Imperial College
London

Imperial College London

Department of Computing

LogicAssistant

Author:
Joshua Zeltser

Supervisor:
Romain Barnoud

Submitted in partial fulfillment of the requirements for the BEng degree in Computing of Imperial College London

September 2015

Contents

1 Introduction

1.1 Motivation

Mathematical logic is an area used throughout the engineering and scientific industries. Whether its developing artificial intelligence software or students completing a Computer Science degree, logic is a fundamental tool. In order to ensure that logic is used correctly a proof system must be used. Natural Deduction provides the tools needed to deduce and prove the validity of logical problems, making it a vital tool for everyone to learn to use. This is why many universities make it a priority to teach this to their students as they begin their studies. For students new to Natural Deduction or even those more advanced users are often left stuck in the middle of a proof not knowing what to do next, and then when they have completed the proof are unsure as to whether it is valid. LogicAssistant is being created to assist users with this problem.

Although the basics of Natural Deduction are easy to learn by just committing to memory the various rules that are defined, when it comes to more complex proofs it can be difficult to work out whether your proof is correct. You may have made an assumption at some point in your proof that was invalid, or just mistakenly used the wrong rule. It is not always the case that you can find someone who is an expert in this field who can check over your proof for you. This problem assumes that you are even able to get to the point of completing your proof. Often you may just have to sit staring at a certain point in your proof unsure of what to do next. You could at this stage go through each rule and try to work out which fits best, but this is quite a short-sighted approach as you may be going off in the complete wrong direction. This is also quite time consuming. When using Natural Deduction in the real world there is not always someone around to point out which rule to use next.

In order to remedy these problems I am building LogicAssistant which will have a number of useful features that will aid users with their Natural Deduction proofs. The first feature which I aim to create is a proof checker which allows the user to type in their Natural Deduction Proof specifying all of the rules they have used, and the software will notify them whether it is valid. I also want the user to be notified which parts of the proof are wrong and are thereby causing the proof to be invalid. This means that a user will never be unsure whether a proof they have created is valid and if it isn't they will know immediately where they went wrong. The other main feature that LogicAssistant will have is the ability to ask for hints at any stage of a proof. This means that if a user is stuck at any point in their proof they can ask the software to provide them with the next rule to apply. This will be very helpful especially for students who are new to Natural Deduction proof techniques. All of these features and more will be condensed together into an easy to use web interface that a user can access whenever they need to check or solve a proof.

There are tools already out there that assist you with Natural Deduction. Most of these tools are very complex and difficult to use, requiring a high technical and mathematical knowledge to understand them. This makes them often a waste of time to use on a simple Natural Deduction proof. There are also tools like Pandora[1] which allows you to enter your Natural Deduction proof step by step and tells you whether it is valid. The drawback of this software is that it only allows you to enter a step in the proof if it is valid. This means that the amount that can be learnt from this tool is limited as through trial and error you could eventually solve your proof without really knowing what you are doing. The motivation of LogicAssistant is to fill in these gaps left by existing solvers to help solve Natural Deduction proofs.

1.2 Objectives

When starting this project I came up with a number of objectives that I wanted to achieve. Over time some of these may change depending on the level of difficulty or ease that I experience during different parts of the project. The first and main objective of this project was to create an easy to use proof checker for Propositional Natural Deduction proofs. A tool that can be used by anyone to easily check whether their Natural Deduction proofs are valid and if not they will be notified as to whether they have gone wrong. Once this was complete my next objective was to add on top of this a hint option which offers the user a hint as to the next step when they are stuck at any part of the proof. These were my two main aims that I wanted to achieve by the end of the project.

In order to ensure that my project is as easy to use and as useful as possible I also added the following additional objectives that time allowing I would like to complete:

- Add proof checks for first order predicate logic Natural Deduction
- Add hints for proofs using first order predicate logic
- Add live error correction
- Create an aesthetically pleasing user interface
- Add the ability to save and load Natural Deduction proofs

2 Background

2.1 Propositional Logic

A Proposition is a statement of some alleged fact which must be either true or false, and not both [2]. This basic form of logical reasoning can be used to represent everyday scenarios that we may face. Although this system has issues when trying to represent time and context this system can be used to represent some quite complex systems.

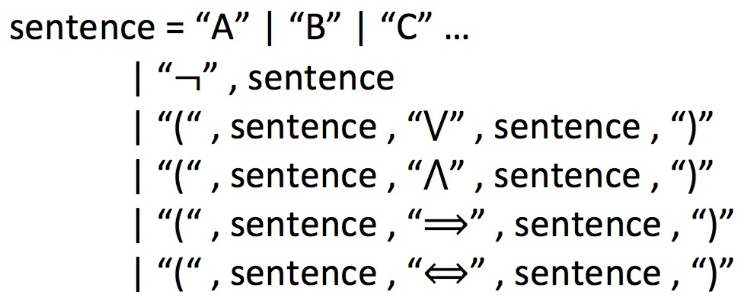
Example 2.1. A list of propositions that could be represented by Propositional Logic

- England is in Europe
- Elephants are big
- three is smaller than two
- Mark smells because he is muddy

Throughout this report I will represent propositions using a capital letter or a word starting with a capital letter. I will also use a number of operators to represent logical statements:

- Logical Truth will be represented by $\top$
- Logical False will be represented by $\bot$
- A logical And will be represented by $\land$
- A logical Or will be represented by $\lor$
- A logical Not will be represented by $\neg$
- A logical Implication will be represented by $\Rightarrow$
- A logical Iff (if and only if) will be represented by $\Leftrightarrow$

Propositions can be grouped together to form sentences using these operators. This is how Propositional Logic is used to model real life scenarios made up of events represented by propositions. The syntax and grammer of a sentence in Propositional Logic is defined in Figure 1.



```
sentence = “A” | “B” | “C” …
           | “¬” , sentence
           | “(“ , sentence , “V” , sentence , “)”
           | “(“ , sentence , “Λ” , sentence , “)”
           | “(“ , sentence , “⟹” , sentence , “)”
           | “(“ , sentence , “⟺” , sentence , “)”
```

Figure 1: Propositional Logic Grammar Rules

We have now defined a Propositional Logic language for this project. We have added brackets into our rules in order to ensure that we fully understand what a Propositional statement is saying. Without brackets some statements may become a bit ambiguous and difficult to understand. Truth tables can be used to interpret all of the possible results of a logical sentence. This helps users to plan for any possible outcomes of a set of propositions occurring.

Example 2.2. Examples of Propositional Statements using our language

- $\mathrm{A} \Rightarrow \mathrm{B}$
- $\mathrm{B} \wedge \mathrm{A}$
- $((\mathrm{A} \Leftrightarrow \mathrm{B}) \vee \mathrm{C})$
- $\neg\mathrm{A} \Rightarrow (\mathrm{B} \Rightarrow \mathrm{C})$

2.2 Predicate Logic

Propositional logic is very useful when trying to represent the truth values of propositions, but there is limited scope in this formalisation as to the type of statements that can be represented. For example if I wanted to represent the statement "some boys like football", it would be very difficult to represent using just Propositional Logic. This is why I decided to also check proofs of the more useful first order Predicate Logic. This logical system allows users to both apply properties to specific objects and add quantifications to them. This is much more expressive and useful in real world applications.

The properties that an object has is called a Predicate. Predicates can be unary where they have just one property, or they can be n-ary describing relationships between them and other predicates. Predicate logic also includes two quantifiers that allow this formalisation to be more expressive. The first quantifier is a 'for all' quantifier ($\forall$). This allows us to represent statements like all dragons are green. The other quantifier is the 'exists' quantifier ($\exists$). This quantifier allows us to represent statements such as some dragons are blue. These two quantifiers together, increase the expressive power we can create using logic. Below are some examples of statements that can be created using Predicate logic.

Example 2.3. Examples of Predicate Statements using our language

- $\forall x.P(x)$
- $\exists y.bird(y)$
- $\forall y.dark(y) \Rightarrow hates(x, y)$

2.3 Natural Deduction

The logical systems that were introduced above present a language and an interpretation using Propositional and Predicate atoms. All the reasoning that can be taken away from these systems are based on truth tables. Truth tables are a way of considering all of the possible outcomes of a statement filled with atoms that can be true or false. For simple systems truth tables are enough to be able to prove and reason about a situation. In order to be able to reason about much more complicated problems, the truth tables would be very large making it a time consuming problem to solve. We must therefore introduce a formal system which gives us the tools to solve and deduce these problems. To solve this problem Natural Deduction was created. Using a set of strict rules Natural Deduction allows us to deduce the consequences of any Propositional or Predicate statements we are given showing us the immediate outcome. This apparatus is very useful in solving logical problems in all areas. When a statement can be deduced from a number of premises using Natural Deduction we put it on the right side of a $\vdash$ symbol, while the original premises will be listed on the left of this symbol. Natural Deduction has a list of rules for both Propositional and Predicate logic allowing us to deduce the consequences of any premise.

2.3.1 Propositional Rules Examples

There are ten Propositional rules for Natural Deduction that can be used to form proofs (see Appendix A.1). Throughout this project I have used the Gentzen Natural Deduction system. This means that I lay out proofs as shown in the examples below. The first example shows a basic proof with the given premise written first followed by the various deductions made throughout the proof. On each line of the proof I write a justification of the rule that was used together with the lines that the rule has used to make the deduction.

Example 2.4. Show that $A \wedge B \vdash A \vee B$

1	$A \wedge B$	
2	A	$\wedge$-Elim (1)
3	$A \vee B$	$\vee$-Intro (2)

The next two examples require an assumption to be made at some point in the proof that will allow the user to deduce their required outcome. Throughout this project when an assumption occurs in a proof I will highlight the assumption and its result by formatting it slightly to the right of the rest of the proof. This makes it stand out and more readable for the user.

Example 2.5. Show that $A \Rightarrow B \vdash \neg(A \wedge \neg B)$

1	$A \Rightarrow B$	
2 $\triangleright$	$\quad A \wedge \neg B$	Assumption
3	$\quad A$	$\wedge$-Elim (2)
4	$\quad B$	$\Rightarrow$-Elim (1)
5	$\quad \neg B$	$\wedge$-Elim (2)
6	$\neg(A \wedge \neg B)$	$\neg$-Intro

Example 2.6. Show that $A \Rightarrow B, B \Rightarrow C \vdash A \Rightarrow C$

```
1    | $A \Rightarrow B$
2    | $B \Rightarrow C$
     |----
3 ▷  | | $A$                  Assumption
4    | | $B$                  $\Rightarrow$-Elim (1,3)
5    | | $C$                  $\Rightarrow$-Elim (2,4)
6    | $A \Rightarrow C$      $\Rightarrow$-Intro (3-5)
```

Example 2.7. Show that $A, A \Leftrightarrow B \vdash B$

```
1    | $A$
2    | $A \Leftrightarrow B$
     |----
3    | $A \Rightarrow B$      $\Leftrightarrow$-Elim (2)
4    | $B$                    $\Rightarrow$-Elim (1,3)
```

2.3.2 Predicate Rules Examples

In Predicate Logic similar Natural Deduction proofs can be created. These proofs as well as using the Propositional rules, have their own set of rules (see Appendix A.2) that the user can use to form a proof. This means that very complex proofs can be created using these rules. Below are some examples of what these proofs may look like.

Example 2.8. Show that $P(m), \forall x.(P(x) \Rightarrow Q(x)) \vdash Q(m)$

```
1    | $\forall x.(P(x) \Rightarrow Q(x))$
2    | $P(m)$
     |----
3    | $P(m) \Rightarrow Q(m)$        $\forall$-Elim (1)
4    | $Q(m)$                         $\Rightarrow$-Elim (2,3)
```

Example 2.9. Show that $\forall x.\forall y.P(x, y) \Rightarrow \neg P(y, x) \vdash \forall x.\neg P(x, x)$

```
1    | $\forall x.\forall y.P(x,y) \Rightarrow \neg P(y,x)$
     |----
2    | $\forall y.P(a,y) \Rightarrow \neg P(y,a)$      $\forall$-Elim (1)
3    | $P(a,a) \Rightarrow \neg P(a,a)$                $\forall$-Elim (2)
4 ▷  | | $P(a,a)$                                     Assumption
5    | | $\neg P(a,a)$                                $\Rightarrow$-Elim (3,4)
6    | $\neg P(a,a)$                                   $\neg$-Intro (4,5)
7    | $\forall x.\neg P(x,x)$                         $\forall$-Intro (6)
```

Example 2.10. Show that $s = t \vdash (s = u) \Rightarrow (t = u)$

```
1    | $s = t$
     |----
2 ▷  | | $s = u$                          Assumption
3    | | $t = u$                          Substitution (1,2)
4    | $(s = u) \Rightarrow (t = u)$      $\Rightarrow$-Intro (2,3)
```

3 Design

The design methodology that I decided to use for this project is the Model View Controller idea. This entails the separation of all the front end website components from the back-end where the actual required functionality of the system is created. These two areas are then controlled by various controller classes, which bring all of these components together to create the finished application. I chose this methodology as it makes the entire system easier to understand and work on. If I want to add an extra feature to this application I can just update the model part of the code base and then change the front-end view code. This is much simpler than having all of the code together in one large code base. This methodology therefore made the development of this application much easier.

3.1 Model

In order to create the functionality required for me to meet my objectives in this project, I had to come up with a fitting design that allowed lots of different tasks to work together. This often meant creating a Java Object with a sole use of solving one of the problems I set out to achieve. I tried to separate the different parts of my code base into obvious classes which would make the code easy to read for any third party.

3.1.1 Component

In both Propositional and First Order Predicate logics, logical expressions can be made up of two different components, Propositions (or Predicates for Predicate logic) and Operators. For this project both of these components require some common functionality such as how they are converted to Strings, how they are interpreted when input and how equality between them is worked out. Due to this common functionality I decided to create an interface called Component which encompasses all the commonly used functions for each of these components. I then created a Proposition and an Operator class which would each represent a component type. Each of these classes would inherit from the Component interface. This set-up was very useful in other parts of the code base as it allows me to pass in a component and then check what type of component I am dealing with.

The Proposition class, as well as inheriting and implementing the common toString and equals methods from the Components interface, contains functionality to set and retrieve the name of a Proposition. A proposition is just made up of a String field which represents the name of the Proposition. This is one of the main components of expressions that are used in this logical system.

The Operator class is used to represent one of the seven operators used in this logical system, as described in the Background section above. This class also inherits to toString and equals methods from Components, making it much easier to use when building expressions. Each Operator object is made of a string which represents its name and an enum value corresponding to each operator. I have created an enum called OperatorType, which contains all of the different operator types used in this logical environment. The reason that I set operators up in this way is that it easily allows me to check what operator is being used in an expression at any time due to its uniquely identified enum value. The rest of this class is filled with getters and setters that help with the general functionality of this logical system. Here is a diagram of how this part of the code base is set up:

TODO: talk about quantifiers for predicate logic when done

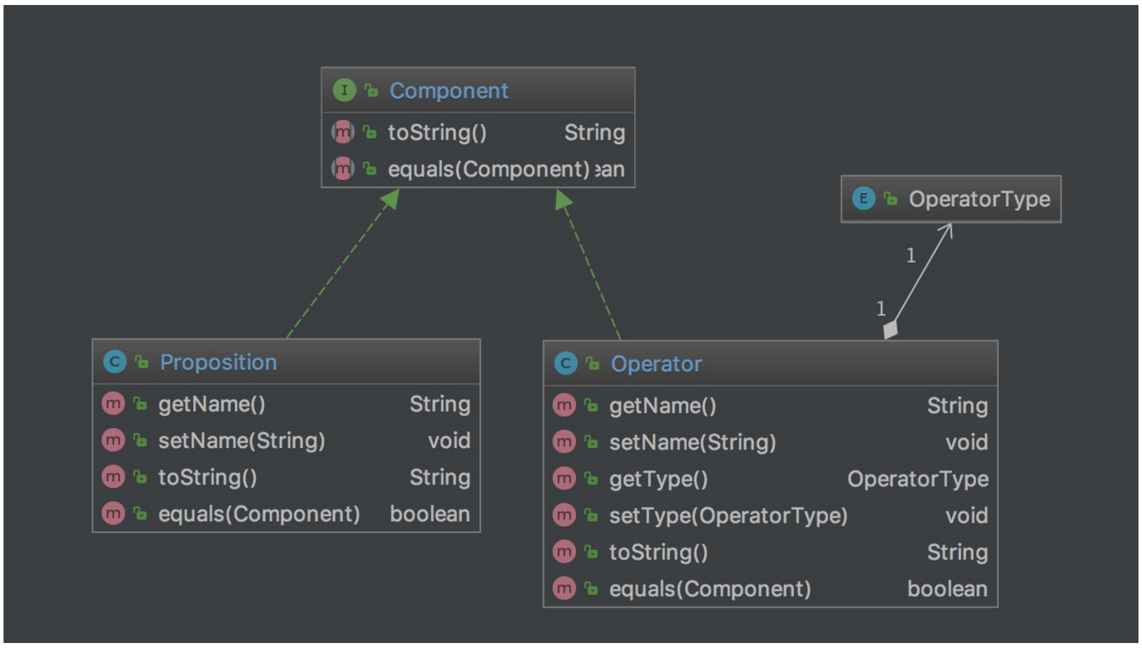


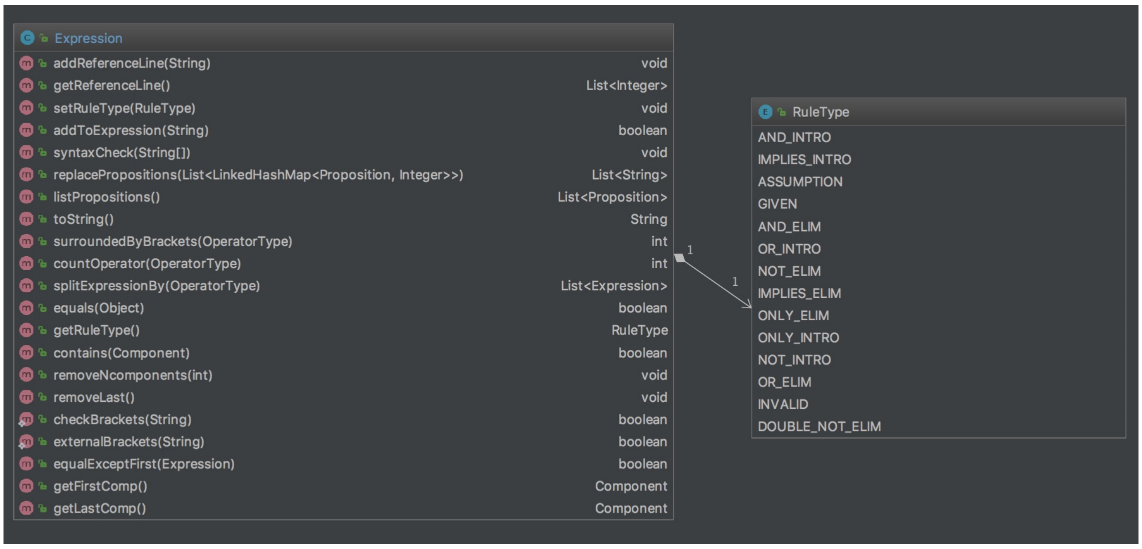
Figure 2: Component class structure

3.1.2 Expression

In Natural Deduction, a proof is made up of a set of expressions each derived from previous expressions based on the Natural Deduction rule set (see Appendix A). I therefore made an Expression class which was made up of a number of components. Firstly, each expression contains a list of Components, making up the expression itself in terms of propositions and operators. An expression also keeps track of its line number in a proof. This is very useful when applying rules that depend on several different lines in a proof. Another key component of an expression is what rule was used to derive it. I therefore created a ruleType enum which enumerates all of the different rule types used in a natural deduction proof. This enum includes rules for premises and assumptions. This makes it very easy for the user to find out what rule was used when an expression is formed.

The most important functionality that the Expression class carries out, is the parsing of expressions from their String inputs. When a user enters an expression into the front end of the programme, the expression is immediately tokenised by spaces and then each token is checked to see what type of component it is. While this tokenization is occurring, the expression is checked for any syntax errors that may have been input by the user. If one is found, the entire process is halted and the syntax error that has occurred is outputted to the user. The tokens are then added to the list of components to make up the expression list. This process allows us to create an internal representation of the expression, making it easier to use when applying the various types of functionality used later on in the programme.

In order to carry out the tokenization correctly, and to help with the various operations that are carried out on the expression throughout the code base, a lot of extra functionality has been added to the Expression class. For example an equals method that allows the comparison of two expressions, and a contains method which allows a user to check whether a certain component is used inside a given expression. This functionality makes it easy to manipulate expressions as they are used to solve complex proofs. One really commonly used function that is implemented in this class is the splitExpressionBy function. This function takes an expression and splits it by the specified operator (assuming it contains this operator). This functionality is very useful when applying the various Natural Deduction rules to an expression, especially for elimination rules which require the specified operator to be split into one or two sub-expressions. A summary of the functionality included in the Expression class can be seen in the following diagram:


Expression

addReferenceLine(String)	void
getReferenceLine()	List<Integer>
setRuleType(RuleType)	void
addToExpression(String)	boolean
syntaxCheck(String[])	void
replacePropositions(List<LinkedHashMap<Proposition, Integer>>)	List<String>
listPropositions()	List<Proposition>
toString()	String
surroundedByBrackets(OperatorType)	int
countOperator(OperatorType)	int
splitExpressionBy(OperatorType)	List<Expression>
equals(Object)	boolean
getRuleType()	RuleType
contains(Component)	boolean
removeNcomponents(int)	void
removeLast()	void
checkBrackets(String)	boolean
externalBrackets(String)	boolean
equalExceptFirst(Expression)	boolean
getFirstComp()	Component
getLastComp()	Component

RuleType

AND_INTRO
IMPLIES_INTRO
ASSUMPTION
GIVEN
AND_ELIM
OR_INTRO
NOT_ELIM
IMPLIES_ELIM
ONLY_ELIM
ONLY_INTRO
NOT_INTRO
OR_ELIM
INVALID
DOUBLE_NOT_ELIM

Figure 3: Expression Functionality

3.1.3 Proof

In order to represent the proofs that the user will be inputting into this software, a Proof class has been created. This class contains a large amount of functionality used to represent these proofs in a way that they can be manipulated to help solve Natural Deduction proofs. In this class a proof is represented by a list of expressions, each representing a line in a proof.

In order to populate a Proof object based on user input, this class reads in expressions and their associated rules and tokenizes them accordingly. It then fills the list of expressions in order to create the proof representation. While this is being carried out the code checks that each expression has been created using the specified rule correctly. For example if an expression was created using And Introduction, the code would check to ensure that both the right and left hand sides of the And operator have been proven earlier on in the code. If a rule has been used incorrectly, or cannot be used at this stage in the proof, then a rule error will be thrown aborting this entire process. Once all of the steps of the proof have been checked and been proven to have used the rules correctly the process will complete. This completion indicates that the proof so far is valid.

When a user inputs each line of their proof along with the rule used, they also must include the lines that these rules used to create this expression. I therefore must, when reading in this information, check that the lines mentioned are valid and correspond to expressions that can be used for this rule. As a proof is represented by a list of expressions it is easy to traverse the list and find the required expression by line number. Another additional complexity that has to be taken into account here, is the scope that the current expression is in. Whenever an assumption is made a new scope (or box), is opened until it is closed by certain rules being completed. Once this has happened nothing after the scope can use any expressions inside the scope for any further steps of the proof (apart from the result of the scope). This is therefore another check that the code must make when checking whether a rule has been used correctly at certain points of the proof.

3.1.4 Error Checking

One of my main goals when conducting this project, was to make this Natural Deduction solver into a tool to help teach people how to successfully solve these types of problems. I therefore made sure to have very detailed error messages that allow a user to see exactly where they have gone wrong, so that they can learn from any mistakes they may have made. In order to make this as easy as possible every error message that is displayed contains the type of error that has occurred, the line that it was thrown on as well as some extra information about the associated problem.

This makes it very easy for the user to immediately understand what they have done wrong. In the case where no errors are thrown the program will display the string "Proof is Valid", alerting the user to their valid proof.

The first type of error check that occurs as soon as the parser starts to read the user input is a Syntax Error. This check ensures that each component is placed in a valid part of the expression, so that each expression is valid. For example if two operators are added to an expression in a row, this error will be thrown. Similarly if there are any mismatched brackets in an expression, a syntax error will immediately be thrown. This type of error is fundamental as the syntax of the proof must be completely correct in order for the user to move onto actually solving the proof at hand. This check is done right at the beginning of the parsing process for this very reason.

The other type of error check which helps to make this an educational tool, is a Rule Error. Whenever a proof is entered by a user there is a function in the Proof class which goes through the proof and checks that each line is valid. It does this by going through the list of expressions (the proof), and checks whether the specified rule can be used in the proof at this point. It also checks that the lines referenced by this rule are valid and are not out of scope. If all of the rules used are used correctly then the user is notified that the proof is completely valid. If not the user will be alerted to the line and rule which was incorrectly used, helping them to learn from their mistake. In order to clearly show the user all of the rules that have been broken at the same time, the Proof class contains an error list which slowly builds up as the proof is checked. The list is then displayed after this process is complete. I decided to allow this accumulation of errors so that a user can fix several errors that exist at a time, rather than having to solve several over a long chain of trial and error. These error checks are done after any syntax errors that may cause expressions to be invalid have been remedied.

3.1.5 Validity Checking

Natural Deduction proofs can be quite long, adding an extra difficulty when trying to debug a mistake you have made in the proof. When solving a proof you are given it would be helpful to know that you can solve the proof with the given premises and result at all. It would be pointless to just start solving the proof if you will never reach the resulting expression. This is a major problem faced by even the more experienced solver of these problems.

In order to remedy this potential issue I have added an extra validation stage before the main solver functionality. This stage allows the user to input their desired premises and result, and they will be used to check whether a proof using these expressions is valid. In order to implement this I have used the Truth Table validation method. This works by calculating the truth values of each premise and the result. The functionality then looks to see whether in the case when all of the premises evaluate to true, the result expression also evaluates to true. If this is true for every case of the premises all evaluating to true, then we know that the proof is valid and that there is a possible proof that can be used to solve it. I implemented this check by first creating a TruthTable object, which is represented by a 2D array. When the user submits their premises and result, this array get populated with the possible input values for each proposition, the truth values of each premise in each of the cases, and the truth value of the result in each case. This object then has functionality that checks that all of the truth values are true in the required cases, and if so designates this proof as valid. This whole mechanism will save a user a lot of time by allowing them to see immediately whether a proof they are about to try to solve is solvable.

3.1.6 Hints

One of the advanced features that I set out to implement during this project is hints. I wanted to allow a user to start a proof and if they became stuck at any step, they could use my software to gain a hint as to what rule to apply next. This feature is something that is lacking in similar software such as Pandora. When first planning to include this feature in this project, I felt that I would first implement a simple form of hints, where when a hint was requested I would try each of the rules and reply to the user with any rules that didn't throw any errors. Obviously, this is quite a short-sighted way of implementing this as a user could be sent off in a completely wrong direction. Also when it comes to introduction rules there can be quite a lot of different possible cases to consider. I therefore decided that this methodology for generating hints is not useful for a user.

Based on this discovery, I decided that the best way to implement this feature would be to actually apply an algorithm which solves the proof first. This would mean that when a hint is requested the system compares the users proof so far to the solved proof, in order to generate a hint for the next step. The first algorithm that I tried was one which took the premises that were given by the user and firstly tries each of the elimination rules on them several times. As this was happening the algorithm built up a list of possible proof makeups up to this point. The algorithm then tried some introduction rules in order to move further into the proof. This process would continue in a loop until either the resulting expression was reached, or a time-out was met. I also added a few enhancements to the algorithm that meant right at the beginning the algorithm looked at the resulting expression to see whether it, for example, contained an implications operator, meaning that an assumption needed to be added to the proof in order to form part of the implies introduction rule. Although this algorithm was able to solve the proofs in order for hints to be generated, in proofs which include any introduction rules, the algorithm took quite a long time. In some cases that I tested, a proof took over thirty seconds to be solved. This is due to the large number of options that needed to be tried for each introduction rule, adding even more possible combinations to the list of possible proofs. Obviously, this algorithm would not be able to scale very well. I therefore decided to search for some more efficient algorithms to solve this problem.

After much research I found a very effective algorithm in a paper called Automated First Order Natural Deduction [3]. This algorithm works in a completely different way to the one I devised above. The algorithm creates two lists, the first of which is called list_proof containing the steps of the proof that have been solved at any point in the algorithms execution. The algorithm also includes another list called list_goal, which contains the goal of the whole proof, as well as any sub-goals that are created through any introduction or the or elimination rules. The algorithm works by first adding the premise, if there are any, to the list_proof list and the resulting expression to list_goal. It then tries out each of the elimination rules to see if they can be applied at this point of the proof. If one is found to be valid, it will be applied and the resulting expression is added to list_proof. Then when each elimination rule has been checked, the algorithm moves onto the introduction rules. If it finds one of these introduction rules as being a valid step in the proof it will add the newly introduced equation to list_goal, in order for the algorithm to work out a way to reach this new sub-goal. After this the algorithm will start checking elimination rules again until this sub-goal is met. Once all of the sub-goals and the main goal itself are met, the algorithm as this last goal to list_proof. list_proof now contains the completed proof that can be used for hints. The main benefit of this algorithm, is that by setting up sub-goals, the algorithm does not have to spend large amounts of time looking for possible combinations of expressions for introduction rules. Rather it knows what the goal of the introduction will be so will apply the appropriate rules in order to meet it. This algorithm therefore works quickly in all cases and can scale very well. This makes this algorithm very efficient and is why I chose to use it for this part of my project.

In addition to the basic algorithm, I added a few features to optimise the speed that the algorithm runs at. In particular at the start of the algorithm I added a check which looks at the resulting expression of the proof and sees from the start whether it contains an implication operator or starts with a not operator. If so then the algorithm now knows that the last rule applied will either be implies or not introduction. The algorithm can therefore skip the elimination rule checks at the beginning and go straight to these introductions. This speeds up the whole process in these cases significantly. The paper that I retrieved this algorithm from defines its Or elimination rule slightly different to how I do in this project. For this project in order to carry out an Or Elimination, I split the two sides of the Or operator into two separate boxes and try to find the same resulting expression, thereby eliminating the original expression. I therefore had to add some extra functionality for this case into the algorithm. How it works is, when this case is entered I add the assumption of the left hand side expression to list_proof, while I add the assumption of the right hand side expression to another temporary list. I then continue with the algorithm as normal, but use both lists at each step. Then when both list_proof and the temporary list have reached the same conclusion, I add the contents of the temporary list to list_proof and add the concluding step for the Or elimination rule. By implementing this rule in this way, I am able to use my definition for this rule whilst still benefiting from the use of this efficient algorithm. Using this algorithm I am able to provide the user with hints at any stage of a proof to help them learn how to solve Natural Deduction proofs.

3.2 View

3.3 Controller

4 Development Tools

4.1 Java

When embarking on a project such as this, the language that is chosen to implement all of the complex functionality is very important. In particular the language that is chosen must be able to be easily compatible with front-end infrastructures. Lots of languages also have extra features such as large testing frameworks and advanced IDE's that would make the whole process of making an application much easier. Also once a language has been chosen it would be a lot of work to then transfer to another language if I changed my mind. I therefore spent quite a long period of time deciding which language was best for this project.

The first language that I considered using was Prolog. Prolog is already a language based on logical systems, and had I used it, representing logical statements would have been much simpler. I also found several user interface libraries that use Prolog itself to create a front-end website for applications. Based on these finds I could have built this application using Prolog. The main drawback for me if I had used this language, is that I am not as familiar at using it as other languages, meaning that I would have to waste a lot of time learning the language first before actually starting the application. Also, some of the more advanced features that I wanted to add, such as hints, would have been quite hard to implement using this language. I therefore in the end decided to look at other languages to use for this project.

The language I ended up settling on was Java. I chose to use Java as the main language for this project for a number of reasons. Firstly, Java is the language that I am most familiar with and I am able to use most of its advanced features and large array of libraries. This made it very easy for to create the representation of Proofs, as well as some of the more advanced functionality for this project. Another reason that I chose to use Java is the number of libraries and data structures it offers. I was able to use advanced lists and hashMaps throughout the code in order to ensure that all of the required functionality worked as efficiently as possible. Other languages like Prolog do not have these resources. On top of all of this, there are lots of tips and references available for the Java language for me to use whenever I needed them. The only drawback that I found of using Java, is that it meant writing quite a large amount of code to create simple functionality, which can be quite time consuming. However due to my knowledge of the language this did not hinder my progress too much.

4.2 Java Spring

4.3 Javascript

4.4 JUnit

When carrying out large scale software projects such as this, one of the most important tasks that must be carried out is testing. There is no easy way to be able to tell whether your code is completing the task you created for without tests that put your code in realistic scenarios and see how they function. This is especially true for edge cases which in reality may never happen, but there is always the possibility. As I used Java for this project it was natural to use the very powerful JUnit test suite. JUnit allowed me to carry out individual tests on almost all of the functionality that I have created in the back end of the application. Through the use of Test Driven Development (see 5.2.1), I was able to use the test suite itself to help me work out how to implement some of the more complicated code. This also allowed me to keep track and ensure that any new code I added wasn't breaking any other parts of the code base. Using the JUnit assertTrue and assertFalse functionality, in each test case I was able to compare the computed result with the expected one, in order to ensure everything was working correctly. JUnit made this a smooth and easy process saving me a lot of time that I could dedicate to adding more features to the application.

4.5 Maven

This project involved me using a combination of different technologies in order to create the finished product. In order for these tools to work together effectively I decided to use Maven. Maven allowed me to bundle together the Java Spring capabilities with the Java code and JUnit tests to create an efficiently functioning system. I was able to add dependencies for each of the tools to the Maven project, and through using the standard Maven directory set up, I was able to also add my front end code to this package. By using this containerisation I was able to easily deploy my application at any part of the development process to check that it was working correctly. Maven also allowed me to ensure that all of the JUnit and front end tests that I had set up were run before the programme was deployed, allowing me to make sure all parts of the code base were functioning correctly. Maven also helped me to set-up other tools such as Github and Heroku for this project, with each of these tools being compatible with the Maven infrastructure.

4.6 Intellij

When writing Java code it is very easy to make small syntactic mistakes in the code without realising. Whether its a missed semicolon or even the wrong number of curly brackets. This is especially poignant in larger code bases. The easiest way to avoid having to spend large amounts of time looking for these bugs and continually rerunning the code as you slowly fix them, is by using an IDE. There are many IDE's to choose from for the Java language, but I decided to use Intellij for a number of reasons. Firstly, Intellij has a very large range of features that would make coding this project much easier. For example on top of just alerting me to syntax errors, there is a code duplication error system, as well as built-in re-factoring functionality that allows me to make my code much more readable and efficient. On top of this, Intellij is compatible with Maven meaning that I can even build my Maven project inside Intellij, making the entire process much smoother. This compatibility goes so far that Intellij also downloads any dependencies that I have added to the Maven pom file. All of these reasons and more make Intellij the ideal IDE to use for this project.

4.7 Git

A key area that needs to be considered when carrying out a large scale project such as this is version control. One cannot afford to lose their projects source code, and it is really useful to have previous versions of the code-base that can be returned to at any point in the project. For this project, I decided to use Git as I am very familiar with its functionality and so would be able to effectively utilise all of its useful features. In order to manage my git repository, I decided to use Github. Github has a really easy to use interface and is compatible with Maven making it very easy for me to upload my project. I was able to create a private project on Github and using its easy to use website I was able to look up previous versions of my code base whenever necessary. Github also provides some very useful statistics about how the version control side of my project is progressing. I used these statistics to try to keep up a good level of Software Engineering practise.

One git feature that was really useful for me throughout the project was the use of git hooks. I have create a git hook which whenever I commit my work, runs all of my unit tests over the code and ensures that they all pass. If any fail the commit is halted and I must fix the associated errors before being able to commit. This is a very useful feature as often after changing one part of the code-base, you may accidentally break the code somewhere else. Without git hooks these errors may not be picked up on, meaning that broken code is committed to git. This is obviously not a good technique to follow, so I made sure to add these hooks as early as possible in the project to avoid this from happening.

4.8 Heroku

As my project is a web based application I wanted to find and easy way to deploy it to a real website. Heroku allowed me to easily do this, as well as offering me many useful features. Heroku allowed me to easily connect my Maven project to its services through my Github account. It then allowed me to set it up so that whenever I pushed my changes to master, my Heroku site would be updated. I therefore used my site on Heroku to house my latest fully functioning version of the app. This allowed me to show others and gain feedback on the parts of the functionality that I had already created. Deploying my app in this way also allowed me to test it on other browsers and operating systems, to ensure that all of the required functionality functioned correctly. Once the project was completed, I permanently deployed the application to this free service.

5 Project Plan and Management

5.1 Schedule

In order to ensure that I am able to meet all of my objectives for this project, I have created a project plan. This project plan consists of seven iterations, each of which builds on the previous adding new features and fixing any discovered bugs. On top of these iterations I will also be regularly meeting with my supervisor to ensure that I am on track with my progress and to make sure that I am heading in the right direction. The iterations that I have created are quite long periods of time to ensure that the jobs set out in each iteration are completed to the best of my ability. The iteration lengths also take into account other university work that I may be working on concurrently with this project. I have created the iterations to also be quite flexible so that I can for example continue work not completed in a previous iteration in the next iteration. This way I will be able to ensure that all my tasks are completed.

5.1.1 Iteration 1 - 1/12/2016 $\Rightarrow$ 8/1/2017

In this iteration I will complete the various tasks needed to set up and plan the project as a whole. To start things off I will research Natural Deduction as much as possible to ensure that I fully understand the rules and other technical terminology involved. This will allow me to envisage exactly what needs to be done to complete my objectives. Based on this research I will next decide the various features that I want to add to the project and based on this decide which tools to use. I will decide firstly which language is best suited for this project and which other tools I feel necessary to use that will help me fulfil my objectives. Once I have decided which tools to use I will set up my version control and create a blank directory for this project. Now that I am ready to begin coding up the project I will plan what I am going to do in future iterations to ensure that I meet all of my objectives. I can also take a step back at this point and make sure that all of my objectives are feasible in the time frame I have. Throughout this iteration I will meet with my supervisor several times to ensure that all of my plans are feasible and on the right track.

5.1.2 Iteration 2 - 9/1/2017 ⇒ 10/2/2017

In this iteration I will start to actually code the first parts of the software for this project. I will start by creating a representation of Propositional logic that will be understood by the system, and a way to tokenise and convert string input into this representation. I will then create a parser that checks whether a rule that was parsed in is valid based on the Propositional Natural Deduction rules. On top of this I will create a back end representation of a proof that can be checked using the parser. Once I have done this I will have a back-end that allows a user to enter a proof and each step in the proof will be checked to ensure that it abides by one of the Propositional Natural Deduction rules. Throughout the creation of the back-end part of this software in this iteration, I will ensure to rigorously test all features that I add. This will ensure that all code I write at this stage is working, stopping bugs from occurring in future iterations.

I will not just be working on the back-end in this iteration but also on the front-end. In order to be able to help test that my proof validity checker works completely I will create a simple web interface that allows a user to enter a proof and click a "check proof" button. This button will then apply the back end functionality to check whether the proof is valid. By the end of this iteration I will therefore have completed my first objective of creating a tool which allows a user to check whether their Natural Deduction proof is valid. As this is one of the core components of the project I have made this iteration quite long to ensure that it is completed.

5.1.3 Iteration 3 - 11/2/2017 ⇒ 10/3/2017

In this iteration I will focus firstly on completing the second of my main two objectives of creating a Natural Deduction proof checker with hints. I at this stage already have the functionality to check whether the proof that has been entered is valid, so can use this to work out what possible rules could be applied next by the user. The functionality that I create in this iteration will hopefully allow the user to stop at any point in their proof and allow them to click the hint button. The software will then produce a suggestion of which rule to apply next. It may be that multiple rules can be applied at a given point and it will be up to the user to decide. This is quite a fundamental part of my project so I have given myself quite a lot of time to complete this in order to ensure that I can test the code making sure it is working correctly.

As well as adding this extra functionality in this iteration I also want to create a nice looking and easy to use user interface. Before this iteration I had only created a basic user interface which allows the user to type in their proof and then press various buttons to check whether their proof is valid. This however is not very intuitive to use and is not very nice to look at. In this iteration I therefore intend to add an HTML template to the interface to make it look much nicer and professional, and insert my functionality into this template. This as well as creating an aesthetically pleased interface ensures that my software is easy to use. By the end of this iteration I will have completed my main two objectives and thereby created a fully functioning and easy to use Natural Deduction proof checker for Propositional logic.

5.1.4 Iteration 4 - 27/3/2017 ⇒ 21/4/2017

Up to this point in the project I have only focussed on Propositional logic and how Natural Deduction proofs using this logic system can be checked. However this logic system is not very expressive and so is not usually used to model everyday situations. A much more useful logical system is first order Predicate Logic. In this iteration I want to firstly start creating a representation in my code for Predicate logic. This will be slightly harder than for Propositional logic as I need to take into account quantifiers which add a complete other dimension onto the logical model. Once I have create this basic Predicate representation, I will start to add functionality that checks whether steps in Natural Deduction proof using this logical system are valid. This is very similar to how I did rule checking in the Propositional case as I will just be going through and checking based on the set Predicate logic rules for Natural Deduction. However these rules are slightly more complicated to check for in a proof so I have allowed myself extra time to complete this task.

As well as this major addition to the back end of my project, as these new features are added I will also need to update the front end accordingly. At this stage of the project I will just add some basic functionality that will allow me to user test this new back end functionality. This would probably just comprise of a dialogue box and a few buttons. This will give me enough functionality to test the new Propositional proofs to make sure they are valid. Overall this iteration is slightly shorter than previous ones, but I now will be able to work on this project full time having finished my other commitments allowing me to focus fully on this project.

5.1.5 Iteration 5 - 22/4/2017 ⇒ 12/5/2017

In the last iteration we created a representation of Predicate logic together with a checker for Predicate Natural Deduction proofs. Now as I did with Propositional logic, I will add hint functionality that the user can use while composing their proofs. As we have already created the rule verification in the last iteration this iteration includes the application of these rules to any step in the proof to give the user an idea of what to do next. This as with the Propositional logic part of this project is a very useful part of this tool.

Now that we have completed the Predicate Logic part of the back end of the project I will make sure that the user interface is user friendly and looking nice when the user tries to use the Predicate part of the tool. This may mean adding extra web pages and features to allow the user to choose between the type of logic they want to use in their proofs.

5.1.6 Iteration 6 - 13/5/2017 ⇒ 2/6/2017

At this point I have completed all of the main features that I wanted to add to my Natural Deduction checking tool. There are however some other features which would be really useful for the user. This iteration allows some time to add these features. One feature in particular that I would like to add is error correction. This would allow the user to type in a line of their proof and the software would be able to tell whether it is valid or not, giving the user an alternative valid rule if not. This would really help the user to find out exactly where in their proofs they have gone wrong, making this an even more useful tool for people learning how to create Natural Deduction proofs.

In this iteration I would also like to spend time adding some other extensions such as the saving and loading of proofs. This would allow the user to load in proofs they have written elsewhere and also save any proofs they have now checked to use elsewhere. These extensions are all dependent on the amount of time I have left at the end of the project.

5.1.7 Iteration 7 - 3/6/2017 ⇒ 30/6/2017

From the start of this iteration I want to have stopped adding any new features. I will go through my code, tidy it up, add comments and ensure that all unnecessary code is removed. I will also look through and try to solve any remaining bugs that I have left in my code. At the same time I will complete my report noting how I carried out my project. I want to allow as much time as possible to do this, to make sure that I can create a high level and well written report. I will also use this iteration to create my presentation of the project making sure it looks professional and is able to inform the audience of exactly what my project can do.

5.2 Programming Methodologies

5.2.1 Test Driven Development

5.3 Testing

5.4 Trello

6 Evaluation

ideas: tableaux instead of truth tables, more realistic plans, Java good language to use, differences to pandora

6.1 Expected Outcomes

By the end of the project there are a number of objectives that I want to fulfil in order to classify it as a successful project. I firstly want to make sure that the tool I have created is easy to use. It would be better for my software to have less features if it means that it is very easy to use by any user. This is one of the areas that I hope my software is better than other similar tools. I also by the end of the project want to have a working Natural Deduction proof checker, with the ability to give hints to the user at any point in the proof. This checker only needs to be for Propositional logic in order for me to fulfil my objectives. This would still mean that I have created an important and useful tool that can be used by anyone learning how to form Natural Deduction proofs. If however I have more time I would also like to extend this to Predicate logic. If I am able to complete these two tasks I will have succeeded in creating a useful tool for Natural Deduction students.

6.2 Testing

Throughout the project I intend to use Test Driven Development (TDD) to ensure that my code is always functioning as expected. This allows me to add new code to the project without worrying about how this may affect other parts of the code-base. This is especially easy for my back-end which is written in Java allowing me to easily add unit tests as I go along. The fact that I am using Java Spring to build a Model View Controller set up for my system, also provides me with some functionality allowing me to test that connections between the front and back ends are always made correctly. The front end of the project will be the hardest component for me to test as it will require user input into the various dialogue boxes provided by my UI. I will therefore work on some basic testing of this part of the project using some of Java Spring's advanced testing features. I will condense all of these tests together using the TeamCity continuous integration software. This will allow me keep track and ensure that all parts of my software are functioning correctly. Overall I intend to create quite a high coverage level of my code by the end of the project.

At the end of the project if the test coverage of my software is high, this will indicate to me that I have completed a successful project which is proven to work. This will also mean that I have carried out the project in a professional way, making sure to always test the code that I am writing. In the end the amount of testing I do will show me how successfully I was able to complete the task brief.

6.3 User Feedback

User feedback is an important tool that should be collected when completing any type of project. This is particularly true with software which is strongly user based and requires user input. I must make sure that the software I have created is intuitive to use and actually solves a problem that users are having. This is why one of my main aims at the start of this project was to ensure that the system is easy to use by anyone no matter the level of subject knowledge they have. In order to ensure that this is the case as soon as I have a working prototype of my software I intend to start asking family and friends to try it out. The more feedback that I get back the better, and the earlier I start to receive this feedback the more time I will have to actually act on it. I also want to make sure to attend the various Department of Computing demonstration days to gain as much feedback as possible about my finished product. This feedback will as well as allow me to improve the software as my iterations go on, will enable me to evaluate how well I have been able to meet my objectives by the end of the project. I will make sure that the feedback I gather is representative of people who are experts in this field as well as people who are new to this subject area. Overall this will indicate to me how successful my project was.

6.4 Project Plan

I have put a lot of thought into the project plan that I created for this project. I made sure that I have enough time to complete all of the tasks that I set out to achieve at the start of this project, as well as time at the end for any extensions I decide to add. The iterations I have created are quite flexible in order to ensure that I am able to complete the more important parts of the project which help me to meet my initial objectives. One indication of my project being successful may be how well I am able to keep to this iteration schedule. The schedule works on the premise that I complete each iteration and then work on improving and adding features to what I have in the next iteration. If this works I will be able to ensure that all the tasks I set out at the beginning are completed. If I end up not being able to complete all of these tasks, I will be able to evaluate how in the future I will be able to structure my work plan for similar projects. It may even allow me to see how I could have planned my timetable better to ensure more work was completed in the time frame. At the end of each iteration I will be able to evaluate how well that iteration has gone, and if necessary I will be able to change future iterations accordingly. This is therefore one indicator of success during the implementation of my project.

7 Conclusion

A Natural Deduction Rules

A.1 Propositional Rules (adapted from [2])

Rule 1 (And Introduction). $\dfrac{A \qquad B}{A \wedge B}$ *or* $\dfrac{A \qquad B}{B \wedge A}$

Rule 2 (And Elimination). $\dfrac{A \wedge B}{A}$ *or* $\dfrac{A \wedge B}{B}$

Rule 3 (Or Introduction). $\dfrac{A}{A \vee B}$ *or* $\dfrac{A}{B \vee A}$

Rule 4 (Or Elimination). $\dfrac{A \vdash C \qquad B \vdash C \qquad A \vee B}{C}$ *where A and B are assumptions*

Rule 5 (Not Introduction). $\dfrac{A \qquad \perp}{\neg A}$ *where A is an assumption*

Rule 6 (Not Elimination). $\dfrac{\neg A \qquad A}{\perp}$

Rule 7 (Double Not Elimination). $\dfrac{\neg\neg A}{A}$

Rule 8 (Implies Introduction). $\dfrac{A \vdash B}{A \Rightarrow B}$ *where A is an assumption*

Rule 9 (Implies Elimination). $\dfrac{A \qquad A \Rightarrow B}{B}$

Rule 10 (Iff Introduction). $\dfrac{A \Rightarrow B \qquad B \Rightarrow A}{A \Leftrightarrow B}$

Rule 11 (Iff Elimination). $\dfrac{A \Leftrightarrow B}{A \Rightarrow B}$ *or* $\dfrac{A \Leftrightarrow B}{B \Rightarrow A}$

A.2 Predicate Rules

Rule 12 ($\forall$ Introduction). $\dfrac{P(a)}{\forall x.P(x)}$ *where a is arbitrary*

Rule 13 ($\forall$ Elimination). $\dfrac{\forall x.P(x)}{P(a)}$ *where a is arbitrary*

Rule 14 ($\exists$ Introduction). $\dfrac{\exists x.P(x) \qquad \forall x.(P(x) \Rightarrow Q)}{Q}$

Rule 15 ($\exists$ Elimination). $\dfrac{P(t)}{\exists x.P(x)}$ *where t is any term*

Rule 16 (Substitution). $\dfrac{m = n \qquad S(n)}{S[m/n]}$ *or* $\dfrac{m = n \qquad S(m)}{S[n/m]}$

References

[1] Pandora (Proof Assistant for Natural Deduction using Organised Rectangular Areas) is a learning support tool designed to guide the construction of natural deduction proofs. Found at https://www.doc.ic.ac.uk/pandora/, last retrieved 7/2/2017

[2] Software Engineering Mathematics by Jim Woodcock and Martin Loomes, 1989 edition

[3] Alexander Bolotov, Vyacheslav Bocharov, Alexander Gorchakov, Vasilyi Shangin *Automated First Order Natural Deduction*. Conference Paper, January 2005