorporate a comprehensive set of features usually desired by users. The features that would be incorporated had to be prioritized and evaluated based on the time required to complete them and the value they would deliver to students. The features the website will not support are listed below:

- Prove theorems. It is not a theorem prover.
• Proof completion. Students are expected to produce a complete proof in one of the systems. The software will not provide useful hints if the user is stuck mid-way through a proof.

• Save proofs. Registering an account and saving proofs for later are not supported.
Chapter 4

Software Technical Implementation

4.1 Overview

As mentioned in the previous chapter, the feature set of the software is not comprehensive in nature. Hence, the software had to be designed to enhance user experience while remaining modular and open to extension. This chapter details how ProofAssistant was built from a technical perspective.

4.2 Requirements

According to John’s best knowledge, and information I gained by scouring the Internet, it was apparent that we did not know of anyone else who had undertaken a similar project to convert proofs between two notations. Hence the software would have to be build from scratch. A few months ago, at the start of the project, I understood the context I was working under. The ProofAssistant project would be constructed ground up by a single programmer in a duration of 12 weeks. Also, for the ProofAssistant to deliver maximum value to students, other programmers would likely have to add features to it in the future.

In this light, one of the highest priorities when designing the technical architecture of the code base and selecting technologies to use was the readability and maintainability of the code. It was also important that the code base be open to extension without having to modify any existing code, thereby adhering to the open-close principle of software development.[8]

It was also important to make the software responsive and to ensure correctness. Not many real life constraints were imposed on the project, such as dealing with Internet bots or protection from digital denial of service attacks.
At the highest lever, the *ProofAssistant* solution is made up of three packages (layer in UML terminology), and contains a fourth deprecated package (see Fig. 4.1). Each of these packages is a project unto itself. The construction of the solution is based on a modular concept where each of these packages could be easily replaced without changing the code in the other packages. Also, additional packages (features) can be added without changing the existing structure. Each of these packages are recursively modular within, at least to some extent, and enjoy similar benefits.

The *Web Client* package contains the classes that the users see as screens on *ProofAssistant.web*. The middle *Business Logic* layer contains controller classes that drive the application behavior. The *Database* layer contains code to interact with the database. The *Forms UI* package is discussed briefly in Section 4.5 later in this chapter.

The aim while designing the software was to decouple all the packages at the highest level, and prioritize decoupling classes within at the lower levels. Unfortunately, due to a change in design (see Section 4.5), the *Web Client* layer is tightly coupled to the *Business Logic* layer. But, only small amount of code in one class needs to be changed to introduce changes to either class. The two classes only interact at one node.
4.4 Backend

Implementation Decisions

The biggest decision for the backend was deciding the technology that would be used to build it. Looking at some of the backends built by Github users who undertook proof assistant projects, it was clear that they were predominantly written using functional programming languages. But, by looking at the Git commit history, it was also clear that they were written in a short amount of time and were not updated since they were written. The *ProofAssistant* solution is meant to be used for a long time and expected to be updated in the future. To keep the code readable, maintainable, scalable and flexible, the object oriented approach to coding was chosen.

Technology

The backend is written using the C# language and targets the .Net v4.5 framework. It is supported by Microsoft with constant feature updates. C# is a fully managed language, which means that its easier to write code that doesn’t suffer from memory leaks. Microsoft also supports Visual Studio, an IDE fully integrated with the .NET framework, which makes the development and testing of code easier. The .Net framework also supports many languages, and multiple languages interact seamlessly in the same solution. These languages include both dynamic (IronPython and IronRuby) and static (C#, VB.NET, C++), both object oriented (C#, VB.NET, C++) and functional (F#).[26] Future developers can extend the project in any of these languages as per their preference.

Database

The database saves application meta data. It includes the data input by the user, system actions and the conversion results or verification results. This data is not immediately useful. Once the database build up as the users us the website, this data can be mined for useful information.

The code base expects to find a SQL database. A script for setting up the database using SQL Sever Management Studio (SSMS) has been included in the code base. A similar database can be set up using other SQL management tools like MySQL if required.

An open source tool like MySQL would suffice to work with the solution as it stands today. SSMS can be used if reporting tools or sophisticated security controls are required. SSMS has the added advantage of being part of the .Net ecosystem.
Fig. 4.2 Access to database has been made asynchronous

The backend makes no assumptions about whether the database has been set up. In fact the database has been described as optional in the installation guide. This brings up the issue of responsiveness. If a database has not been setup, the only way to find out is if the connection attempt times out. So the database transactions have been made asynchronous to make the software appear responsive if the database is inaccessible. The Task library provided by C# has been utilized for this purpose (see Fig. 4.2).

Fig. 4.3 User added to ensure database security

By default, administrative privileges are required to access the database to ensure integrity and security of the database. System administrators are unwilling to give third party applications administrative privileges. Hence, the database security has been handled by creating an explicit user, called NETWORK SERVICE, that the solution can use to connect to the database when hosted on a web server (see Fig. 4.3).

4.5 Frontend: Web Client

Implementation Decisions

At the start of the project, the Forms UI layer was based off a forms application (see Fig. 4.4) that could be installed and used locally on a personal computer. The main advantage of doing so was that the code was entirely composed of C#. But soon, it became clear that a richer
The forms interface had everything that was required for this project, but there was no guarantee of this for the future. So the UI layer was pivoted to a web based client where no local installation would be required. It would also be easier to roll out updates and bug fixes, and make the application accessible over the Internet. The modularity of the solution made it easy to replace the *Forms UI* layer.

The main focus of the web based frontend was to make it look appealing, and intuitive and easy to use.

All the UI elements scale organically with change in display resolution and display dimensions. The website also uses a minimalist design and animations for UI elements. These factors make the software look appealing.

Both the Fitch and Lemmon proof text areas have scrollable static line numbers that are displayed at all time. The text box used to enter formulas are scrollable both horizontally and vertically. Also, when a button to insert a symbol is clicked, the symbol is added to the last known cursor position of the most recently active text box. This is done by saving the id of the text box and the recent cursor position every time a text box is active and there is a change of scope (scope changes when a user interacts with a different UI element, like clicking a button). In addition, any messages displayed to the user, like error messages, make user of the browser’s inbuilt alert function. This was done to save the development effort of
designing a display box, when the browser already had an inbuilt implementation. All of these factors make the website intuitive and easy to use.

**Technology**

A mix of technologies was used to build the frontend, including JavaScript to interact with UI elements, CSS to design UI elements, HTML to format the web page, Ajax to post messages between the JavaScript and C# code, and C# to communicate with the backend. Each technology was carefully chosen and used where required for the best user experience and system performance.

This project was my first experience building a significant front end and I was happy with the overall results. After doing more research, I realized that React would be a better technology to use in place of JavaScript to interact with UI elements. React helps design UI elements that can manage their own state, and a set of these elements can be used to compose a complex UI[4].

### 4.6 Using Mixed Technologies

An interesting problem that was encountered when using a large number of technologies was getting all the languages to talk with each other. An example of this was passing data entered into UI elements to the logic layer for processing.

The proof input by users could be read by JavaScript. After which Ajax was used to communicate the data captured by JavaScript to C# code in the frontend using a json object. This communication was performed through a HTTP Post (See Fig. 4.5).

```javascript
function fitchConvert() {
  var data = {
    system: "Fitch",
    assumptionNumbers: "",
    proofText: fitchBox.value,
    ruleText: fitchRuleBox.value
  }
  var dataObj = JSON.stringify(data);
  $.ajax(
    {type: "POST",
      url: "/Home/ConversionService",
      data: dataObj,
      contentType: "application/json; charset=utf-8",
      dataType: "json",
      success: handleFitchConversion
    });
}
```

Fig. 4.5 Ajax code that does HTTP Post
The C# code in the frontend then intercepts the message using a HTTP Get. It then communicates with the code in the backend as usual by packing the data into an object and then passing objects.
Chapter 5

Proof Conversion and Verification
Methodology

5.1 Overview

This chapter presents the techniques used to verify the correctness of proofs, produce meaningful error messages, and convert proofs between the Fitch and Lemmon notations.

5.2 Parsing

The user input must first be parsed before verification or conversion can take place. The parsing process can fail for a number of reasons. For instance, if the parser comes across a symbol that does not belong in proof the parsing process will fail. In such an instance, the software returns an error message with the relevant details.

If parsing is successful, the user input is stored as a proof object. At the object level, a proof object is an ordered collection of proof lines. A proof line object is made up of a formula, a rule used to derive the formula and either the indentation number (for Fitch proofs), or the list of assumption numbers (for Lemmon proofs).

A formula is a tree made of data items. A data item can be one of an atom, a connective or a quantifier. The root of the tree contains the leading connective or quantifier, or an atom if the formula has no connectives or quantifiers. This is done to avoid parenthesis in the formula, and make the processing of the formula easy.

From above, it is obvious that the proof object data structure contains different information at different levels of abstraction. This is ultimately useful because different stages of proof verification and conversion only require some of the information and not all of it. This
layered abstraction approach helps the code identify the required information easily. Object oriented programming enabled this process of abstraction and encapsulation.

5.3 Proof Verification

The process of proof verification includes three stages. They are identifying the rule used, checking if the rule is applied correctly, and checking for correct indentation (in case of Fitch proofs), or checking for correct assumption lines (in case of Lemmon proofs).

Identifying Rules Used

Both the Fitch notation, and the Lemmon Notations have a limited number of rules to choose from. Hence, the process of rule identification is a process of pattern matching. C# provides a powerful regular expression engine that was used for pattern matching. The use of this regex engine produced compact and readable code.

When the right rule has been identified, it is placed into an object meant to hold that particular rule. The object has the required properties. In the above example, for the not elimination rule, a NotElimRule has the properties corresponding to the disjunction line, and the two sub proofs. This makes extracting the required information from a rule much easier when required.

Verifying Rules

Once the rule was identified, the process of checking involved looking at the rule and lines referenced in the rule. For each of these lines, the code checks if the referenced lines contain the expected formulas. For instance in the picture below (See Fig. 5.1), the rule was identified to be the AndElimRule. The code the looks into the line referenced and checks if the formula appears as a subtree on that line and its parent is the $\wedge$ symbol. In case assumptions are discharged, the correctness of this is also verified along similar lines.

Indentation and Assumptions

After checking the rule, the code checks calculates either the indentation of each line (for Fitch proofs), or the list of assumption numbers for each line (for Lemmon proofs), and verifies this calculation against the user input.

Expected indentation is calculated as follows:

- Each line starts with indentation of 0.
5.4 Proof Conversion

Fig. 5.1 Code that checks the correct application of $\land$ elimination

- Each time an assumption is discharged, indentation of all lines in the relevant subproof leading to the discharged assumption is incremented by one.

Expected assumption line list is calculated as follows:

- Assumption line list of a line whose rule is AssumptionRule contains one entry, its own line number.

- For all lines whose rule is not AssumptionRule, the line list contains the union of line lists of all lines referenced in the rule that was applied, minus discharged assumptions.

**Displaying Error Messages**

It was important to produce useful error messages to display in case the rule verification is unsuccessful.

Custom error messages were crafted for errors that were likely to be committed by students. These likely errors were educated guesses on the part of John Slaney and me. More generic error messages will be displayed for other errors.

### 5.4 Proof Conversion

The process of proof conversion involved the construction of a proof tree from the proof object, handling notation agnostic rules, converting the tree back to a proof object, and setting the indentations and assumption like lists.
Build a Proof Tree

The best way to go about proof conversion was to build an abstract data structure to hold the proofs. The same tree representation used to hold a formula (tree of data items) has been employed to construct a proof tree. The root element of the tree is the last line of the proof, and the children of any node will be those proof lines the node has a direct dependency on. By extension, the leaf nodes will be assumptions. The figure above shows the proof tree for a conjunction of two assumptions (See Fig. 5.2).

```plaintext
/// <summary>
/// 1 A Ass
/// 2 B Ass
/// 3 A\&B \&-I, 1, 2
/// is transformed to
/// A\&B
/// A    B
/// A    B
/// </summary>
```

Fig. 5.2 Comment on function that constructs proof trees

This abstract data structure was designed to hold proofs of both notations. Each notation has its own set of constraints. For example, subproofs can be interleaved in the Lemmon notation, but they have to be separate in Fitch. The constraints of the proof tree structure is the union of the constraints of both notations. To convert proofs from one system to another, the tree will have have to be massaged to meet the constraints of the other system. This process is made easy by using the same data structure to hold proofs of both notations.

This notion of abstract representation of proofs can be used to easily extend the feature set of ProofAssistant. A new proof notation can be added to the mix, and to convert proofs between all three of these notations, code has to be written to convert from the new notation to the abstract tree, and from an abstract tree to the new notation. The constraint set of the existing tree structure can be increased without affecting the existing functionality.
Handle Notation Agnostic Rules

In some cases, such as when encountering notation agnostic rules, merely massaging the tree will not yield the correct proof. Instead, new tree nodes will have to be generated to replace existing ones.

Double Negation in Lemmon

The $\neg\neg$ elimination and introduction rules found in the Lemmon notation do not have analogous counterparts in the Fitch notation. Hence, instances of these rules in the Lemmon proof would have to be replaced by a subproof in the Fitch proof.

Examples of double negation rules in Lemmon, and their replacements proofs are shown below.

### Double Negation Introduction

1. (1) $P$ Assumption
2. (2) $\neg\neg P$ $\neg\neg$-Intro, 1

<table>
<thead>
<tr>
<th>Step</th>
<th>Formula</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$P$</td>
<td>Assumption</td>
</tr>
<tr>
<td>2</td>
<td>$\neg P$</td>
<td>Assumption</td>
</tr>
<tr>
<td>3</td>
<td>$P \land \neg P$</td>
<td>$\land$-Intro, 1, 2</td>
</tr>
<tr>
<td>4</td>
<td>$\neg\neg P$</td>
<td>$\neg$-Intro, 2-3</td>
</tr>
</tbody>
</table>

### Double Negation Elimination

1. (1) $\neg\neg P$ Assumption
2. (2) $P$ $\neg\neg$-Elim, 1

<table>
<thead>
<tr>
<th>Step</th>
<th>Formula</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\neg\neg P$</td>
<td>Assumption</td>
</tr>
<tr>
<td>2</td>
<td>$P$</td>
<td>$\neg\neg$-Elim, 1</td>
</tr>
</tbody>
</table>
\[1 \quad \neg
eg P \quad \text{Assumption}
\]
\[2 \quad \neg P \quad \text{Assumption}
\]
\[3 \quad \neg
eg P \land \neg P \quad \land\text{-Intro, 1, 2}
\]
\[4 \quad P \quad \neg\text{-Elim, 2-3}
\]

In the tree structure, two new tree nodes are fabricated and inserted in the right position.

**Contradiction**

The negation rules in both the notation require a contradiction. Contradiction in a Fitch proof is represented by the conjunction of the two formulas that contradict each other, derived using the \(\land\) introduction rule. In a Lemmon proof, it is represented using the \(\bot\) symbol, and the rule used is called the \(\neg\) elimination rule that references the two lines containing the contradicting formula.

When converting between systems, both the formulas and the rules will have to be swapped out. In the tree structure, the required tree node is fabricated and the offending tree node is replaced with the fabricated one.

**Negation Elimination in Fitch**

The \(\neg\) elimination rule in a Fitch proof discharges a not symbol from a assumption if that assumption is part of a contradiction. This rule does not have an analogous counterpart in the Lemmon notation, and hence, all instances of this rule would have to be replaced by a subproof in the Lemmon proof. An example is shown below.

\[1 \quad P \land \neg P \quad \text{Assumption}
\]
\[2 \quad \neg Q \quad \text{Assumption}
\]
\[3 \quad P \land \neg P \quad \text{Repeat, 1}
\]
\[4 \quad Q \quad \neg\text{-Elim, 2-3}
\]
In the proof tree, some tree nodes will have to be modified while others replaced entirely.

Fig. 5.3 Formula and assumptions are may have to be derived more than once in the Fitch notation.
Conversion to Proof Object

When converting from Lemmon proofs to Fitch proofs, the complexity lied in setting the right line numbers. This conversion process introduced duplicate lines in the Fitch proof because in the Fitch notation (unlike in Lemmon), the same formula cannot be used again from another scope. It has to be derived or assumed afresh. In Fig. 5.3, in the Fitch proof, the formulas on lines 3 and 7 are the same, and so are the assumptions on lines 2 and 6.

The issue lied with the fact that in the proof tree, duplicate formulas were in fact references to the same object. So when printing line numbers (which are generated dynamically), sets of duplicate formulas on different lines would appear with the same line numbers. The solution was to identify and clone these tree node objects before converting the tree to a proof object. This cloning was performed by serializing the bit stream representing the object in memory and copying it across to a new destination address in memory.

Similarly, when converting from the Fitch notation to Lemmon, these duplicate tree nodes had to be removed before transforming the tree back to the proof object. The process of removing nodes involved dereferencing one of the memory locations containing duplicate data.

Set Indentation and Assumption Numbers

The indentation (for Fitch proofs) and assumption numbers list (for Lemmon proofs) were calculated in a similar fashion to that indicated in Proof Verification section (Section 5.3) earlier.
Chapter 6

Testing and Evaluation

6.1 Overview

This chapter details the process used to determine the performance and correctness of the ProofAssistant software.

The data represented in this chapter was collected on a laptop computer. It is configured with an Intel i7-5500U low power CPU and DDR3 ram. While this test bed has a fraction of the computing power than that expected from a website hosting server, the following statistics are meant to demonstrate that the website performs adequately even on low power hardware.

6.2 Performance Evaluation

Visual Studio, the IDE primarily used to write the code, possesses a collection of handy debugging tools there were used to derive the statistics listed below.

- Average time taken to verify a 10 line proof - 28 milliseconds.
- Average time taken to convert a 10 line proof - 32 milliseconds.
- Average time taken for a database write transaction - 63 milliseconds.

The Logic Notes website provides exercise problems for students to practice. The solutions to these problems usually range from about 10 to 20 lines. The performance numbers above look reasonable for proofs of this size. This data shows that the program does not suffer from performance bottlenecks.
Database Transactions

In the above mentioned statistics, the database transactions run independently from proof conversion and verification, and not part of the sequential execution of code.

If the database has not been setup, or is otherwise inaccessible, the only way to find out is if the connection attempt times out. It is advisable to set the connection timeout to a few seconds (the default timeout is set to 30 seconds).[16] If the database transactions are part of the sequential execution, and the database is inaccessible, users will have to wait a few seconds every time before the software responds to their request. Conventional wisdom in software engineering says that if users are made to wait for more than 4 seconds before a web page responds, they tend to get frustrated.[18] Database transactions are made asynchronous for best performance from the point of view of the user.

Memory and CPU Allocation

![Graph showing resource allocation](image)

Fig. 6.1 Allocation of memory and CPU to ProofAssistant

The graph (see Fig. 6.1) shows the resource allocation for the program when run under Visual Studio debug mode.

In terms of memory use, the application quickly consumes 136 MB of data and stays there, even when verifying and converting proofs. This shows the absence of memory leak in the application. Most of the 136 MB is used up to spawn a new Google Chrome tab to render the website.

The CPU usage on the graph comes from the new instance of Google Chrome that is spawned. Otherwise, the application does not consume a measurable amount of CPU time.
6.3 Testing for Correctness

Unit Testing

Black box unit testing of parsing, proof verification, proof conversion, and database transactions was done to ensure the correctness of the code. Unit testing was a cheap way to ensure maximal code coverage with minimal effort. Integration tests would have to be written to test the synchronous behavior of the Web Layer, the Business Logic Layer and the Database Layer together. These tests are much harder and time consuming to design, and hence I settled for unit testing and some sanity checks that are explained below.[15]

Sanity Checks

There were also some sanity checks performed to ensure that the software worked as advertised. Firstly, proofs were taken from the exercises (See Fig. 6.2) of the Logic Notes website (which has the types of problems students usually solve) and manually input these proofs and checked for correctness.

```
<table>
<thead>
<tr>
<th>Proof exercises</th>
<th>Propositional Natural Deduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p ∧ q) ∧ r ⇒ r'</td>
<td>Answer</td>
</tr>
<tr>
<td>(p ⇒ q) ⇒ (q ⇒ p)</td>
<td>Answer</td>
</tr>
<tr>
<td>(p ∧ q) ⇒ r' ⇒ (p ∧ r) ⇒ q</td>
<td>Answer</td>
</tr>
<tr>
<td>p ⇒ (q ⇒ r) ⇒ (r ⇒ q) ⇒ (¬q ⇒ ¬p)</td>
<td>Answer</td>
</tr>
<tr>
<td>p ∧ q ⇒ (p ⇒ q) ∧ (q ⇒ p)</td>
<td>Answer</td>
</tr>
<tr>
<td>¬(p ∧ q) ⇒ p ∨ q</td>
<td>Answer</td>
</tr>
<tr>
<td>p ⇒ ¬q ⇒ (p ∧ q) ⇒ ¬r</td>
<td>Answer</td>
</tr>
<tr>
<td>p ∨ q ⇒ ¬q ⇒ q</td>
<td>Answer</td>
</tr>
<tr>
<td>¬q ⇒ ¬q ⇒ p ∨ q</td>
<td>Answer</td>
</tr>
<tr>
<td>¬(p ⇒ q) ⇒ p ∧ q</td>
<td>Answer</td>
</tr>
<tr>
<td>(p ⇒ q) ⇒ (¬q ⇒ ¬p) ⇒ p</td>
<td>Answer</td>
</tr>
<tr>
<td>p ⇒ (q ⇒ r) ⇒ (q ⇒ ¬p) ⇒ r</td>
<td>Answer</td>
</tr>
<tr>
<td>(p ∧ q) ⇒ (r ∨ s) ⇒ (¬r ∨ ¬s)</td>
<td>Answer</td>
</tr>
</tbody>
</table>

Fig. 6.2 Logic Notes sample exercise proofs
```

Other sanity checks included performing experiments such as taking a proof in the Fitch notation, converting it across to the Lemmon notation, and then back to Fitch again to check if the proof generated was the same as the one at the start.
Error Log

The software comes with an error logger in case the program has a bug that throws an exception or an error. The error message is stored along with the stack trace that is responsible for the error and the date-time stamp of when the error was encountered. This feature was added to help record and debug program crashes.
Chapter 7

Limitations and Future Work

7.1 Overview

This chapter outlines some of the deficiencies of the current approach to helping students with natural deduction, and suggests improvements and potential extensions.

7.2 Support Quantifiers

Despite setting out to support both connectives and quantifiers, the proof verification and conversion process could not be completed for quantifiers. The code for input and parsing already exists, along with scaffolding for proof verification and conversion. The relevant classes will need to be extended to support these features.

7.3 Investigate Real Sources of Error

It was mentioned earlier that potential sources of error committed by students were contrived by John and I. This approach runs the risk of not being representative of real errors committed by students while practicing proofs.

A better approach to useful error messages could be to investigate real sources of errors students commit in assignments and exams. A good extension to ProofAssistant could involve combing through the answers students produce for assignment and exam questions to find common errors and investigate their cause. If new errors were discovered, checks for these could be added to the code base, and the result of the investigations can be used to produce more helpful error messages than those that currently exist.
The reason for not incorporating this approach in the original project is that this would have been a substantial and complex project in its own right, so it was not pursued within the scope of the present work. It would in any case have required ethics committee clearance because of the human subjects involved. According to the ANU, this process is also time consuming.[25]

7.4 Use Logged Data

The database, as mentioned in previous chapters, is configured to save all the data that flows through the system, including user input, error messages if any, and converted proofs. If enough data is collected, this data has enormous potential to increasing the quality of error identification and the error messages displayed as a response.

It could become easier to identify patterns students are using to solve proofs. These patterns could be used to identify potential sources of errors. This data could also be used in the design of teaching material.

Another extension to the project could be the addition of user logins and tracking user inputs. Then past data could be used to influence future behavior of the system. For example, if students can’t solve the issue immediately after receiving an error message, it would be apparent that the error message would have to be improved.

7.5 Migration from .Net to .NET Core

The software stack used to produce ProofAssistant targets the .Net framework. The .Net framework is a proprietary framework that is officially supported on Microsoft Windows.

The .Net Core framework in contrast is an open source, cross-platform framework that works on Windows, OSX and some Linux distributions. It has support for common libraries across the different platforms and is also supported my Microsoft. It is more container technology friendly as most container providers use Linux. It also works with the Kestrel server, a across platform web server that is part of the ASP .NET framework.[13]

The reason .Net Core is not already being used is that it is a new entry into this space. In fact official support for .Net Core 1.0 first arrived with Visual Studio 2017, which was released after the project started, and at the time contained some preproduction quality code.[14] Also, x86 support for non-windows platforms was not ready yet.[12]

.Net Core is compatible with the .Net Framework[14] and this project does not use any of the features that are not supported by .Net Core. So, migration between the frameworks will be a relatively uncomplicated process.
References


Appendix A

Fitch Notation Rules

Propositional Calculus

\[(\land I) \quad \frac{p \quad q}{p \land q} \quad \quad (\land E) \quad \frac{p \land q}{p} \quad \frac{p \land q}{q} \quad \frac{[p]}{[q]} \quad \frac{\vdots}{\vdots} \quad \frac{r}{r} \quad \frac{r}{r} \]

\[(\lor I) \quad \frac{p}{p \lor q} \quad \quad (\lor E) \quad \frac{\vdots}{\vdots} \quad \frac{q \lor p}{q} \quad \frac{q \lor p}{p} \]

\[\rightarrow (I) \quad \frac{q}{p \rightarrow q} \quad \quad (\rightarrow E) \quad \frac{p \quad p \rightarrow q}{q} \quad \frac{[\neg p]}{\vdots} \quad \frac{\neg p}{p} \]

\[\neg (I) \quad \frac{q \land \neg q}{\neg p} \quad \quad (\neg E) \quad \frac{q \land \neg q}{p} \]

Predicate Calculus

\[\forall (I) \quad \frac{P(a)}{\forall x. P(x)} \quad (a \text{ arbitrary}) \]

\[\forall (E) \quad \frac{\forall x. P(x)}{P(a)} \]

\[\exists (I) \quad \frac{P(a)}{\exists x. P(x)} \]
Appendix B

Lemmon Notation Rules

\[
\frac{A \land B}{A} \quad \land E \\
\frac{A \land B}{B} \quad \land E \\
\frac{A \rightarrow B}{B} \quad \rightarrow E \\
\frac{X, A \vdash B}{X \vdash A \rightarrow B} \quad \rightarrow I
\]

\[
\frac{X \vdash A \lor B}{X, Y \vdash C} \quad \lor E \\
\frac{A \lor B}{A \lor B} \quad \lor I \\
\frac{A}{A \lor B} \quad \lor I \\
\frac{A \lor B}{A} \quad \lor I \\
\frac{A}{A \lor B} \quad \lor I
\]

\[
\frac{\neg A}{\bot} \quad \neg E
\]

\[
\frac{X, A \vdash \neg B}{X, Y \vdash \neg A} \quad \neg A
\]

\[
\frac{X \vdash A}{\neg \neg A} \quad \neg \neg A
\]

\[
\frac{A \lor A}{A} \quad \lor I \\
\frac{A \lor A}{A} \quad \lor I \\
\frac{X \vdash A}{X \vdash \forall v A_m} \quad \forall I
\]

\[
\frac{X \vdash \exists v A_m}{X, Y \vdash B} \quad \exists E \\
\frac{X \vdash A}{A^*_v} \quad \exists I
\]

\[
\frac{X \vdash A^*_v}{X \vdash \forall v A} \quad \forall E
\]
Appendix C

Independent Study Contract

INDEPENDENT STUDY CONTRACT

SECTION A (Students and Supervisors)

Unit: 5822888
Surname: NAQAPATIKA
First Name: SIBHEDZI
Project Supervisor: [Insert name]
Course Supervisor: [Insert name]
Course code, title and unit: COMP4560, 12 units
Semester: 1
Year: 2017
Project Title: Natural Deduction Theorem Proving

Learning Objectives:
- Understanding of proof and proof search at both abstract and concrete levels
- Experience in project based research and communication of results

Project Description:
Produce a theorem prover capable of searching for natural deduction proofs and presenting them in different formats including Fitch and Leibniz styles, and transforming these objects into each other. Investigate the use of this software in helping students with the process of learning to construct such proofs. Help can include recognition of users’ proof strategies, generation of hints and of explanations.
### ASSESSMENT (as per course’s project rules web page, with the differences noted below):

<table>
<thead>
<tr>
<th>Assessed project components</th>
<th>% of mark</th>
<th>Due date</th>
<th>Evaluated by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report: name style</td>
<td>60 (60%)</td>
<td>May 2017</td>
<td></td>
</tr>
<tr>
<td>Artifact: name kind</td>
<td>20 (30%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation</td>
<td>10 (10%)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### MEETING DATES (IF KNOWN):

- Date:

### STUDENT DECLARATION: I agree to fulfill the above defined contract:

- Signature:
- Date:

### SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student’s academic record and believe the student can complete the project.

- Signature:
- Date:

### REQUIRED DEPARTMENT RESOURCES:

- Date:

### SECTION C (Course coordinator approval)

- Signature:
- Date:

### SECTION D (Projects coordinator approval)

- Signature:
- Date:

Research School of Computer Science  

*Form updated Jan-18*
Appendix D

README.md from Code Repository

The README file contains the information required to setup and use the software on a local personal computer.
TheoremProver

ANU COMP4560 project.

Targets the .Net v4.5 framework.

ProofSolver.web is a web based software tool the is meant to be hosted on a server, and accessed online through a web browser.

Nonetheless, here are the instructions to install and use the software on your local Windows PC. This guide will take you through building the code and hosting the Proof Assistant website on your local Windows machine (optional).

Prerequisites

Operating system - Microsoft Windows 7 or later.

Tool to compile the solution - Microsoft Visual Studio (Community/Enterprise/Professional).

(Optional) Tool for hosting database - Microsoft SQL Server Express

(Optional) Tool for creating and managing database - Microsoft SQL Server Management Studio

Use the software locally

1. Download and extract the solution to TheoremProver directory on your machine. It is recommended that you do not download the code into your home directory, or subdirectories of your home directory (like Documents ). Not all tools we use below will have access to your home folder. Use a path in your C directory like C:\Code .

2. Open TheoremProver.sln from TheoremProver directory in Visual Studio.

3. In Visual Studio, click on Build menu, click on Build Solution . Ensure that the code compiles without any build errors.

4. To Run the website, click the run button (green arrow).

Note: This method runs the code within the Visual Studio debugger framework. While it is useful to test the functionality of the software, it is not a true reflection about its speed and performance.

Optional: Host the website on your local machine

Why you may want to do this

To get a true idea about the performance of the software.

Prerequisites

Ensure you have built the code following the instructions above.

Turn on the required Windows features
1. In Windows, go to Control Panel. Search for 'Windows features'.

2. Select the Turn Windows features on or off option.

3. In the window that opens, under Internet Information Services, select Web Management Tools and World Wide Web Services.
   - For Internet Information Services, the default selection will do.
   - But for World Wide Web Services, select the check boxes as indicated below.


6. Click **OK**.

7. Restart your computer.

**Install website**

Internet Information Services (IIS) is the tool we will use to host the website on your local machine.

1. Open the IIS Manager application.
   - In Windows 10/8.1/8, type IIS Manager into search to find it. In Windows 7, find it under **Start** -> **System and Security** -> **Administrative Tools**.
   - By default it will be installed as **InetMgr.exe** in `C:\WINDOWS\system32\inetsrv` if you need to manually locate the executable.

2. On the left hand **Connections** pane, expand the entry with your computer name.

3. Right click the sites folder -> **Add Website**.

4. Set the following fields:
   - **Site Name**: *ProofAssistant*
   - **Physical Path**: *TheoremProver/WebLayer*. This is the code you downloaded.
   - Ensure **Type** is set to **HTTP**.
   - **Host Name**: *proofassistant.web*

5. Click **OK**.

**Use the software**


**Troubleshooting**

- If your browser displays a DNS not found error,
  - 1. Open **Notepad** as administrator (important).
  - 2. File -> Open
  - 3. In file name bar, paste `C:\Windows\System32\drivers\etc\hosts`. Click **Open**.
Optional: Set up database on your machine

Setup the database if you want to log user input, and the output generated by the system.

Prerequisites

You have installed SQL server and SQL Server Management Studio (SSMS). You have built the application.

Instructions

1. Launch SSMS as Administrator (will not work otherwise). In the Connect to Server dialog box, set Server name: to . (to represent local machine). Click connect.

2. File -> Open navigate to the directory containg the soource code you downloaded. Open TheoremProver\CreateProofAssistantDB.sql

3. Click on the execute button (Green Arrow). You will get an error if your SSMS version is not v17, or if you did not use the default install settings. If so, you will have to change the filename address to the right directory.

Usage

1. Open Visual Studio as Administrator (will not work otherwise).

2. You can now use the database when you lauch your application through Visual Studio by clicking on the Run Button (Green Arrow). If you have set up your website through IIS, read on.

Giving IIS website access to the database

Currently, the application needs Administrator privilages to access the database. So the IIS website does not have permission to access the database. You can still use the website, but it will not log any data. The following instructions will give your website access to the database.

Modified Instructions from here If something goes wrong, you can find more verbose instructions by folloing the link.

Configure IIS

1. Open IIS Manager

2. Select Application Pools on the left and locate the application pool ProofAssistant . From there, right-click and select Advanced Settings.

3. Scroll down in the list until you see the setting Identity . Click the ... button and select the Network Service account. This will allow all the permissions you need without exposing your system too much. Now click ok . You’ve now configured the application pool.

4. Expand Sites on the left and select the ProofAssistant website. Double-click on the Authentication tile on the middle panel. Make sure that Anonymous Authentication is Enabled and the rest Disabled , Right-click on Anonymous Authentication and select Edit . Change the identity to Application Pool Identity.

Configure Database

1. Open SSMS as Administrator. Connect to local database.


3. a. At the top, in the Login Name field, enter NT AUTHORITY\NETWORK SERVICE.

     b. Select the User Mapping tab on the left. Tick the ProofAssistant database.

     c. In the schema column for the database, click ... button. Enter dbo in the text box and click OK.

     d. In the Database role membership section, set permissions for db_datareader, db_datawriter and public. Click OK.
4. Expand Databases on the left. Expand ProofAssistant database. Expand Security and expand Users. The NT AUTHORITY\NETWORK SERVICE user you just added should be visible. Right-click and Properties for this user.

5. Select the Securables tab on the left. Click on the Search button.
   a. Select Specific Objects and click OK. Click the Object Types button. Scroll down and tick Schemas. Click OK.
   b. In the textbox below, enter the schema dbo. Click OK.
   c. At the bottom, select all the permissions you want to give for that database. For futureproofing, select Alter, Control, Create Sequence, Delete, Execute, Insert, References, Select & Update.
   d. Click OK. The website proofassistant.web loaded through your favourite browser should now have access to the database.

Notes

The web code has been designed to work best on Google Chrome.

In case you get an Internal Error!!! message when using the software, the error with stack trace is logged in C:\Users\Public\TheoremProver_ERRORLOG.txt on your PC.