NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DIOGO DA SILVA CAGICA CARVALHO

ONLINE INTERACTIVE TOOL FOR LEARNING LOGICAL PROOF SYSTEMS

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
12, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

ONLINE INTERACTIVE TOOL FOR LEARNING LOGICAL PROOF SYSTEMS

DIOGO DA SILVA CAGICA CARVALHO

Adviser: Ricardo Gonçalves
Assistant Professor, FCT-NOVA

Co-adviser: João Costa Seco
Associate Professor, FCT-NOVA

Examination Committee

Chair: Rui Pedro da Silva Nóbrega
Assistant Professor, FCT-NOVA

Rapporteur: João Pedro Guerreiro Neto
Assistant Professor, ULisboa

Adviser: Ricardo João Rodrigues Gonçalves
Assistant Professor, FCT-NOVA

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
12, 2023

Abstract

For many areas, including mathematics and computer science, logic is a basic but important topic. In particular, the notions of natural deduction and proof systems for natural deduction are of fundamental interest.

Students learning logic benefit from the use of interactive, visual tools where they can solve exercises and receive feedback from their attempts. However, most of these tools are based on installable programs, or static, making it hard to expand them with additional learning material and exercises. The few available online tools and courses focusing on logic also tend to skip the topic of natural deduction.

The goal of this thesis is the development of a tool that hosts an interactive, visual proof assistant in which a student can create (or load) a proof and attempt to solve it, occasionally receiving feedback to help the student learn and progress at their own pace. The tool's design would leverage the work done on previously created tools of this kind and function as a complementary tool for computational logic courses or any course that includes natural deduction.

To achieve this, the system must implement basic logic algorithms, feature a logic reasoner for the supported branches of logic, be capable of parsing text into formulas, have mechanisms to provide feedback to users regarding errors and finally some kind of exercise management system, allowing a qualified user to save proofs that can later be loaded by any user, while maintaining the state it was saved as.

Finally, it's intended to leverage its usability for automatic evaluation purposes, by incorporating the system, if possible, with existing online e-learning platforms, such as Moodle.

Keywords: Logic, Proof System, Natural Deduction, Online, Interactive, Propositional Logic, First-order Logic

Resumo

Para muitas áreas, incluindo matemática e ciência da computação, a lógica é um tópico básico mas importante. Em particular, as noções de dedução natural e sistemas de prova para dedução natural são de interesse fundamental.

Estudantes que estejam a aprender lógica beneficiam da utilização de ferramentas interativas e visuais onde podem resolver exercícios e receber feedback durante as suas tentativas. No entanto, a maioria destas ferramentas baseia-se em programas instaláveis, ou estáticos, o que torna difícil expandi-los com exercicios e material de aprendizagem adicional. As poucas ferramentas e cursos online disponíveis que se focam em lógica também tendem a omitir o tópico da dedução natural.

O objetivo desta tese é o desenvolvimento de uma ferramenta que aloje um assistente de prova interativo e visual, no qual um estudante pode criar (ou carregar) uma prova e tentar resolvê-la, ocasionalmente recebendo feedback que o ajude a aprender e progredir ao seu próprio ritmo.

O design da ferramenta aproveitaria o trabalho realizado em ferramentas deste tipo criadas anteriormente e funcionaria como uma ferramenta complementar para cursos de lógica computacional ou qualquer curso que inclua dedução natural.

Para alcançar isto, o sistema deve implementar algoritmos lógicos básicos, conter um raciocinador lógico para os ramos de lógica suportados, ser capaz de transformar texto em fórmulas, ter mecanismos para fornecer feedback a utilizadores referente a erros e, finalmente, algum tipo de sistema de gestão de exercícios, permitindo a um utilizador qualificado guardar provas que podem posteriormente ser carregadas por qualquer utilizador, mantendo o estado em que foi guardada.

Por fim, pretende-se aproveitar a sua usabilidade para fins de avaliação automática, incorporando o sistema, se possível, em plataformas de e-learning online existentes, como o Moodle.

Palavras-chave: Logica, Sistema Dedutivo, Dedução Natural, Online, Interativo, Logica Proposicional, Logica de Primeira-Ordem

Contents

List of Figures

List of Tables

Acronyms

DTO Data Transfer Object *(p. 49)*

JNLP Java Network Launch Protocol *(p. 26)*

LEM Law of Excluded Middle *(pp. 5, 8)*

MOOC Massive Open Online Course *(pp. 1, 29, 30)*

1 Introduction

1.1 Context

Logic can be defined as the study of rules and the philosophy of valid reasoning and plays a fundamental role in mathematics and computer science. Logical reasoning plays an important part in, for example, the design and analysis of algorithms, the development of verification tools, and others. Due to its importance, computational logic is an important topic to learn when studying computer science, with most universities offering computer science degrees including a course that teaches it. Traditionally, logic has been taught in universities as part of computer science courses, with teachers explaining the theory and concepts to students and then providing exercises to consolidate them. In logic, a way to infer the validity of reasoning is by utilizing a deductive system, and a type of deductive system taught in logic courses is natural deduction, a method of formal logical reasoning that is based on inferences and is very often a concept that students struggle with. To better understand this concept, students are normally required to obtain a considerable amount of practice through solving exercises as well as requesting help from teachers whenever struggling to overcome an exercise. However, teachers are not always available to help and there are many students enrolled in these courses. To accommodate this issue, a look should be given at online tools focused on teaching, and how they can act as a complementary tool to help with traditional in-person teaching by providing a platform where students can attempt exercises, receive feedback whenever an error is made or when the student is unable to solve the exercise and, once finished, check if their attempt is correct. A tool such as this one, but geared towards helping in learning natural deduction, can prove very useful to both students and teachers and help in alleviating issues such as the, at times, unavailability of teachers.

In recent times, there has been an increase in both supply and demand for online teaching tools, even before the COVID-19 pandemic. The popularity of Massive Open Online Courses or MOOCs, for short, has been ramping up, with MOOCs such as EdX, Udacity and Coursera offering a large variety of courses as early as 2015 [10]. The increasing popularity of MOOCs shows that there are many people interested in learning but either

cannot or do not want to attend in-person classes. Through online teaching, these people can follow courses in areas of their choice from anywhere in the world. These tools can offer learning material, exercises, homework, tests, and help through contact with teachers and feedback given with exercises. Some also have discussion forums where students can trade ideas with each other. However, only students with more expertise in subjects tend to use them, with the others remaining passive. [17]. The idea of student autonomy is very prevalent in MOOCs, from the way students can pace themselves to how much effort they apply to the course and more. While autonomy is often desirable, the number of people who enroll in these far outnumber those who complete them, even though the number of people who do complete them is still significant [5].

With the surge in demand for online teaching tools, it might come as a surprise that logic-focused MOOCs are rare. Back in 2016, the number of MOOCs tackling logic was very small, possibly even none [11]. Nowadays there is a MOOC focused on teaching logic, Iltis [7, 8], but even it doesn't tackle the topic of natural deduction. Tools with a focus on natural deduction do exist, such as proof assistants, which are tools available on computers where students can write down their proof and have the assistant check its validity. These tools are helpful in the process of learning natural deduction as they can assist the user with understanding what is wrong with their proofs, especially if the tool provides feedback whenever necessary. While the tool cannot substitute a teacher completely, it can be of good use to both students and teachers alike as a complementary tool to a logic course, allowing students to get some form of assistance when facing problems when the teachers are not available to help.

1.2 Problem

While there are many visual proof assistants available online or as an installable, it is difficult to find one that meets all of the requirements needed to act as a complement to an in-person course. Some of the most important requirements for this kind of tool would be the type of proof format, its capability to provide feedback, and its capability to save and load exercises.

Many proof assistants represent their proofs in the Fitch-style proof format as it seems to be the most popular format. However, the Gentzen Tree-style proof format is advocated for teaching purposes by some authors [14] as it provides a "better explanation of the meaning (...) of logical connectors and quantifiers.", better reflects usual reasoning and allows errors to be pointed out with more accuracy. Unfortunately, the Fitch-style format and the Gentzen tree-style format are very different in regards to visuals and have some characteristics that are unique to them and not present in the other, such as marks in Gentzen trees and subproofs in Fitch. These formats are explained with more detail in section 2.3.1. The way these tools program their logical reasoning is also of some importance for this topic. Assistants rely on one of two different options, those being writing and programming the reasoning from scratch and utilizing high-end proof

assistants such as Coq and Isabelle/HOL to do the verification for them. Both of these options need to be considered for the development of a tool, as they also influence the choice of programming language for the development of the tool. For the second option, for example, the programming language would need to be capable of communicating with an API that can in turn communicate with an high-end proof assistant.

Exercise management is mostly done by saving the proof in a viable format that allows it to later be reloaded in the same state that it was left in previously. As for the format, it varies from tool to tool. Some tools, such as Logan [1], import the proof into the LaTeXformat, but they cannot reload the proof this way. While the Iltis MOOC doesn't offer natural deduction, it generates its exercises in a specific, custom format, which is then stored in an XML file. The same is done by another proof assistant, namely, Panda [6], which indicates that it is possible to create a format for natural deduction proofs that is then saved in a data format file such as XML or JSON. For this criteria, the main concerns are the format in which to save the exercise, where to store them and how to load them.

Lastly, feedback. It seems most proof assistants have this criterion as a second thought, with the feedback offered by many ranging from none to minimal. Still, there are a few who provide it, and notes can be taken from them. Some assistants show which parts of the proof are valid and which are not. Assistants that require the proof to be written via textual input show feedback when something is incorrectly typed, ranging from pointing out errors to explaining what was expected to be written. Some provide hints on how to start the proof or how to continue in case the student gets stuck. An interesting way of providing feedback is the way the Iltis MOOC does it. Feedback is generated and provided to users based on a set of rules. The users can also control the amount of feedback they receive. All of these and more can be considered, as the quality of feedback given, as well as how it is given, is important to the usefulness of the tool for teaching purposes. Not giving enough feedback can cause users to get stuck in exercises without a way to proceed while giving too much feedback can lead to users defaulting to it as soon as they face any difficulties, which hinders the learning process of a student.

Following this research, it was decided that a tool that could complement an in-person logic course should be capable of fulfilling all of these criteria:

- Capable of visually representing proofs in a Gentzen Tree format.
- Support for propositional classical logic and, if possible, first-order classical logic.
- The logical reasoning must be sound and complete.
- Capable of creating, saving and loading exercises.
- Capable of providing feedback to the user.
- Online, to later connect it with online platforms.

A search for a tool that was already available was done and while most proof assistants are capable of fulfilling some of the criteria, and do it well, none of the ones found during the search could fulfill all of the criteria. With this being the case, it was concluded that a tool must instead be developed.

1.3 Contribuitions & Objectives

The expected contributions of this thesis are:

- The development of an online tool that can help students better learn and understand the concepts of natural deduction by acting as a complement to an in-person course.
- The implementation of feedback mechanisms for the online tool.
- The implementation of an exercise manager capable of creating, saving and loading exercises for the online tool.
- The writing of a paper document that includes an explanation of the implementation process.

1.4 Document Structure

There are five more chapters in this document. They are divided into chapters 2, 3, 4, 5 and 6.

Chapter 2 covers background information that serves to explain and contextualize some of the concepts that will be used throughout the document. This includes propositional and predicate logic, natural deduction and proof assistants.

Chapter 3 covers related work, describing some of the online logic teaching tools that have been studied and analyzed. At the end of this chapter, there is an overview section that measures these to some of the criteria proposed earlier in Chapter 1.

Chapter 4 covers the architecture and implementation of the system. It starts by showing a diagram with the structure and components of the architecture, followed by a section detailing the back-end of the system, which includes explanations for components such as the logic reasoner and the exercise manager. It is concluded with a section detailing the front-end of the system and a demonstration/tech demo of the tool in action.

Chapter 5 covers the testing of the tool. It begins by explaining the system testing done throughout the development of the tool as well as some of the errors that were found during it. It also contains a section regarding user testing.

Chapter 6 concludes the document and contains a general reflection on the work done, the results of the work and thoughts on what could be further done with the tool in the future.

2

Background

This section will introduce and describe some necessary basic notions of logic. It introduces a short explanation of propositional and first-order classical logic, a section distinguishing classical and intuitionistic logic, a description of natural deduction, complete with the proof rules for propositional and first-order logic, its two most prominent proof formats as well as an introduction to the concepts of soundness and completeness. Finally, it introduces the concept of proof assistants in addition to introducing Coq and Isabelle/HOL, two of the most widely used proof assistants.

2.1 Classical Logic

Classical logic, also known as non-constructive logic, is a type of logic in which every formula is always assumed to be either true or false, but not both at the same time. Classical logic distinguishes itself by including the Law of Excluded Middle (LEM) [20], which describes the fact that a formula is either true or false, even if its actual value is unknown, and tends to be represented as such:

$$A \vee (\neg A)$$

2.1.1 Classical Propositional Logic

Propositional logic is a branch of logic that focuses on propositions and the relations between these propositions. We can make formulas utilizing atomic propositions, such as "I am alive"or "She is sleeping", which can only be true or false. Their content is irrelevant, and as such, they are denoted only by capital letters [9]. We can relate propositions via the use of connectors, which are:

- Conjunction ($\wedge$).
- Disjunction ($\vee$).
- Negation ($\neg$).

- Implication ($\rightarrow$).
- Equivalence ($\leftrightarrow$).

Propositions and connectors fall under the syntax of propositional logic. They help us define and structure sentences, but this is not enough to give meaning to the sentence. For that we have semantics. For propositional classical logic, semantics are given by the truth value of formulas or, in other words, if the formula is true or false. When evaluating formulas in classical logic, it is typical to construct truth tables, tables that test the formula for every possible scenario that can occur depending on the truth value of the subformulas that compose the formula. Fig 2.1 showcases, as an example, the truth tables of conjunction and disjunction. The truth table of a formula can give us some information about characteristics such as satisfiability and validity. Satisfiability indicates that there's at least a model where the formula is true, which is shown in a truth table when at least one of its lines has a 1 (true). Validity indicates that the formula is true for every scenario, and is shown in a truth table when all of its lines have a 1.

Another important concept regarding semantics is that of semantic consequence, often denoted by the symbol $\models$. In propositional logic, it defines a relationship between formula G and a set of formulas F. A formula G is said to be a semantic consequence of a set of formulas F if G is true whenever the formulas of F are also true. This can be denoted by F $\models$ G.

p	q	$p \wedge q$
0	0	0
1	0	0
0	1	0
1	1	1

p	q	$p \vee q$
0	0	0
1	0	1
0	1	1
1	1	1

Figure 2.1: Truth tables showcasing formulas with the AND and OR connectors.

2.1.2 Classical Predicate or First-Order Logic

Predicate or first-order logic is an extension of propositional logic that deals with predicates, expressions that can only return true or false and have one or more variables, which in turn can be quantified or assigned values. First-order logic counts with various symbols, which can be separated into different groups. The following three groups represent elements in a particular domain of interest:

- Constants, which represent a specific element of a set. These symbols tend to be letters from the start of the alphabet.

- Variables, which represent arbitrary elements of a set. These symbols tend to be letters from the end of the alphabet. Variables are considered bound if they are quantified, or free if not.
- Functions, which take terms and produce other terms. These symbols tend to be written in lowercase.

There are three more groups which differ from the previous:

- Propositional connectors, such as $\wedge$ and $\vee$.
- Quantifiers, which are used to quantify things in a certain domain. In this case, we have $\exists$, meaning "there exists at least one"and $\forall$, meaning "for all".
- Predicate symbols, which denote relations between terms. These symbols are represented by predicates and tend to be represented by having their first letter be a capital letter.

Complex formulas can be made by combining symbols from these different groups. For example, the expression "All cats are sneaky", which cannot be expressed under propositional logic, can be expressed in first-order language thanks to the existence of quantifiers and relations, as $\forall$x (Cats(x) $\rightarrow$ Sneaky(x)).

Just like with classical propositional logic, this only denotes the syntax of first-order classical logic. It is still necessary to establish the semantics for first-order logic, which is more complex than that of propositional logic. In this context, we must do the following:

- Indicate the domain or universe of the elements being considered.
- Interpret the domain or universe's function symbols.
- Interpret the predicate symbols (relationships) that involve those elements.

The interpretation of function and predicate symbols in a certain domain can be called interpretation structure. Additionally, we need to interpret the variables of the domain. To do this, we consider variable assignments that map a set of variables V to an interpretation structure M, which associates X's variables to elements of the M's universe. With this, we can now evaluate formulas of first-order logic for characteristics such as satisfiability or validity. To achieve satisfiability, a formula must be true in at least one interpretation structures **and** at least one of its variable assignments. To achieve validity, a formula must be true in all of its interpretation structures and all of its variable assignments.

Semantic consequence is slightly different in first-order logic when compared to propositional logic. If a formula ϕ can only be satisfied by interpretation structure M and all of the variable assignments of a variable set in M if another formula θ can also be derived from the same struct M and its attributions, then ϕ is a semantic consequence of θ, indicated by the notation $\theta \vDash \phi$.

2.2 Intuitionistic Logic

Intuitionistic logic, also known as constructive logic, is another type of logic that follows intuitionistic mathematics and doesn't assign formulas with values of true or false. Instead, formulas are either provable or disprovable, as intuitionists believe that the verification of formulas holds much importance. For example, a formula can only be asserted as true when it has been verified as such [20]. This follows constructive reasoning, which seeks a definitive, explicit answer to questions.

Unlike classical logic, intuitionistic logic rejects the LEM, as well as the rules and characteristics that can be derived from it, which makes some proofs that would otherwise hold for classical logic invalid. In summary, intuitionistic logic rejects the following rules in addition to LEM:

- Negation ($\neg$) acts as a toggle between true and false.
- Double negation ($\neg\neg$) elimination theorem.
- Reductio ad Absurdum theorem, also known as proof by absurdity ($\bot$).

2.3 Natural Deduction

Natural deduction is a type of logical system that was initially proposed by Gerhard Gentzen and Stanislaw Jakowski in the year 1934 [22]. It is a method that uses proof systems, which are composed of inference or proof rules that are applied to formulas, to prove or derive new formulas with the intent to prove the validity of a goal. These rules and their application are designed in a way that tries to behave how a person normally reasons and draws conclusions. Different natural deduction systems can differ in aspects such as proof format, the approach used to construct the proof and even which rules they use. Proofs in natural deduction can be built from the bottom up (backward reasoning) or from the top down (forward reasoning). Gentzen Tree and Fitch style proofs are two natural deduction proof formats that are widely used in logic teaching and proof assistants.

2.3.1 Proof Formats

As previously mentioned, there are some different ways to represent proofs in natural deduction. This section will cover the Gentzen Tree and Fitch style formats.

2.3.1.1 Gentzen Tree

In tree-style proof systems, the root of the tree is the formula to be proved (the goal), and its leaves are premises or assumptions, with the tree growing upwards. Each "branch"of the tree represents a different subproof. Applying rules to formulas in a branch of the tree will grow the branch upwards and generate new subgoals, separating the new subgoals

$$\dfrac{\dfrac{\dfrac{\psi_1\,{}^{1}\quad \psi_1 \to (\psi_2 \wedge \psi_3)\,{}^{2}}{\psi_2 \wedge \psi_3}\to E}{\psi_2}\wedge E_d}{\psi_1 \to \psi_2}\to I,1 \qquad \dfrac{\dfrac{\dfrac{\psi_1\,{}^{3}\quad \psi_1 \to (\psi_2 \wedge \psi_3)\,{}^{4}}{\psi_2 \wedge \psi_3}\to E}{\psi_3}\wedge E_e}{\psi_1 \to \psi_3}\to I,3$$

$$\dfrac{(\psi_1 \to \psi_2) \quad (\psi_1 \to \psi_3)}{(\psi_1 \to \psi_2) \wedge (\psi_1 \to \psi_3)}\wedge I$$

Figure 2.2: An example of a tree-style proof system.

from the existing formula with a line that shows the rule and marks used. Hypotheses can be assumed when needed, as long as they are properly marked and discharged.

Marks are a way to keep track of a formula by assigning them some sort of identification, most commonly numbers. All marked formulas are either open (undischarged) or closed (discharged), depending on whether the mark has been eliminated by a proof rule or not, with only certain rules being able to eliminate proof rules as shown prior. A proof can only be considered complete when all of its non-premise formulas have been closed.

An example of a tree proof can be seen in Figure 2.2. The goal of this proof is $(\psi 1 \to \psi 2) \wedge (\psi 1 \implies \psi 3)$ and there's a premise that is $\psi 1 \to (\psi 2 \wedge \psi 3)$. All instances of the premise in the proof are marked and will remain open as premises cannot be closed. At some point in the proof, $\psi 1$ is assumed in both branches of the tree and properly marked with different numbers so as not to conflict. As these are assumptions and not premises, they must eventually be closed, which they are when the rule →I, or introduction of implication, is applied, referencing the mark for the assumption. This eliminates the mark and closes the hypothesis.

It should be noted that in this format, a rule that can eliminate marks can still be applied without eliminating any mark, which can prove useful in certain situations.

2.3.1.2 Fitch

(1)	P ∧ Q		*hypothesis*
(2)	R		*hypothesis*
(3)	Q	(1)	*(∧rE)*
(4)	Q ∧ R	(3)(2)	*(∧I)*
(5)	R ⇒ (Q ∧ R)	(2) … (4)	*(⇒I)*
(6)	(P ∧ Q) ⇒ (R ⇒ (Q ∧ R))	(1) … (5)	*(⇒I)*

Figure 2.3: An example of a Fitch style proof system.

Fitch-style proof systems follow a linear representation, with each line containing a formula and either the rule (and necessary formulas for it to be valid) from which it was

inferred or, in the case of assumptions, an indication that this is the case. It differs from Gentzen trees in the following aspects:

- No marks to represent assumptions.
- Formulas can be inferred from a single assumption.
- To apply a proof rule there must be a reference to all lines or subproofs needed to make it valid.

The lack of marks leaves the system with problems concerning keeping up with and representing open and closed formulas. To solve this issue, the Fitch system utilizes subproofs, in which a new box is opened in a proof every time an assumption is made. The assumption is open while inside the scope of the box and is closed on all lines that come after the final line of the subproof. Subproofs can also contain different subproofs within themselves, which have their assumptions and scope. If the assumption of an outer subproof must be used on an inner subproof, a reiteration rule can be called, referring to the formula that is being repeated, though most Fitch-style proof systems simply refer to the original assumption if it's needed to be used in the application of a proof rule.

While in Gentzen trees there's sometimes a need to assume the same formula in different branches, for example assuming a conjunction twice so we can get both of its sides, Fitch-style systems only need a single assumption, which can be referenced at any time and any number of times as long as it is done inside the scope of its subproof.

An example of a Fitch-style proof is shown in Figure 2.3, which was taken from the Edukera [16] tool. This proof only has a conclusion, which is in line 6, and no premises. It has two subproofs, one inside the other, each with its assumption, which is lines 1 and 2, respectively. The application of proof rules references the lines for the formulas that make them valid, with lines 5 and 6 being justified by entire subproofs.

2.3.2 Proof Rules

In this section, some proof rules used for classical propositional and first-order logic natural deduction are shown. The rules have properties such as arity, which counts the number of formulas required to apply the rule, and mark elimination, or hypothesis elimination, which means the rule removes a mark from one of the formulas, which closes that formula.

2.3.2.1 Propositional Logic

It should be noted that instead of introducing and eliminating negation, the concept of contradiction or absurdity ($\bot$) can be used. Just like them, it will eliminate a mark.

Figure 2.4 shows a graphical representation in the tree-style format of the propositional logic proof rules.

Acronym	Rule	Arity	Eliminates Marks
$\wedge$I	Introduction of Conjunction	1	
$\wedge$E	Elimination of Conjunction	1	
$\vee$I	Introduction of Disjunction	1	
$\vee$E	Elimination of Disjunction	3	✓
$\leftrightarrow$I	Introduction of Implication	1	✓
$\leftrightarrow$E	Elimination of Implication	2	
$\rightarrow$I	Introduction of Equivalence	2	✓
$\rightarrow$E	Elimination of Equivalence	2	
$\neg$I	Introduction of Negation	2	✓
$\neg$E	Elimination of Negation	2	✓

Table 2.1: Propositional logic proof rules.

2.3.2.2 First-Order Logic

This first-order logic retains the proof rules from propositional logic but adds to it four more rules, two for each quantifier, which are:

Acronym	Rule	Arity	Eliminates Marks
$\exists$I	Introduction of Existential	1	
$\exists$E	Elimination of Existential	2	✓
$\forall$I	Introduction of Universal	1	
$\forall$E	Elimination of Universal	1	

Table 2.2: First-order logic proof rules.

Unlike propositional logic rules, all of the first-order logic rules have pre-conditions that must be fulfilled to make the rule application valid. To understand these pre-conditions, we must first look at the graphical representation of the first-order proof rules:

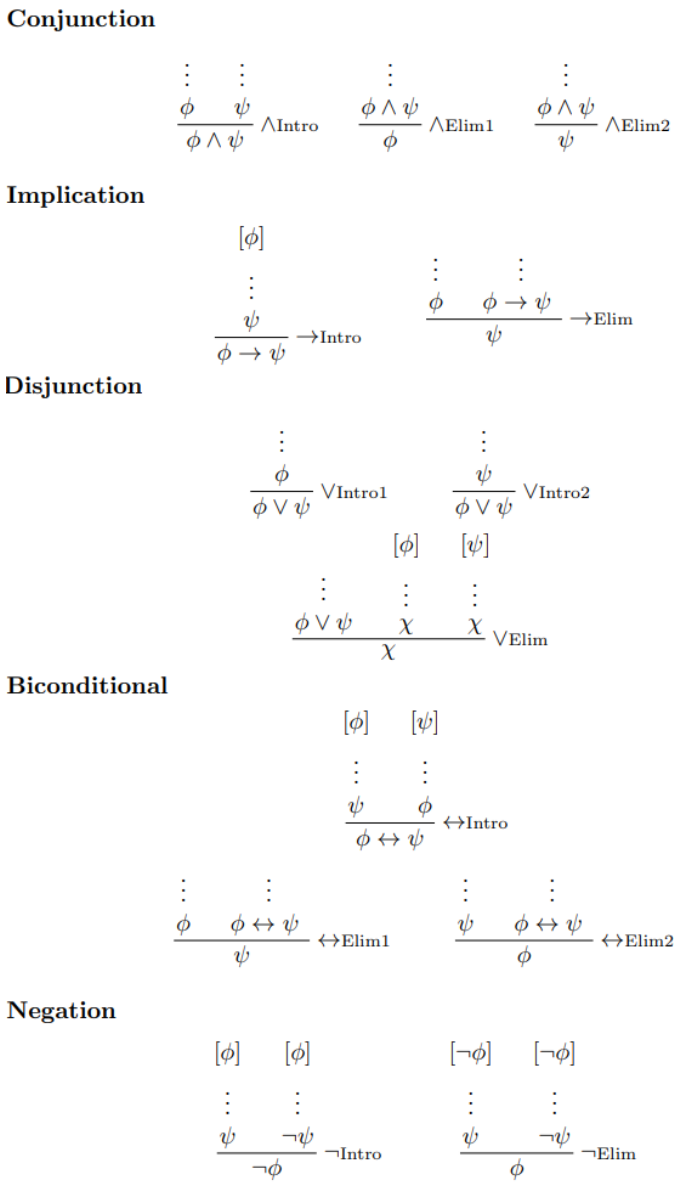


Figure 2.4: Proof rules for propositional logic. Taken from the NDPack by Alastair Carr [28].

$$\frac{\begin{array}{c}\mathcal{D}\\ [\varphi]_t^x\end{array}}{\exists_x \varphi}\ (\exists_I)$$

$$\frac{\begin{array}{ccc}\mathcal{D}1 & & ([\varphi]_y^x)^m \\ \exists_x \varphi & & \mathcal{D}2 \\ & & \psi\end{array}}{\psi}\ (\exists_E, m)$$

$$\frac{\begin{array}{c}\mathcal{D}\\ [\varphi]_y^x\end{array}}{\forall_x \varphi}\ (\forall_I)$$

$$\frac{\begin{array}{c}\mathcal{D}\\ \forall_x \varphi\end{array}}{[\varphi]_y^x}\ (\forall_E)$$

The pre-conditions are as follows:

- For $\forall_E$ and $\exists_I$, t has to be free (not bound by a quantifier) for x in φ.
- For $\forall_I$
  - y must not appear free in open hypotheses of D.
  - if x ≠ y, then y ∉ V L(φ).
- For $\exists_E$
  - if x ≠ y, y can not appear free in φ.
  - y can not appear free in ψ or the open hypotheses of $\mathcal{D}2$ that are not $[\varphi]_y^x$.
  - the mark m can only (eventually) close hypothesis $[\varphi]_y^x$ in $\mathcal{D}2$.

The concept of substitution is also important for first-order logic natural deduction. The application of the elimination rules substitutes quantified formulas for non-quantified formulas by assuming a constant in place of the quantified variable. Likewise, the application of introduction rules does the opposite, turning non-quantified formulas into quantified formulas. As such, applying these rules has the added challenge of knowing which variables to substitute in or out.

2.3.3 Soundness & Completeness

Soundness and completeness are properties of a deductive system. Due to the nature of these properties, we can use them to measure how strong the deductive system is, as the lack of these properties in a system will diminish its worth as deductions from it cannot be fully trusted.

For a system to be sound, it must hold the following property:

If a formula G can be derived from a set of formulas F, then G is a semantic consequence of F.

For a system to be complete, it must hold the following property:

If a formula G is a consequence of a set of formulas F, then G can be derived from F.

Natural deduction systems that are composed of the proof rules which were previously shown can be proven both sound and complete [2].

2.4 Proof Assistants

Proof assistants are tools that serve to assist with deductive proofs. Most assistants for first-order logic exist for teaching and assisting students who are learning logic. Several of these are shown in the next chapter, Related Work, where their features are shown and compared. The previously mentioned proof assistants are more specific in their purpose, and often base themselves on more generalist proof assistants which, while still capable of assisting in teaching, are more geared towards proving mathematical theorems and other problems of the sort. This section will show two of the most widely used proof assistants that fit into the latter: Coq and Isabelle/HOL.

These two assistants are generally more complex and complete than others, characteristics which come with the added need of learning their syntax in addition to knowing logic and mathematical concepts, as they are capable of working with several deductive systems thanks to their access to libraries and add-ons.

Coq and Isabelle/HOL share several characteristics, which, according to Bartzia [3] and Yushkovskiy [19], are:

- Open source and still actively maintained and developed.
- Proofs are built via textual input.
- Offer little feedback, with just small messages whenever a rule does not apply or when an expression isn't typed correctly.
- Only show the current subgoals, never showing the full proof.
- Allow navigation of the steps of the proof to see what the state of the proof was directly after application of the step.

- Both have been used to formalize and prove many mathematical theorems.
- A large number of libraries filled with already proven lemmas.
- Can function as a functional programming language.
- Support forward reasoning.
- While Yushkovskiy only mentions Isabelle supporting backward reasoning, both support it [36].

Despite sharing these characteristics, these assistants have major differences regarding other aspects.

2.4.1 Coq

Coq has been developed since 1984 by INRIA and was first released in 1989. It is based on the Calculus of Inductive Constructions [19] and as such, was implemented with intuitionistic (constructive) logic in mind. It utilizes a command-based language, with the commands being called tactics. These tactics, which follow backward reasoning, are applied to proofs, changing their state and can, in a way, be seen as similar to inference rules. Coq logic can be extended to classical logic by adding the Law of Excluded Middle to it via libraries. Due to its command-based language, proofs end up taking forms that usually don't resemble how they would be done by hand. It does, however, attempt to ease this difference by identifying and showing variables and hypotheses in the scope of the proof at each of its steps.

In Figure 2.5, we can see a complete proof done in a Gentzen tree-style format and a possible way of writing it in Coq's online IDE jsCoq [24], which was done by hand.

$$\dfrac{\neg P \vee Q \qquad \dfrac{\dfrac{[P]^2 \quad [\neg P]^1}{Q}\,\neg\text{Elim} \qquad [Q]^1}{Q}\,\vee\text{Elim}^1}{P \to Q}\,\to\text{Intro}^2$$

```
Theorem Core5 : ∀ P Q : Prop,
  ~P ∨ Q → P → Q.
Proof.
  intros.
  unfold not in H.
  destruct H as [H1 | H2].
  apply H1 in H0.
  exfalso. apply H0. apply H2.
Qed.
```

Figure 2.5: The core 5 proof from NDPack [28] and a possible way to write it in Coq.

2.4.2 Isabelle/HOL

Isabelle/HOL was developed by Lawrence Paulson and Tobias Nipkow and was initially released in 1986. It is built around classical higher-order logic, an extension of predicate logic with additional quantifiers and semantics. In Isabelle proofs are built by applying

proof rules and tactics, not to be confused with Coq's tactics as these are closer to functions. The application is done utilizing the apply command. The proof rules are separated into introduction and elimination rules, being applied with the keywords "rule"and "erule", respectively. They also differ in the way they are applied to the proof, with introduction rules being applied to the current subgoal's conclusion, and the elimination rules being applied to the hypotheses of the subgoal. As the proof is built, the assistant will generate the necessary assumptions, which the user can apply by utilizing the assumption tactic. While Isabelle also shows all the hypotheses of a subgoal, it doesn't explicitly identify them as Coq does. It does, however, closely resemble a textual version of a proof done by hand, as the application of proof rules is explicitly shown in a way that closely resembles the usual proofs.

Figure 2.6 shows a complete proof and a possible way of writing it in Isabelle's IDE, which was done by hand.

$$\dfrac{\dfrac{[P\vee Q]\qquad \dfrac{\dfrac{\dfrac{(P\to Q)\wedge(Q\to P)}{P\to Q}\wedge\text{Elim}\qquad [P]}{Q}\to\text{Elim}\qquad [P]}{P\wedge Q}\wedge\text{Intro}\qquad \dfrac{\dfrac{\dfrac{(P\to Q)\wedge(Q\to P)}{Q\to P}\wedge\text{Elim}\qquad [Q]}{P}\to\text{Elim}\qquad [Q]}{P\wedge Q}\wedge\text{Intro}}{P\wedge Q}\vee\text{Elim}}{(P\vee Q)\to(P\wedge Q)}\to\text{Intro}$$

```
lemma "(P ⟶ Q) ∧ (Q ⟶ P) ⟹ (P ∨ Q) ⟶ (P ∧ Q)"
  apply (rule impI)
  apply (erule disjE)

   apply (rule conjI)
    apply  assumption
   apply (erule conjE)
   apply (rule mp)
    apply assumption
   apply assumption

  apply (rule conjI)
   defer
   apply assumption
  apply (erule conjE)
  apply (rule mp)
   apply assumption
  apply assumption

  done
end
```

Figure 2.6: The disjunction 13 proof from NDPack [28] and a possible way to write it in Isabelle.

3

Related Work

This chapter will present some tools with a focus on teaching logic, both online and installable. Not all of them are geared only towards the teaching of logic, but all have the option available at the least. Apart from the very first one shown, every other tool in this section has natural deduction available, with many having their sole focus placed on assisting with the teaching of natural deduction. There is also an Overview section at the end which compares the tools according to certain metrics such as format, exercise creation and feedback.

3.1 Logic Teaching Tools

3.1.1 Iltis

Iltis [7, 8] is a kind of MOOC developed by the University of Dortmund for teaching logic, which was very rare even just a few years ago [11]. Currently, it features learning material and exercises for Propositional and Modal logic, as well as some First-order logic. It is available for free to anyone who wants to use it.

Iltis' main features are its "composite task model" and "Sample feedback strategies and generators" according to some of the project's main contributors [8]. Regarding its composite task mode, Iltis makes up its several exercises by making several smaller tasks, which then are composed with its other. These tasks take an input and produce an output, which can then be used as the input for the following tasks. According to its contributors, Iltis allows teachers to create exercises using this task model, by specifying the various tasks in an XML file, as shown in Figure 3.1. According to its contributors [7], Iltis also supports anonymous data collection. This comes in the form of loggers that come with tasks that collect data that can be used for various purposes.

Also specified in this XML file are the feedback generators and their rules for the tasks. Iltis features a feedback system in which each task is assigned a feedback strategy and several feedback generators, each one producing different kinds of feedback according to the rules given to them. This feedback is then displayed to the student in a predetermined order, with each piece of feedback given being from a different feedback generator. The

```
<!-- State formulas for the statements -->
<Task type="CreateFormulas" feedbackLevels="0,1,2"
      assimilationGenerator="syntaxServer">
  <Input>VARIABLES</Input>
  <Title>Step 2: Modelling the statements</Title>
  <Description>
    Devise a formula for each observation! Use the propositional...
  </Description>
  <Formula>
    <Description>
      If the database is faulty then so is the back end.
    </Description>
    <Solution>D → B</Solution>
  </Formula>
  <Formula>
    <Description>
      The back end is only faulty if both the database and the user...
    </Description>
    <Solution>B → (D ∧ U)</Solution>
  </Formula>
  <Formula>
    <Description>Not all three components are faulty.</Description>
    <Solution>¬(B ∧ D ∧ U)</Solution>
  </Formula>
  <!-- Feedback generator -->
  <FeedbackGenerator>
    <Feedback type="VariableNames">
      <Variable name="U">the user interface</Variable>
      <Variable name="B">the back end</Variable>
      <Variable name="D">the database</Variable>
    </Feedback>
  </FeedbackGenerator>
  <Output>FORMULAE</Output>
</Task>
```

Figure 3.1: An example of the task specification for an exercise in Iltis [7].

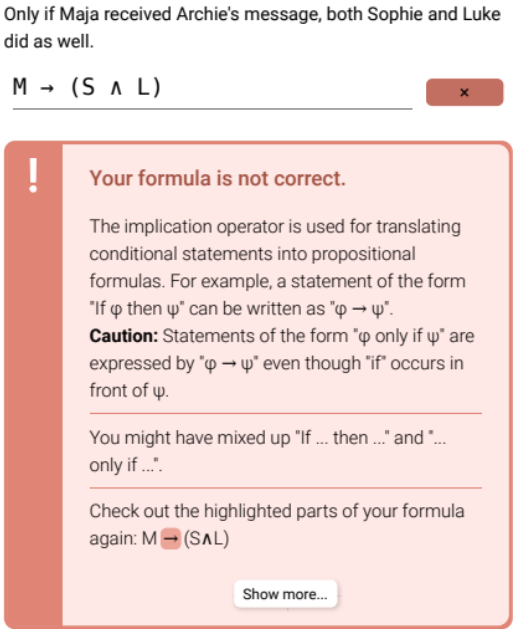


Figure 3.2: An example of the way feedback is given in Iltis [8]. The outputs given by each feedback generator are divided by a line.

student is also allowed to control the amount of feedback given to them, with the system providing a small amount of feedback and an option for the student to receive more, different feedback (if available) as can be seen in Figure 3.2. This system allows for control of feedback to both the professor, when specifying the task, and the student, depending on their difficulties and preferences. At the time of writing, however, Iltis doesn't feature any kind of natural deduction exercises. It does however offer great examples in regards to exercise creation and creation and distribution of feedback, as well as being hosted online.

3.1.2 Edukera

Edukera [16] is an online tool whose purpose is to teach students logic and math, as such, it contains several different topics, such as calculus, formalization and logic. This will focus on the topic of logic, for which Edukera presents sets of premade tutorials and exercises for both propositional (Connectors) and first-order languages (Quantifiers).

The provided tutorials and exercises are proofs in which the student must select the correct proof rules, which are present on the left side of the page and can be selected by clicking, for each step to correctly solve the exercise, occasionally having to justify the chosen rule, as shown in Figure 3.5. After selecting a proof rule the tool automatically constructs the next step of the proof following the structure needed to make the rule valid. Textual input is only optionally required when justifying a proof rule, however, there are on-screen buttons to write terms instead. Two different proof formats of the proof system are available in Edukera for logic exercises, those being Fitch style and a non-Gentzen

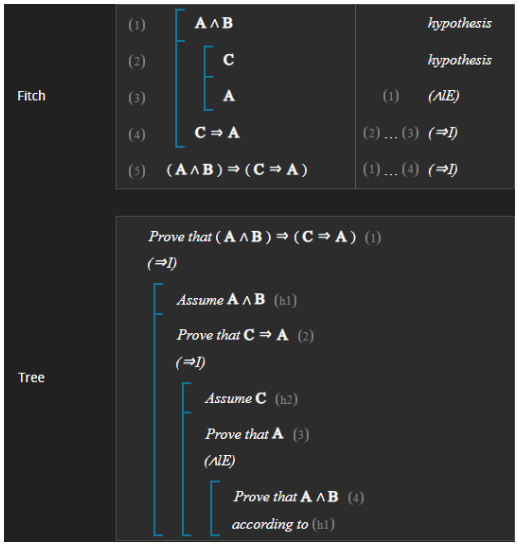


Figure 3.3: The proof formats that can be used for logic exercises in Edukera.

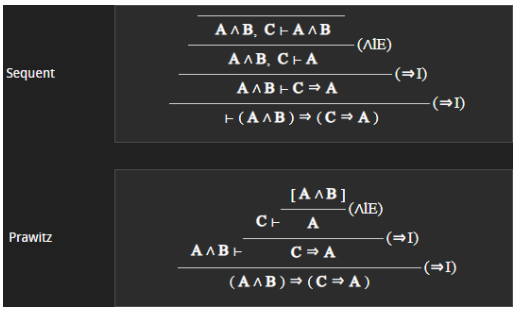


Figure 3.4: The proof formats that can only be visualized separately for logic exercises in Edukera. They can be used in mathematical exercises.

tree-style format, which isn't a Gentzen tree, shown in Figure 3.3. An additional two, Sequent and Prawitz, are shown in Figure 3.4 are also available but for visualization only for Logic exercises. Explanations of the rules and their structure are also provided by clicking the magnifying glass icons next to the specific rule. When selecting the first tutorial, the tool will give a step-by-step guided explanation of how it works and how to use it. It should also be noted that Edukera uses Coq as a "mathematical engine"[16] to verify their proofs.

There are some downsides to Edukera. To start, it is unfortunately no longer maintained [3] and as such, it cannot be expected for more features to be added to it. The tool also doesn't offer much feedback, with the provided feedback amounting to checking if an attempt at using a proof rule is valid for the current goal and, upon completion of the proof, showing a green checkmark alongside a button to access the next exercise.

Finally, while Edukera allows a user to create a class, which students can enroll in

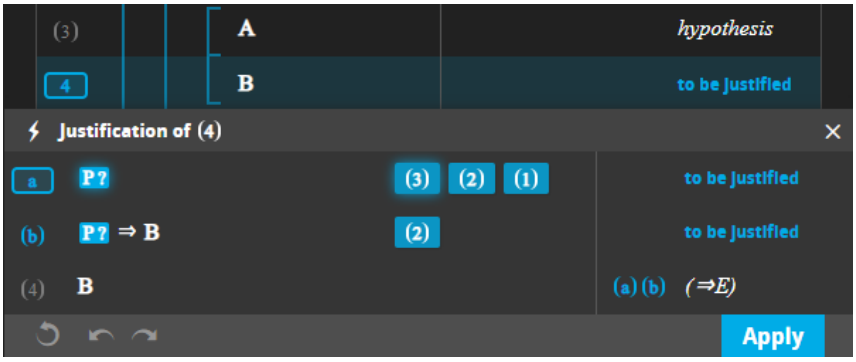


Figure 3.5: Whenever a proof rule needs to be further justified in Edukera, a box such as this one will appear, which must be complete with the correct terms to continue with the proof.

with the given credentials, and have the teacher select exercises for the class to complete, as well as homework and evaluations, the exercises available are those that were already available on the tool, with no possibility for the teacher to create completely new exercises on the tool.

3.1.3 Holbert

Holbert [15] is an open-source project described by its creators as a "work-in-progress pedagogical proof assistant and online textbook platform". It functions fully online, and its main aim is to help teach programming language theory. Holbert's combination of an online textbook and a proof assistant allows students to follow the topics while solving exercises without needing to switch between two different tools.

Holbert's text editor allows a user to create their theorems, axioms, inductive definitions and more. Its key features are:

- Gentzen tree format.
- Goal tag.
- Automatic proof building.
- Creation of exercises, including partially solved ones which might make good beginner exercises.

It provides graphical representations of terms, rules and proofs, allowing the creation of exercises in the form of solving proofs, which are represented as Gentzen trees. For natural deduction exercises, creating a proof involves first establishing a theorem, giving it a name, and then writing the conclusion and the premises of the theorem using Holbert's term language, which is in the form of "untyped lambda calculus"[15]. After creating the theorem, the tool will automatically generate a proof system object, which will display the conclusion of the proof and a goal tag, which is shown in Figure 3.6. The proof is then solved backwards, by clicking on the goal tag, which displays on the right side of the screen, the current goal, the available rules and possible assumptions, as shown in Figure 3.8. Upon selecting a rule, the assistant will automatically generate the structure needed for that rule to be valid, as well as the necessary goal tags for each newly generated subgoal. All of this is done using mouse clicks. The proof will be considered complete once there are no more goal tags present as shown in Figure 3.7.

While its key features are great, Holbert does lack in other aspects. Holbert's term language can start to get confusing whenever writing larger formulas containing several connectors, further amplified by the lack of, for example, buttons that would automatically type out a connector or a term, present in tools such as Fitch [4], which will be introduced later in this section. It should also be noted that, according to its authors, the way the term language is written technically makes its logic unsound [15]. Holbert also noticeably lacks

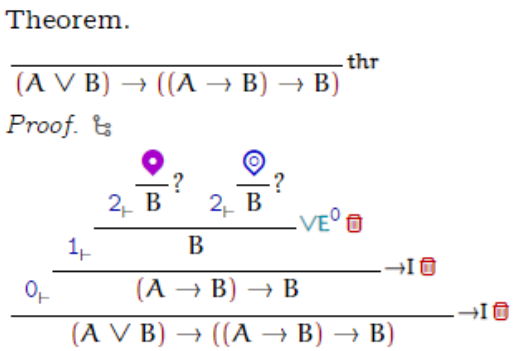


Figure 3.6: A partially built proof in Holbert, showing its goal tags which are present for every remaining subgoal.

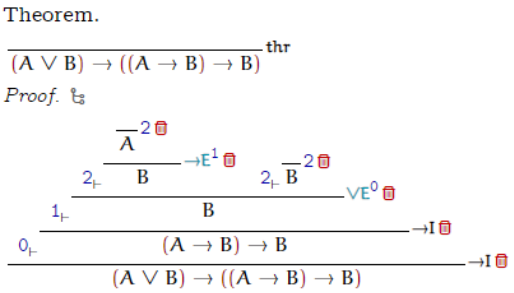


Figure 3.7: A complete proof in Holbert, marked by the lack of any goal tag.

any kind of feedback besides the disappearance of all goal tags once the proof is complete. There is no help given by the assistant to a student who gets stuck on a part of the proof even after multiple attempts, with the student having to resort to asking for help from a teacher or simply using another proof assistant that provides some feedback.

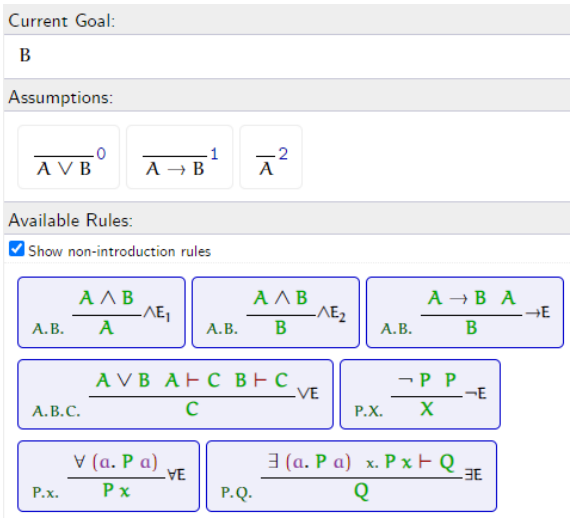


Figure 3.8: Menu displayed by Holbert when clicking on a goal tag.

3.1.4 Fitch

Fitch is a tool provided alongside the Language, Proof and Logic book [4] and whose purpose is to construct and solve formal proofs in first-order logic. It comes in the form of a local application, which can be run locally on a computer, and a set of premade exercises that come from the textbook.

When opening the application, an interface is shown to the user containing a section where the proof is written, in Fitch style format, a section that states the goal(s) of the proof and a set of buttons which when pressed type out the corresponding term or connector.

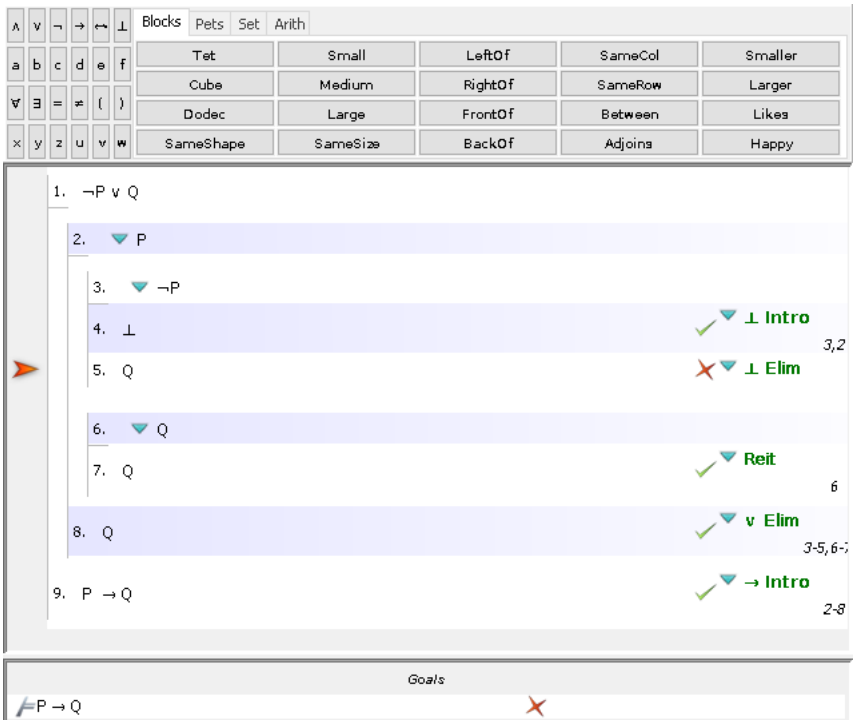


Figure 3.9: An example of Fitch's interface, proof structure and feedback. The tool can show which statements cause the proof to fail.

To write in Fitch the user must move the red cursor towards the step they want to write on and then write, utilizing the term buttons on top when necessary. To construct a proof the user has the following options:

- Add premises by using the Ctrl + R.
- Add and delete goals by accessing the Goal menu in the toolbar.
- Add and delete steps of the proof with Ctrl + A for add after, Ctrl + B for add before and Ctrl + D to delete, being possible to write the formulas as well as select the proof rule for that step.

Additionally, under the Edit section of the toolbar, an option called Author Mode can be toggled. When off, the user is limited to only actions that a student can do to solve the proof. On the other side, to solve a proof a student can do the following actions:

- Add and delete steps of the proof, in the same way as mentioned above.
- Add and end subproofs with the Ctrl + P and Ctrl + E hotkeys, respectively. These function just like regular proof, with the possibility of adding steps, writing on them and selecting proof rules.
- Verify the proof with the Ctrl + F hotkey. This will also provide feedback to the student.

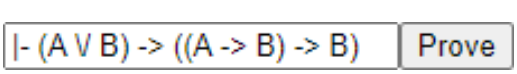
With these actions, the student then needs to write the necessary steps and subproofs (when needed), and apply the proof rules, which require the user to put the red cursor on top of the step which is being proven followed by selecting the steps that make the proof rule valid and then select the rule, and finish by verifying the proof. An example of a proof in Fitch, showing its buttons for terms, proof structure, feedback provided and more are shown in Figure 3.9.

Fitch allows easy construction of a proof, which is a very valuable feature, but not the only one. Another strong feature of Fitch is the feedback it provides, as it will individually check each part of the proof for validity and correctness, displaying a green checkmark if the step is well written and correct in the context of the proof or a red cross in case something fails. While it may look simple, this provides a great amount of information to the user, who will be made aware, in case of an error, where this error is and which steps are causing it.

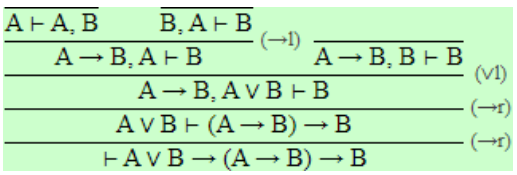
3.1.5 Logitext

Logitext [18] is a tool available online that functions as an educational proof assistant. Logitext utilizes sequent calculus and is intended for students learning proof systems with a Gentzen tree format. Two versions of the tool are available, one which works with classic first-order logic and another which works with intuitionistic propositional logic. Logitext doesn't offer any special features that distinguish it from other proof assistants, and its existence comes from the developer's intention of reusing pre-existing software as much as possible, in an attempt to show the utility of doing so. As such, Logitext was constructed with a combination of Coq, Haskell and Ur/Web, with Coq being used to check the validity of the proof steps. In the case of the intuitionistic propositional logic version, G4ip was used instead of Coq.

As a proof assistant Logitext requires the user to manually type the theorem that they want to prove, and after clicking the Prove button will redirect the user to a page where the proof is done. To solve the proof, which is done bottom up, the user clicks on the connector or term they want to prove and the assistant will automatically build the next



(a) The textual input for this proof.



(b) The complete proof, with the green background after it's complete and correct.

Figure 3.10: An example of a proof in Logitext.

Figure 3.11: This is how feedback is given in Logitext.

part of the proof, assigning the proof rule that would generate the goal and the subgoals necessary to make it valid. In case the user needs to redo the proof, they can click the turnstile of a subgoal, which resets everything above that part of the proof. An example of a proof in Logitext is shown in Figure 3.10. Apart from the background turning green when completing a proof correctly, Logitext will also give a message when attempting to select a term or connector that isn't valid for the current goal, which is shown in Figure 3.11

Having been released in 2012 [18], this tool doesn't do much different from more recent and visually attractive proof assistants like Edukera and Holbert, but it does manage to fulfill its role in showing the use and utility of reusing pre-existing software in the creation of newer software.

3.1.6 Yoda

Yoda [12] was a tool available online for the Universidad de la República in Uruguay. It seems to no longer be maintained, and isn't even available anymore, as the links where it used to be hosted no longer work. With the tool no longer available, the only information that can be found are in articles. According to the paper written by its developers, it was a tool that focused on natural deduction utilizing the Gentzen tree format, implemented with Javascript, JQuery and CSS which worked with both propositional and first-order logic, each one managed by different assistants.

According to its developers, to build a proof of type $\rho \vdash \varphi$ with Yoda a user starts by specifying the conclusion φ and the premises ρ via textual input. Once submitted, the assistant would show a partially built tree with the conclusion as the root, alongside a drop-down list whenever a rule, premise or formula needed to be specified. The system would then check if the action was applicable. If it was, it would build the corresponding subtree, and if it wasn't, it would present the user with an informative message. Premises are displayed in blue when they appear in the proof itself, and while marks aren't explicitly shown, a formula that has been closed will show up crossed and in red. An example of a proof is shown in figure 3.12.

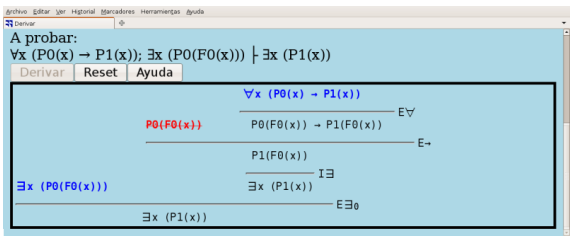


Figure 3.12: Example of a proof in Yoda [12].

3.1.7 Panda

Panda [6] is an open-source proof assistant tool written in Java and created with the intent to help in the task of teaching natural deduction. Its creators were looking for tools that

could fulfill several requirements, which are shown in Figure 3.13, but couldn't find one that suited everything and, as such, decided to develop their own tool. Panda uses the Gentzen tree style for its proofs and the creators argue in favor of this format, saying that "it seems to us that Gentzen's style proofs are more readable because the tree-like structure of proofs (...) is more evident."

- Easy to install and to use;
- Allows one to check a given proof;
- Allows one to build a proof either with some help, or without help;
- Allows both backward and forward reasoning;
- Allows to easily compose proofs from subproofs;
- Uses Gentzen's proof trees style.

Figure 3.13: The requirements needed by Panda's developers for a natural deduction tool. [6]

To create a proof in Panda, the user must click the "Add a formula..."button and write the formula they want to prove. As stated in their requirements, solving a proof can be done with both backward reasoning and/or forward reasoning. To achieve this, Panda allows the creation of partial proof trees, which can then be moved across the screen and connected to other partial proofs to form a single proof. To make sure the logic of the proof remains sound, Panda keeps track of formulas introduced in the branches of proofs and verifies their variable composition [6]. Proof rules are shown on the left side of the interface with a visual representation of how they can be applied, as shown in Figure 3.14. The rules shown are those that can be applied to the currently selected formula, with the other rules being hidden from the user. The assistant will also show the user the possible decompositions of a proof rule, which can be clicked by the user, and the assistant will automatically apply the appropriate rule, which is also shown in Figure 3.14. Panda also supports first-order logic.

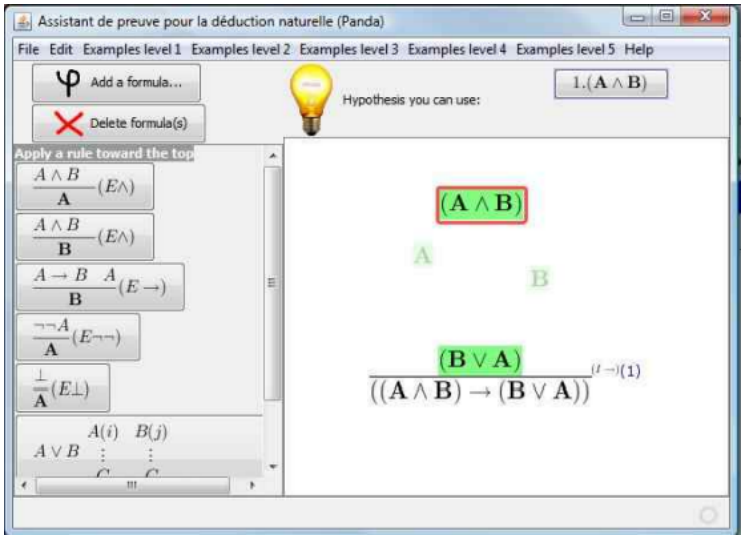


Figure 3.14: Panda's interface, showing proofs, rules and suggestions. [6]

As mentioned prior, the tool allows for the composition of various partial proofs into one. For example, two partial trees can be joined by connecting the root of the first to the leaf of the second. According to the developers, Panda will "use some heuristics"and be able to determine if the trees can be linked via some rule and, if so, apply the corresponding rule. Proofs can also be saved in a LaTeXformat and their specific format, stored as an XML file, which can be reloaded later and can serve as a way to create exercises.

Unfortunately, it seems that Panda is no longer maintained, with the last commit of the source code to GitHub being in 2020 and the latest date of modification found in the source code being from 2013. Both the master application and preview application that come with the source code were built around JNLP files, which seem to have been deprecated in JDK 9 and later removed entirely.

3.1.8 Fitch-Checker

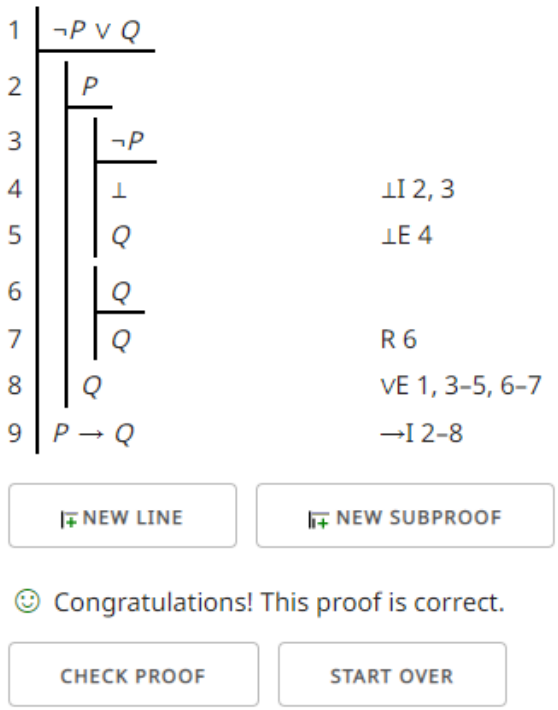


Figure 3.15: Example of a proof in Fitch-Checker.

A simple proof assistant for the proof system used with the Forall x: Calgary book [13]. It is available online [21] for free. The proof assistant, instructions for it, some sets of sample exercises and a visual representation of the proof rules are all present at all times. The visual representation of the proof rules is on the right side of the page and follows those used in the systems taught in the book.

To start a proof, the user must first textually input the premises and conclusion into their specific boxes and then click the create problem button. This will make a proof system appear below it, which follows a Fitch-style notation and, as such, behaves in a

very similar way to Fitch 3.1.4. There are buttons to add new lines and new subproofs. When creating a new line, the user must write down both the formula and the proof rule, the latter of which has to be written in the notation shown on the right side of the page. When creating a subproof, the user must write the formula that is being assumed, and then has the choice to add lines, add another subproof and finish the subproof. Every option is shown as a small button when hovering over a line, on its right side, including an option to delete the line/subproof. Additionally, at the bottom of the proof, there are buttons to start over and check the proof. Checking the proof will give an error when something isn't correctly written (or get stuck in "Checking...") and give a positive message when the proof is correct. An example of a proof is shown in Figure 3.15.

3.1.9 Logan

Logan is an open-source proof assistant developed as part of a thesis project [1] with the intent of being used by students learning logic. It is available for free online [26]. Upon entering the website (or when starting the program) we can see three sections which are Instructions, About the rules and Proof.

In the instructions sections, there are two buttons, one for a manual and another for shortcuts. The manual contains relevant information for the page, explains the layout and then shows a couple of examples of proofs done on the tool, accompanied by photos showing various details that can occur during the writing of a proof. The shortcut tab shows shortcuts for symbols and certain actions such as adding and deleting lines.

The about the rules section contains several buttons which, when clicked, display information about its rule as well as a visual representation of it. It also shows the textual shortcut to write the rule.

The proof assistant section shows 6 buttons, 2 which add lines to the proof, one which resets the proof, one that gives a hint on how to develop the proof at the start, one to save the proof as a pdf and another to export it as LaTeX. The proof assistant uses a Fitch-style format for its system and is suited to handle first-order logic. To start a proof, the user must input the premises and the conclusion into their specific boxes. The assistant will then automatically create a line for each premise, marking them accordingly on the rule slot. After this, the user has to add lines, which are composed initially of two separate boxes, the left one which holds the formula, and the right one which holds the proof rule. To create a subproof, the user has to assume a formula by writing it on the left box and then write "as"on the rule slot, which is a shortcut for "Assumption.". The assistant will automatically write it down as an assumption and give visual feedback to show that it's now a subproof. As mentioned prior there are two buttons to add lines. The difference between these is that one will create lines inside the subproof, while the other will create the line outside of the subproof. To delete a line, the user has to click on the respective line's formula box, delete whatever is written there and then click backspace again. When writing a proof rule on the right box, the box will adapt to the arity of the proof rule,

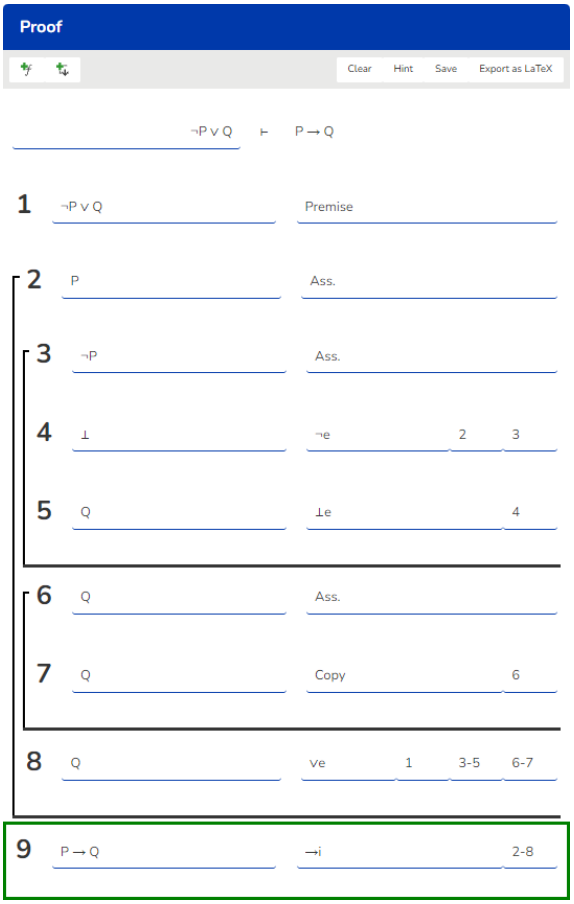


Figure 3.16: Example of a proof in Logan

adding the corresponding number of small boxes, where the user can write down the lines which are needed to make the rule valid. The assistant is constantly checking if what is written in the rule box is valid, and will turn red when something isn't in addition to giving feedback describing the error and, when possible, what's expected to be written. Just like Fitch, the assistant will point out which lines of the proof are causing it to be incorrect. When the conclusion is reached and all lines are correct, the box for the line containing the conclusion will turn green. As fully complete proof in Logan is shown in Figure 3.16

Logan is very complete as far as proof assistants go. Unfortunately, it is only available in a Fitch-style format, having no option to be displayed as a Gentzen tree. It also isn't fully capable of creating exercises. While it does allow a user to export the proof in the LaTeXformat, there is no way to import proofs.

3.2 Overview

This section is an overview and comparison of the logic teaching tools that have been analyzed and described. Since the goal of this thesis is to create a proof assistant tool for educational purposes, it is critical that what is already available is examined, whether it

meets the requirements wanted for our tool, and what we can take from it to utilize in it.

First, the format of proofs is an important matter. Most of the listed proof assistants follow a Fitch-style format. Yoda, Holbert, Panda, and Logitext, however, adhere to a Gentzen tree-style format which matches the format that we intend to utilize in our tool, as previously explained in Chapter 1. Of these, the last three are all open-source projects, which allows for the opportunity to study how they construct their proofs in the programming language chosen by their developers.

Another important aspect of the proof system is how the assistants verify and reason the logic of their proofs. Most of them appear to have their logical reasoning built with no assistance from external reasoners. However, exceptions such as Edukera and Logitext, both of which utilize Coq as an external reasoner, do exist.

The possibility of creating exercises is another important aspect of consider. How one judges what exercise creation consists of will differ from person to person, as one could simply assume that if a tool allows the input of premises and the conclusion of a proof, that is enough to do the job. If we follow these criteria, then every assistant previously listed except for Edukera can fulfill it. But, in educational terms, an exercise doesn't always consist in fully solving a proof from just its premises and conclusion. For example, a possible beginner's exercise could be one where the student is presented with a fully built proof, except that it doesn't have some of its proof rules written, and the student must correctly write them. Another example could be a partially constructed proof that the student must complete. Following the criteria needed for these types of exercises, tools such as Logitext won't be able to fulfill them. The tool must allow the partial construction of proofs and then be capable of saving that state, so it can be reloaded later for the students in the same way it was before. Tools such as Fitch, which can save its proofs as files and later reload them in the same state can fulfill this criterion.

Some extra criteria would be welcomed, such as data collection and integration with online platforms, however, none of the proof assistants present here seem to be built around them, and as of their current state, they won't be allowed to provide them. In this case, despite not providing a proof assistant, we have to mention the Iltis MOOC, which features anonymous data collection for purposes similar to what is desired.

Lastly, feedback must be considered. Feedback is important for an online teaching platform, as it exists to assist students in place of qualified teachers. While many tools are built for educational purposes, the availability of teachers or tutors can vary in online courses, and students may also want to test and verify their knowledge without relying on other humans. Many proof assistants were designed with the idea of simply verifying proofs that were previously written by hand, and if a student becomes stuck on the proof, they should seek guidance from their teachers. These tools will then see feedback as mostly trivial or completely unneeded. From the tools listed, Logan is the only one that continuously provides feedback during the construction of the proof, although Panda and Fitch also provide a good amount of feedback. It should also be noted that, of these three, only Logan is currently available online. For the remainder of the assistants, the

feedback is mostly minimal. Once again, a mention must be given to the Iltis MOOC, which provides a great example of how to provide feedback for exercises of other concepts of logic, as well as allowing the feedback given and received to be dosed in the quantity intended by both the teacher and student.

4

System Architecture and Implementation

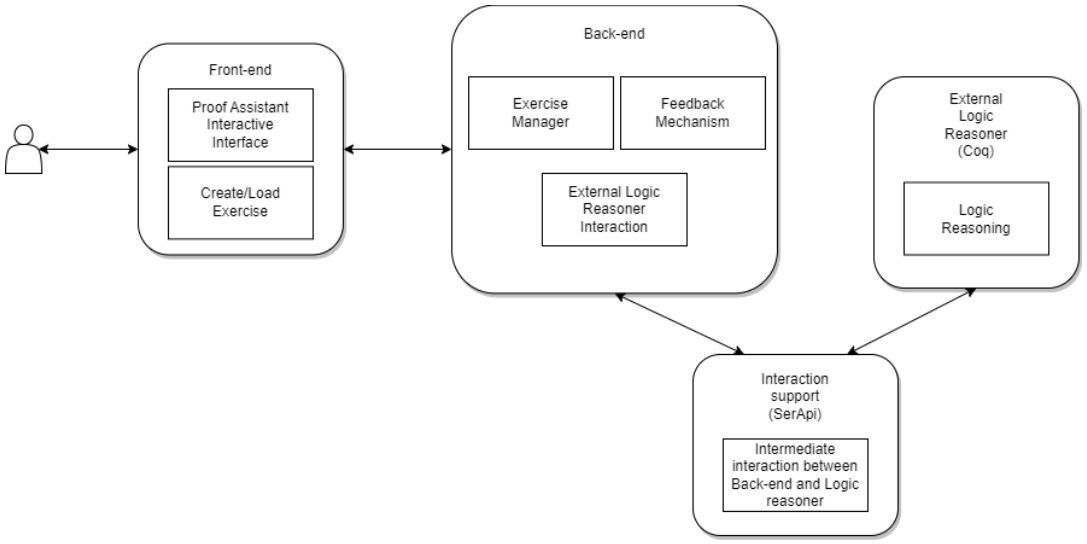


Figure 4.1: Old diagram of the system with an external reasoner.

This chapter will focus on explaining the architecture and implementation of the system as well as show a demonstration of it. Originally, as seen in Figure 4.1, the system was supposed to feature logical reasoning done via external sources, namely Coq, utilizing another framework that would act as a middle-man in the communication between the back-end and Coq, translating the actions performed by the back-end to the language of Coq and then vice-versa for the given response. This was briefly explored, where it was understood that the back-end would need RESTful services so that it could communicate with the middle-man framework. However, during this time the middle-man framework was deprecated in favor of another one which functioned as a language server protocol. Besides the added challenges of understanding this new framework, a few days were also spent trying to connect the framework's server to a client provided by its developers, which ended up in a failure, with not a single one of these many attempts managing to establish a connection between the two. These failures caused a fear that

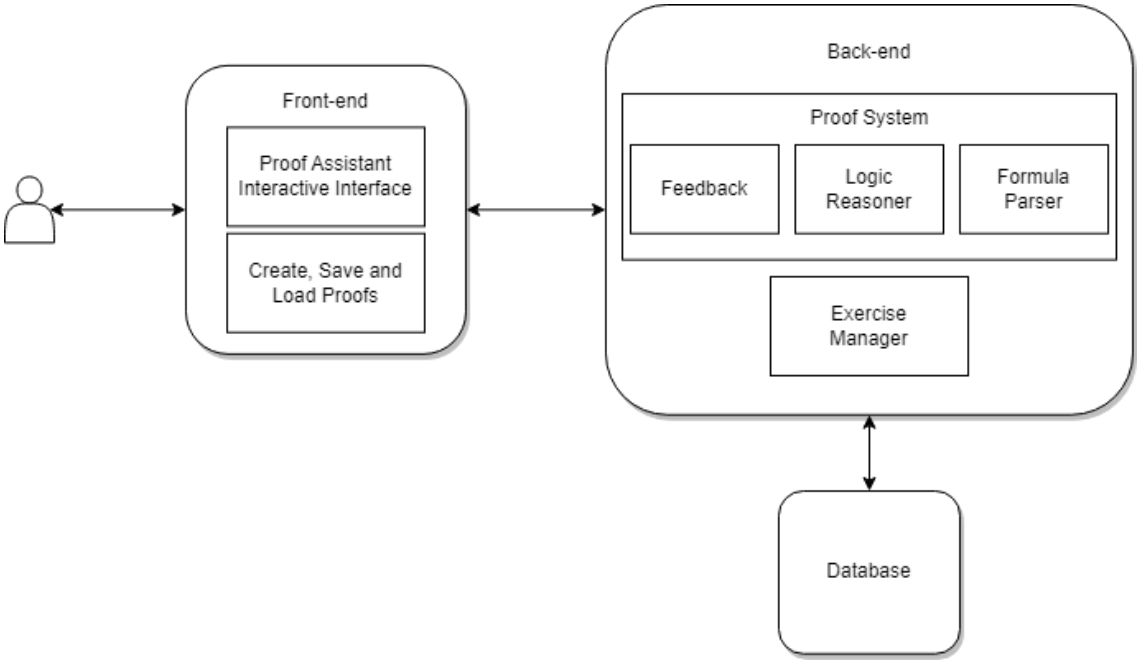


Figure 4.2: The new and final diagram for the system, featuring a local logic reasoner.

our tool would also be unable to connect with the server, and as such a choice was made to instead build the logic reasoning from scratch. Thus, the architecture of the system was changed to the one shown in Figure 4.2. In the new diagram we have two main components, the front-end and the back-end, as well as a database where proofs are stored. The back-end is responsible for taking care of the processing of proof systems, featuring three subcomponents, the formula parser, which is a component responsible for parsing strings into corresponding formula objects, a logic reasoner component that is responsible for verifying if the reasoning behind the application of a rule is valid, and a feedback component which is in charge of providing error messages that serve to help a user understand the error, without revealing the solution or correct step itself. The proof system component itself takes care of the processing of actions that a user can do to a proof, such as applying a proof rule or resetting the proof. The back-end also has an exercise manager component, which is responsible for saving proofs into files with a .json format, and loading those same files back into proofs. The front-end component is responsible for hosting the visual interface of the proof system, in which the users can interact with proofs and attempt to solve them, alongside some other features. It communicates with the back-end in order to process the actions done by the user in the interface of the proof.

This chapter will first feature a section detailing the back-end and its various components and sub-components. The back-end's chosen programming language is Java due to familiarity with the language as well as its capabilities to set up RESTful APIs utilizing a framework such as Spring [34]. This section will also start by explaining the Java object created to represent formulas of both propositional and predicate logic.

At the end, the chapter features a section about the front-end, which will focus on the

different web pages of the system, complete with various images and a full tech demo of the proof assistant, showing the various steps for the resolution of proof and other features. The front-end was built utilizing the popular React [33] framework in combination with the typescript language and various React packages to help in aspects such as maintaining the state of a proof and displaying feedback to the user.

4.1 Back-End

4.1.1 Formula Object

The first step was to create a representation of a logic formula. Since the programming language of choice is Java, this representation will be an object containing the necessary variables, flags and methods to fully represent and manipulate any kind of propositional or predicate logic formulas.

4.1.1.1 Propositional Logic

As previously stated in Chapter 2, all propositional formulas are composed of atomic propositions and a number greater or equal to 0 of connectives. Formulas can be distinguished between those with no non-negation connectives and those with non-negation connectives, utilizing an idea of arity. For this tool, a formula with none or only negation connectives is called unary. Formulas with a non negation connective have formulas to the left and right of it and are called binary. By allowing the left and/or right formulas of a binary formula to be either unary or other binary formulas, it is possible to represent every propositional formula that is not negated. For example, $(P \wedge Q) \wedge R$, is a binary formula with right side being the unary formula R and left side being the binary formula $P \wedge Q$. To represent the connectives, binary formula objects store an enum that can take the form of any propositional connective. This enum also helps with the visual representation of the formula, as each connective has an attached label with the specific Unicode character.

While this can take care of non-negated propositional formulas, a way to deal with negation is still necessary. Since formulas can have several negation connectives, it can be dealt with by storing an integer flag in the objects which has a value equal to the number of negation connectives of the formula.

With both arity representations of a formula in addition to the negation flag, the tool can now represent a propositional formula in the form of a formula object. These formula objects will then be manipulated in various ways, such as applying proof rules to them. To support this, some functionalities were added to the formula objects such as a way to create a string representation of the formula and a way to compare two formulas with each other to check if they are the same. The first one resulted in a method that converts the stored variables and flags into a string. The second was done by overriding the default equalsTo java method to instead compare formulas by checking their arity and if the atomic propositions, connectives and negation flags are the same. This ensures that two

different instances of a formula object that represent the same formula, for example both unary formulas with an atomic proposition P, return true when compared to each other, despite being different objects.

4.1.1.2 Predicate Logic

Predicate logic formulas have additional symbols, as shown in Chapter 2. Processing these extra symbols evokes the need for some more variables and methods than propositional formulas, while still keeping everything that those have, with a few changes. As such, these are governed by a different object which extends the functionality of the propositional objects which will be referred to as FOLFormulas.

The first symbols to take care of are quantifiers. First, quantifiers are applied to logical variables. Second, a formula is negated when there are negation quantifiers behind its quantifier, such as ¬∀P(x).

As such, a possible representation of a quantifier object can be thought of as a combination of the quantifier itself, the logical variable and a negation flag, with this negation flag being taken into consideration when looking at the whole formula affected by the quantifier. The quantifiers themselves are represented by an enum object, similar to how connectives are represented. Since formulas can have a number greater or equal than 0 of quantifiers at once, each FOLFormula stores a list of quantifier objects.

The remaining two symbols are constants and logical variables, both typically represented by lowercase characters. These are only present in unary predicate formulas, with the number of these symbols per formula determining the arity of a predicate logic formula, not to be confused with the arity used to determine the formula objects. To represent these, each unary FOLFormula stores a list of pairs, where each pair consists of the character and a boolean which determines if the character is a logical variable (true) or a constant (false).

To accommodate these additions, changes were made to the formula comparison method, such as checking if the quantifiers of both formulas are the same, in the same order and with the same variable, checking the constants and logical variables of each formula to verify if they are the same and if the order matches, and also checking the negation flag of the quantifier object.

4.1.1.3 Additional necessities

While the variables, flags and methods described above allow the representation of any propositional or predicate formulas, these aren't the only ones present in formula objects. As a proof assistant, the tool features proof systems and the manipulation of these, which are explained later in the chapter. These proof systems utilize these formula objects to represent all of the formulas within it, from the goal to premises and others. To aid in the manipulation of proofs, these objects store additional information, which helps

in correctly applying the actions available in the system, which are described in a later section of this chapter.

```
private List<IFormula> child_formulas;
private UUID father_formula;
private UUID id;
private boolean mark_status;
private int markNumber;
private RuleAssumptionPair rule;
private int formula_type;
protected int negated;
private Map<UUID, String> assumptions;
```

Figure 4.3: The variables held by every single IFormula object.

The additional informations stored by formula objects, shown in Figure 4.3, are:

- An Id. A Java UUID object that identifies the formula object.
- The formula type, an integer flag that details if a formula is a premise (1), an assumption (2) or if it was inferred from a proof rule (3). If unknown, it is set to 0.
- The id of its father formula. As the proof system used is of the Gentzen Tree style, the father of a formula is the formula directly below it. The goal of a proof is the only formula without a father in this context.
- A list of its children's formula objects. Empty if formula type is not inferred.
- A pair of Rule enum and a list of UUIDs. Empty if formula type is not inferred. This indicates which rule was applied to this formula, and which assumptions it generated.
- The number of its mark. -1 if formula type is not 1 or 2.
- The status of its mark, which indicates if it is open or closed. Set to -1 if the formula type is not 1 or 2.
- A map of the assumptions in its domain, with the key being its UUID and the value being a String representation of the formula.

4.1.2 Formula Parser

With the formula objects established, it was now necessary to be able to parse a string containing a formula into the corresponding formula object. While there are some Java libraries featuring string-to-formula object parsers, there were some issues with those, such as lacking support for predicate logic, or the formula object not having the capability to hold necessary information. Due to this, a custom parser was written, the FormulaParser object.

At its core, the FormulaParser is a simple object. It receives a string, removes any whitespaces from it and then processes each character, like a token. However, doing the process this way comes with various checks that need to be performed, which adds complexity. Additionally, due to the differences between propositional and predicate formulas, there are additional steps in the process of the latter, as well as some changes to the parts held in common with propositional formulas.

4.1.2.1 Propositional Formula

```
function parseConnector(Connector c, String formula, int negationValue) throws InvalidFormulaException

    boolean noShift Set to false //temporary flag that checks if currentFormula is adjusted at the end.
    if previous token was a Connector
        then throw InvalidFormulaException

    IFormula newFormula

    if currentFormula is null
       then Set newFormula to (create new UnaryFormula with (termName, negationCount)) //termName and negationCount are global variables.

    else
        IFormula rhs Set to (create new UnaryFormula with (termName, negationCount))
        IFormula rightFormula Set to currentFormula.getRight()

        if function was called after the last right parenthesis
            then Set postFinalRB to false
        end if

        if currentFormula will be inside newFormula is false
            then if rightFormula is null
                then Set currentFormula.SetRight to rhs
            end if
            Set newFormula to currentFormula
        else if newFormula will be inside currentFormula and rightFormula is not null
            then if rightFormula is an instance of IBinaryFormula and rightFormula.getRight() is null
                Set newFormula to rightFormula
                then if previous token was equal to ')' then
                        Set newFormula.SetRight to rhs
                    else
                       Set newFormula.SetRight to (create new BinaryFormula with (rhs, null, c, negationValue))
                       Set noShift to true
                    end if
                end if
            end if
        else
            Set newFormula to rhs
        end if
    end if

    if noShift is false then
        BinaryFormula aux
        Set aux to (create new BinaryFormula with (newFormula, null, c, negationValue))
        if newFormula is inside currentFormula then
            Set currentFormula to aux
        else
            Set currentFormula.SetRight to aux
        end if
    end if

    call adjustVariables with (formula, c)
end function
```

Figure 4.4: Pseudo-code for the parsing of a connective in a propositional logic formula.

Starting with propositional formulas, each token is processed by first checking if it is a parenthesis (left or right) or a connective ($\wedge$, $\vee$, $\leftrightarrow$, $\rightarrow$, $\neg$). If it isn't any of these, the parser assumes that the token is part of a term, and stores it for later. If it is any connective other than negation, the parser starts building a binary formula with that connective and the term it has stored. This has a couple of different paths that can be taken, described by the pseudo-code in Figure 4.4. The parser contains various flags within it, which can be toggled on and off during the processing. The path chosen by the parser depends on the status of these, as well as if there is already a formula in the process of creation. For the first time the parser processes a connective, it will create a binary formula with its

left side and the connective, with the right side initially staying empty, to be filled later. From then on, further connective processing will need to check flags and act accordingly. For example, if the inside flag is toggled on, the parser knows that the binary formula currently being created will be placed inside of the formula that was created in a previous connective processing, and will be stored on its right side.

Once processing of the connective is completed, several flags and variables have their values adjusted, such as resetting the stored term to an empty string. Additionally, this step also checks to see if the next token is a left parenthesis or a negation connective followed by a left parenthesis and, if so, toggles the inside flag to true, as it means that the upcoming formula is inside of the binary formula that was just created in the previous step.

When creating the binary formula during the processing of a connective, the parser also assigns a value to its negation flag. This value is obtained from the processing of negation connectives. When encountering a negation connective, the parser will first increase a counter by 1, then check the next token for parenthesis or another negation connective followed by a parenthesis. In case this happens, the negation values will be applied to whatever is inside the parentheses. Since the processing of a formula's negation value is done during the processing of a connective (if there are any), the value of the negation counter is stored, with additional information so that the flag is applied to the correct formula, as we can have various negation connectives belonging to different formulas before a connective is reached.

Finally, the parser checks for left and right parentheses. When processing a left parenthesis, a couple of values are updated, such as:

- Incrementing the left parenthesis counter.
- Incrementing the list of negation counters by one.
- Resetting the local negation counter.

Processing the right parentheses is more complicated. This is because a right parenthesis that isn't following up another right parenthesis signals the end of a formula. As mentioned prior during the processing of a connective, a binary formula with an empty right side is always formed, and that right side needs to be filled eventually. As such, one of the steps for the processing of right brackets is to check for the presence of the latest binary formula previously created by the processing of a connective and fill up its right side with the stored term. In addition to this, there are also some flag verifications, like checking if the number of left and right parenthesis are imbalanced or adjusting the inside flag depending on the remaining number of parentheses.

Once every token of the formula has been processed, a couple more things are verified and adjusted. First, it is checked if the left parentheses counter is higher than 0, throwing an exception if so as this means there were more left parentheses than right parentheses.

Second, the parser checks for the existence of a binary formula. If there aren't any, it means the formula didn't have connectives, being an unary formula. The parser then creates this unary formula with the stored term and returns it. Finally, if there is a binary formula, there's a check to see if its right side is empty and, if so, an unary formula is created with the stored term and placed on the right side. Finally, the entire formula is returned.

4.1.2.2 Predicate Formula

```
function parseConnectorFirstOrder(Connector c, String formula, int negationValue) throws InvalidFormulaException
    boolean noShift Set to false
    if previous token was a Connector
        then throw InvalidFormulaException

    IFormula newFormula
    if currentFormula is null
       then Set newFormula to (create new FOLUnaryFormula(quantifiers.removeLast(), negationCount))
    else
        IFormula rhs Set to (create new FOLUnaryFormula(quantifiers.removeLast(), negationCount))
        IFormula rightFormula Set to currentFormula.getRight()

        if function was called after the last right parenthesis
            then Set postFinalRB to false
        end if

        if currentFormula will be inside newFormula is false
            then if rightFormula is null
                then Set currentFormula.SetRight to rhs
            end if
            Set newFormula to currentFormula
       else if newFormula will be inside currentFormula and rightFormula is not null
            then if rightFormula is an instance of FOLBinaryFormula and rightFormula.getRight() is null
                Set newFormula to rightFormula
                    if previous token was equal to ')'
                        then Set newFormula.SetRight to rhs
                    else
                        List<Quantifier> quants
                        while quantifiers.size() > 0 do
                            Add all members of quantifiers.removeLast() to quants
                        end while
                        Set newFormula.SetRight to (create new FOLBinaryFormula with (quants, rhs, null, c, negated))
                        Set noShift to true
                    end if
                end if
            end if
        else
            Set newFormula to rhs
        end if
    end if

    if noShift is false then
        BinaryFormula aux

        List<Quantifier> quants
        while quantifiers.size() > 0 do
            Add all members of quantifiers.removeLast() to quants
        end while

        FOLBinaryFormula aux
        Set aux to (create new FOLBinaryFormula with (quants, newFormula, null, c, negated))

        if newFormula is inside currentFormula then
            Set currentFormula to aux
        else
            Set currentFormula.SetRight to aux
        end if
    end if

    call adjustVariables with (formula, c)
end function
```

Figure 4.5: Pseudo-code for the parsing of a connective in a predicate logic formula.

For predicate formulas, some additional symbols need to be checked, such as quantifiers, constants and variables. Additionally, the processing done for connectives and parenthesis needs some adjustments to deal with those symbols. When creating a proof,

there must be an indication that the proof is of the first-order logic branch. This indication comes in the form of a boolean flag, and the parser will act based on its value, with the proof being considered of first-order if the flag is true.

The first issue that occurs with the parsing of predicate formulas is how to deal with constants and logical variables. Since these are typically lowercase characters that occur after a term, it is hard for the parser to distinguish when it is looking at one of these or a character of an atomic predicate. As such, a constraint was forced, which is that variables and constants must be within parentheses, for example, an atomic predicate P with a constant a must be represented as P(a). As we know these symbols appear after an atomic predicate, a check is now made during the processing of a character that checks if the next character is a parenthesis. If so, another flag is then turned on, and further characters will now be treated as variables or constants, until the flag is turned off again.

When the parser finds a quantifier ($\exists$ or $\forall$), it checks the character that follows it and checks if it is a lowercase character. If so, then the quantifier is added to a list of lists of quantifier objects (quantifier, logical variable and negation value), the logical variable is added to a list, which makes it easier to check if a variable is already in use, and the negation counter is reset. The reason for multiple lists of quantifiers is that there can be multiple sets of quantifiers belonging to different formulas before coming across the processing of one of these formulas. As such it is necessary that each quantifier list is stored, to be used when the processing of its corresponding formula happens.

When processing logical variables or constants, these are added to a list of pairs of a char and a boolean flag. If the character is a logical variable (as in, is contained inside of the previously mentioned variable list) then it is added to the pair list with the flag set to true, indicating that it is a logical variable. Otherwise, it is added with the flag set to false, indicating it is a constant.

The processing of connectives is adjusted to also deal with quantifiers, as can be seen in the pseudo-code in Figure 4.5. Now the parser creates FOLFormulas instead of Formulas, and these require a list of quantifiers to be created (even if the list is empty). Due to the order that the parsing takes, the list of quantifiers is obtained by removing the last member of the list of lists of quantifiers. If the created FOLFormula is unary, then it is also given the variables and constants list. One thing to note here is that every list of quantifiers from the list of lists is used in the creation of the binary formula, being left empty after the processing of the connector.

Following the processing of the connective there is an adjustment of flags and variables just like with propositional formulas, with the additions of clearing the list of logical variables and constants, as well as the addition of a new, empty list of quantifiers to the list of lists.

There are also some additions to the processing of parentheses when compared to propositional formulas. For the left parenthesis, an empty list is added to the list of lists of quantifiers. For the right parenthesis, FOLformulas are created instead of regular formulas and a check is performed to see if the flag regarding the processing of variables

and constants is turned on. If so, it is turned off, as it signals the end of that formula's variables and/or constants.

Finally, the processing of negation is unchanged from propositional formulas.

4.1.2.3 Error catching / Feedback

As was mentioned earlier, the parser is always checking for various errors that the provided formula might have. Upon catching any error, the execution of the parser is halted by an exception throw, which contains an error message detailing the specific error that occurred, which is later used for feedback in the front-end part of the application. While the given formula can have many different errors, this method will only inform the user of the first error it finds, as execution is halted whenever an error is found.

The possible errors that can be found are:

- More left/right parenthesis than right/left.
- One or more negation connectives without a formula, for example ¬∧Q.
- If the variable/constant processing flag is turned off, the presence of a left parenthesis without a connective right **before** it.
- If the variable/constant processing flag is turned off, the presence of a right parenthesis without a connective right **after** it.
- Empty formula.
- Two or more connectives in a row.
- No variable after a quantifier.
- Predicate formula's atomic predicate without variables or constants.

4.1.3 Logic reasoner

As the idea of using an external logic reasoner was discarded, it was necessary to build one from scratch. The result of that is the Logic Reasoner component, which consists in an object, named LogicReasoner, with several methods, one per proof rule, which checks if it is possible to apply the rule, by checking the various conditions. This section will explain how the reasoner works, first for propositional proof rules and then for predicate proof rules.

Due to how the system was built, the reasoner is only capable of backward reasoning, which allows users to completely build a proof tree from the bottom (goal) to the top. While this type of reasoning is intuitive regarding introduction rules, it is less so in the case of the other rules, such as predicate rules. This is because the proof is built from the bottom up, starting with the goal, which means in the case of other proof rules, they are

being called when that sub-goal is the result of its application on other formulas. The solution found for this was to ask the user to provide extra information, most commonly the formulas, as shown in table 4.1.

If the reasoner finds the reasoning to be incorrect at any point, it will stop its execution and throw an exception containing an error message, much like FormulaParser does.

4.1.3.1 Propositional Rules

All non-negation propositional proof rules share the same pre-conditions that must be fulfilled to be applied. Those are the formula to which the rule is being applied must not be negated, the formula must be binary and it must contain the specific connective for that rule. The formula chosen to verify the rule application is different based on whether the application needs a formula provided by the user. In those cases, the chosen formula is the one provided by the user. The remaining cases use the current sub-goal, which is the formula to which the rule is being applied.

There is a fundamental difference between introduction and elimination rules in this case, that being the formula that is taken into consideration when testing the validity of applying the rule. Introduction rules use the sub-goal to which it is being applied, while elimination rules test the formula provided by the user when asked.

Additionally, three of the elimination rules have an additional check:

- Conjunction Elimination and Equivalence Elimination also check to see if the sub-goal it is being applied to is on the correct side of the provided formula.
- Implies elimination checks if the formula of the sub-goal it is being applied to is on the right side of the provided formula.

Finally, the negation proof rules, Negation Introduction and Negation Elimination check if the sub-goal they are being applied to is negated or non-negated, respectively.

4.1.3.2 Predicate Rules

Predicate rules have some pre-conditions that affect all of them, such as the formula or quantifier not being negated and that the quantifier being be the same as the one specified in the rule. Just like with propositional rules, the formula tested depends on the rule, with the introduction rules testing the sub-goal formula and the elimination rules testing the formula provided by the user. Due to this, the formulas tested always have at least one quantifier, which is why the quantifier is always tested by the system.

However, as shown in chapter 2, individual predicate rules have some pre-conditions they must follow. Some of these pre-conditions relate to the assumptions that occur in the proof and are dealt with by the Proof System component instead, but some are still dealt with by this component:

- Universal Introduction verifies if the provided constant is present in itself or the premises of the proof.
- Existential Elimination verifies if the provided constant is present in itself, in the goal of the proof or in the premises of the proof.

If any of the above conditions are fulfilled, the rule application process is halted and an exception is thrown.

Universal Elimination is a special case in the context of this system. As a predicate rule, it has the same pre-conditions as Existential Introduction, however, due to the system only supporting backward reasoning, the application of this rule cannot consist in a simple substitution of all constants of the provided kind to the provided variable, as a substitution in formulas with n-ary predicates, such as P(aa) where P is a predicate and a is an arbitrary constant, can be applied to less than n constants. As such the system's application of this rule asks the user to provide the entire formula instead, and the role of the reasoner with this rule is verifying if it matches a correct substitution of one kind of constant to the variable present in the Universal quantifier. This consists in checking the following:

1. If the formulas have the same arity.
2. If the provided formula has one more, universal, quantifier than the current sub-goal, as well as having the rest being the same.
3. If the constant to variable substitution that was performed is valid.

4.1.4 Proof system

The proof system component is responsible for maintaining and manipulating the state of the proof. It consists of a Proof System object, created by providing a formula object for the goal, and an array of formula objects for the premises. Each proof system object represents a single proof. The object contains several methods, allowing the user to perform actions such as resetting a proof, applying a proof rule to the current sub-goal, labeling a formula as a premise or an assumption and deleting an application of a proof rule. This object also contains references to the LogicReasoner and FormulaParser objects.

To aid in the manipulation of the proof, certain variables are stored, starting with the provided goal and premises formula objects. The sub-goals of the proof are also stored inside a list, which is constantly updated following the actions performed by the users. To help with managing sub-goals, there is also an idea of a current sub-goal, a concept inspired by proof assistants such as Coq, in which the system locks onto a sub-goal, and every action done by the user is applied to it. The current sub-goal is stored as a formula object, and will always be one of the formulas inside of the sub-goals list. The user can change the current sub-goal to another one on the list using one of the available actions. It will also be switched automatically and removed from the list of sub-goals once certain actions are applied to it, such as applying a proof rule.

To further aid the manipulation of the proof, the object also stores a list of every assumption in the proof, independent of its domain, as well as a map of every formula object in the proof system, stored with its id as the key, and an integer counter for mark numbers.

This chapter will further explain the actions available to the user, including the application of proof rules to the current sub-goal. After, there will be a section detailing this component's contribution towards providing feedback to the user.

4.1.4.1 Actions

These are the actions available to the user that are not proof rules.

Switch Sub-Goal

This action switches the current sub-goal to the formula from the list of sub-goals with the corresponding id to the one provided by the user. This action is also responsible for checking if the proof has been finished. Once the sub-goals list is empty, a check is performed on every formula of the list of assumptions to check if they've been closed. If an assumption is open, the system checks if it ever appears on the proof. Otherwise, the proof is considered finished.

Reset Proof

Resets the proof to its original state. For a new proof, this is as if starting the proof from zero, however for a proof loaded by the Exercise Manager it is reset to the initial state saved on the proof file.

Delete Rule

Deleting a rule implies deleting the branch created by that rule application. This method deletes the children from the formula that the rule was applied to, and changes its formula type and mark number to their default values. Figure 4.6 shows the pseudo-code for this action.

Before deleting the children, their formula type is checked. If they are type 2 (assumption), the corresponding assumption in the list of assumptions from the ProofSystem object is updated to its default values (such as its markStatus being open again) and the mark number counter is decreased by 1. If the formula type is 3 (inferred), the system checks its rule and assumptions pair and removes them from its list of assumptions. Once this check is done, the formula is deleted from the map of formulas as well as the sub-goals list, if the formula was a sub-goal.

Once the deletion of the child formulas is complete, the formula's rule and assumptions pair is deleted, and the formula is added to the sub-goals list and the current sub-goal is switched to the first element of the list.

Label a formula as a premise (isPremise)

This action labels a formula as a premise if it matches one of the formulas inside the system's premises set. If this is the case, then the current sub-goals formula object has its flags updated, specifically the formula type becomes 1 and it is given a mark. Afterward,

```
function deleteAssumptions(IFormula toDelete)

    if toDelete is the goal of the proof then
        Clear assumptions list
        Clear proof formulas list
        Clear subgoals list
        Set mark number counter to premises.size() + 1
        Add toDelete to map of proof formulas
    end if

    List<IFormula> arr
    Add toDelete to arr

    while arr is not empty do
        IFormula currentFormula Set to arr.remove(0)
        if currentFormula's formula type is 3 then
            Add every child formula from currentFormula into arr
            for every formula's id in the list of currentFormula rule pair //
                Delete corresponding assumption formula from assumptions list
            end for
        else if currentFormula's formula type is 2 then
            for every formula in the list of assumptions
                if (formula is equal to currentFormula)
                    Set formula's mark number to -1
                    Decrement mark number counter
                    Break for loop
                end if
            end for
        end if
        Remove currentFormula from map of proof formulas
        Remove currentFormula from list of sub-goals, if present
    end while

    Add toDelete to proof formulas map

end function
```

Figure 4.6: Pseudo-code for the deletion of a rule.

the current sub-goal is removed from the list of sub-goals and a new current sub-goal is set.

Label a formula as an assumption (isAssumption)

This action labels a formula as an assumption, in a similar but slightly more complex way as isPremise. The current sub-goals isAssumption method is called, which checks its own assumptions map to see if it is contained inside. If this is the case, the formula type is updated to 2, the mark status flag is changed to false and a mark number is attributed, once again the value of the mark number counter, which is passed to this method as a parameter. The method then returns the id of the assumption, which is used to find the correct assumption in the system's map of assumptions and update it accordingly. Afterward, the current sub-goal is removed from the list of sub-goals and a new current sub-goal is set.

4.1.4.2 Proof Rules

Proof rules are applied to the current sub-goal and cause some changes in the proof. Each proof rule is considered as an action, as the system only supports backward reasoning. If forward reasoning was available, there would be two actions per rule. Due to this, as

previously mentioned in the LogicReasoner section 4.1.3, certain rules request additional information or context from the user so that the rule can be correctly tested and applied.

For every application of a proof rule, there is a set of steps that happen, which are:

1. Testing the logical reasoning of applying the action. This step is either applied to the current sub-goal or the formula provided by the user, depending on the rule.
2. Generating the formulas that will be both the children of the current sub-goal and the next sub-goals.
3. Generating the assumptions for the rules that eliminate marks, such as Introduction of Implies.
4. Processing the generated children, labeling the current sub-goal as inferred, removing it from the list of sub-goals and switching it with another one.

The first step involves calling the specific rule's method on the FormulaReasoner object and sending the necessary parameters such as the formula that is being tested. If the reasoning fails, an exception will be thrown, which halts the action.

The second and third steps involve creating new formula objects in correspondence to the specific rule. For example, the rule Introduction of Implies will generate two new formula objects with the same formulas as their left and right side, by parsing the string representation of those formula objects. This will be shown in the next section regarding the front-end. The new right-side object represents the next sub-goal and will be processed during step 4. The new left-side object represents a generated assumption and, if the conditions are right, is added to the system's list of assumptions, as well as the current sub-goals map of assumptions. It should be noted that assumptions act on a certain domain of formulas, and that is replicated by adding them only to the assumption maps of the formulas within the correct domain. The formulas generated by steps 2 and 3 can be seen in Table 4.1.

```
function processArityTwo(IFormula firstFormula, IFormula secondFormula, RuleAssumptionPair rule)

    Set firstFormula's father formula to current sub-goal
    Set secondFormula's father formula to current sub-goal

    Set current-subgoal's formula type to 3
    Add firstFormula and secondFormula to current sub-goal's list of children formulas
    Add current sub-goals map of assumption's elements to the firstFormula and secondFormula assumptions map
    Set current sub-goals's RuleAssumptionPair to rule.

    Add firstFormula to list of sub-goals
    Add secondFormula to list of sub-goals.

    Add firstFormula to map of proof formulas
    Add secondFormula to map of proof formulas

    Delete current sub-goal from list of sub-goals
    Switch current sub-goal to first element of the sub-goals list

end function
```

Figure 4.7: Pseudo-code for the processing of a rule application that generated two sub-goals.

The fourth step involves processing the child formula objects. Figure 4.7 features the pseudo-code for this step in a case where two children were generated. In essence, the children's and father's formula objects are connected and the children are added to the list of all formulas and the list of sub-goals.

These steps happen in all rule applications, but some rules, notably the predicate proof rules, may have extra steps, such as extra reasoning and substitution of variables and constants. The steps are:

- Variable for constant substitution: Introduction of Universal, Introduction of Existential.
- Constant for variable substitution: Elimination of Existential.
- Check assumptions map for open assumptions containing the provided constant: Introduction of Universal, Elimination of Existential.

The table 4.1 contains information for all rules regarding which information is necessary to be provided by the user (User Info), which assumptions are generated and finally which child formulas are generated and placed on the sub-goals list. It should be noted that any formulas or constants shown in User Info are demonstration examples.

The implementation of the proof rules varied in difficulty. Propositional proof rules that do not generate assumptions are fairly straightforward, with the only difficulty being coming up with an appropriate system for elimination rules that worked with backward reasoning, which ended up being asking the user to provide the child formula with the corresponding connective. For the rules with assumptions, there is another degree of difficulty, as there was both a need to be able to delete the assumptions from the system whenever a rule is deleted, as well as maintaining a notion of domain, which differs from rule to rule. These requirements lead to the addition of the rule and assumptions pair to the formula objects and the domains listed in table 4.1, which are enforced with the formula object's map of assumptions.

Predicate proof rules also had to be adjusted to backward reasoning by asking the user to provide additional information, such as the constant to perform a substitution. Some also required additional reasoning to be added which was described above. Finally, methods were added to FOLFormula objects to help in the process of substitution, both for constant-to-variable and variable-to-constant substitutions.

4.1.4.3 Feedback

The ProofSystem object is also responsible for organizing and distributing feedback. In essence, every action in this component is surrounded by try-catch mechanisms, which work on catching several different types of exception, both thrown by the method itself or from the other components like the LogicReasoner and FormulaParser which have their methods invoked during the processing of these actions.

When an exception is caught during an action, it comes with a specific error message that details what error was committed and attempts to succinctly explain the error. These messages aren't very large or complex, as that could give away too much information to the user, which would go against the purpose of the feedback, which is to help the user and not solve the exercise for them. With an exception caught, the action is halted and rolled back, and the error message is returned instead. This allows for the front end of the system to receive this message and show it to the user. Should the action proceed smoothly, an OK message is returned instead, allowing the front-end to easily distinguish between a success and an error. The front-end section 4.2 contains example figures of how the front-end shows the user these messages.

4.1.5 Exercise Manager

The exercise manager is a component consisting of an Exercise Manager object that creates files via the serialization of a ProofSystem object into a .json file and the deserialization of those same .json files back into ProofSystem objects. This is done by utilizing the Jackson [23] library, a JSON parser built for Java. Both the process of creating/serializing and loading/deserializing a proof requires a filename to be provided by the user, as they work by writing and reading data Java file objects, which are created by providing a file path that includes the name of the file.

4.1.5.1 Serialization

The serialization done to the ProofSystem objects is rather simple. Jackson can serialize an object as long as there are getter methods for each of the variables that need to be saved. In some systems, it may not be possible to have these methods for some of the variables, for example, if the developer utilizing Jackson does not have access to the source code. This isn't the case here, however, and methods for every variable besides the FormulaParser and LogicReasoner objects exist. As to not confuse Jackson, the variables that are not meant to be saved are also annotated with @JsonIgnore, which will keep them hidden from Jackson for all purposes. With these methods, Jackson can obtain all the information about the object and parse it into a .json file, which is then stored in the path given when creating the File object. Serialization done this way has the benefit of maintaining the state of a proof, which allows for the creation of exercises with partially built proofs, and also allows for resets of these exercises to go back to that state, instead of a regular reset back to just the goal.

4.1.5.2 Deserialization

The deserialization process, however, is more complicated. In practice, one of the ways Jackson can use to deserialize an object is by converting the stored text back into the corresponding Java object and then attributing them to the specific variable via setter

methods. This way requires the presence of setter methods for all of the variables, which can cause problems in certain, more complex programs, but since most of these methods already existed, it was a simple matter of adding the few that were missing. Unfortunately, a problem arises with this method, in the form of object duplication. For ease of management, the reference for the same object can be present in multiple places, such as the children of formula also being present in the map of formulas in the system. Due to the way Java's object-oriented approach works, there is only one instance of an object, which is created in the heap memory, and all other places where the object "appears"are simply references that point towards the object in the heap. Jackson's deserialization method cannot always perceive that, and will instead create two different objects and reference each one separately. This is a major issue as it would break several of the methods of the ProofSystem object. For example, with a single object and various references, updating the object via one of the references will lead to every other reference also showing the updates, as they all point to a single object, but with two objects, the update done to one will not be reflected on the other, and with the references now either pointing to one or the other, some will not be correctly updated.

To fix this, Jackson allows the creation of a custom deserializer, which can then be affixed to the ObjectMapper object, which is the object Jackson uses to serialize and deserialize. This will replace the deserializer for that object with the custom deserializer. The custom deserializer was written to first deserialize the goal of the formula, and then remake the map of all formulas by going through the goal and its children, which make up all the formulas in the proof. By then assigning all other variables such as the list of sub-goals to the corresponding formula object from the proof formulas's map, there will only be a single object per formula, allowing for the various actions that can be applied to function correctly. The same thing must be done to the assumptions of the proof, to maintain the reference between the maps and lists.

4.1.6 Service and Controller

This system functions with an MVC architecture, with the front-end being responsible for the View, while the back-end is responsible for the Controller and Model/Service. The Service layer is responsible for processing all the business logic required to fulfill the requests made to the back-end, and makes use of a class that manages the back-end components that have been detailed earlier in this chapter to do so. The Controller layer is responsible for exposing the API of the back-end, receiving requests from the front-end to process an action and responding to the front-end. This layer is managed by a controller class that extends an API interface, containing the various actions that the back-end offers. To help with setting this up, the system makes use of the Spring [34] framework and Spring Boot [35]. These tools help launch the back-end online with the help of annotations that are placed on specific classes, such as the controller with @RestController, which makes Spring aware of their existence when it is time to expose the API's functionalities.

In the API interface, each action is mapped with an annotation that gives information about its URL and the type of HTML mapping, i.e. GET, POST, etc... Additionally, the API itself is annotated with a @RequestMapping annotation, which designates the URL for the whole API.

Whenever a request is made to the back-end, the controller will catch it and process it by calling the specific action on the service class with the given request body information as parameters. Upon being called, the service will check the contents of the request body and process the action utilizing its content. For requests that apply an action to an already created proof, the request body will come with the id of the proof so that the service can find the specific proof in the repository, allowing multiple users to utilize the back-end for different proofs at the same time. Upon finishing the processing of any action, the service will return a DTO with information relevant to whoever made the request, such as the id of the proof for newly created or loaded proofs, or the current state of the proof system in the case of proof rules. Figure 4.8 shows the ProofSystemDTO, a DTO that carries information of the proof that is useful to be displayed to users.

```
public record ProofSystemDTO(IFormulaDTO goal, String[] premises, IFormulaDTO currSubGoal, String[] subgoals)
```

Figure 4.8: A DTO representing a proof system object. As a Java record, each of the fields has an underlying setter and getter.

With both of these, the back-end's API can now be exposed online for the front-end to establish a connection and start making requests to manipulate proofs.

Formula	Rule	Provided User Info	Generated Assumptions	Generated Sub-Goals
$P \wedge Q$	Introduction of Conjunction			Left: P, Right: Q
P	Left Elimination of Conjunction	Formula: $Q \wedge P$		Provided formula: $Q \wedge P$
Q	Right Elimination of Conjunction	Formula: $Q \wedge P$		Provided formula: $Q \wedge P$
$P \vee Q$	Left Introduction of Disjunction			Right: Q
$P \vee Q$	Right Introduction of Disjunction			Left: P
P	Elimination of Disjunction	Formula: $Q \vee R$	Case 1: Q, Case 2: R	$Q \vee R$, 2x Formula: P
$P \rightarrow Q$	Introduction of Implies		P	Right: Q
P	Elimination of Implies	Formula : $Q \rightarrow P$		Provided formula: $Q \rightarrow P$, Left of provided formula: Q
$P \leftrightarrow Q$	Introduction of Equivalence		Left Domain: Q, Right Domain: P	Left: P, Right: Q
P	Left Elimination of Equivalence	Formula : $Q \leftrightarrow P$		Provided formula: $Q \leftrightarrow P$, Left of provided formula: Q
Q	Right Elimination of Equivalence	Formula : $Q \leftrightarrow P$		Provided formula: $Q \leftrightarrow P$, Right of provided formula: P
$\neg P$	Introduction of Negation	Formula: Q	P	Provided formula: Q, Negation of provided Formula: $\neg Q$
P	Elimination of Negation	Formula: Q	$\neg P$	Provided formula: Q, Negation of provided Formula: $\neg Q$
P(a)	Elimination of Universal	Formula: $\forall x$ P(x)		Provided Formula: $\forall$ P(x)
P(a)	Introduction of Existential	Constant: a		Substitution: $\exists x$ P(x)
$\forall x$ P(x)	Introduction of Universal	Constant: a		Substitution: P(a)
$\exists x$ P(x)	Elimination of Existential	Formula: $\exists x$ Q(x), Constant: a	Q(a)	Provided formula: $\exists x$ Q(x), Formula: $\exists x$ P(x)

Table 4.1: Table showing what is necessary for the system to process each proof rule, as well as what is generated by that processing.

4.2 Front-End

The front-end of the system is responsible for the graphical representation of the proof assistant and was built utilizing the React [33] framework in conjunction with the Typescript language. It consists of two web pages, a home page and a proof page.

The home page is where the user starts a proof. This can be done either by providing a goal and premises or loading a previously saved proof file. Once the proof is inputted, the user will be redirected to the proof page via the use of the React-Router [31] package.

The proof page is where the user can manipulate the proof by using the actions previously shown in the Proof System section, as well as various other information about the proof, such as its goal, premises and assumptions, which are all displayed on the sides of the screen. This page also features a 'Save Proof' button which allows the user to save the proof into a file.

All of the actions available on the web pages displayed by the front-end are processed by the back-end system. When the user chooses an action in the front-end, a request will be sent to the back-end, where the action is processed, followed by the back-end sending a response back to the front-end. These responses include information that is stored by the front-end in a store from the React-Redux [30] package, allowing the front-end to maintain a state for the proof.

The following sections will explain both web pages in further detail, including a tech demo of a proof resolution in the proof web page, complete with images detailing most steps.

4.2.1 Home / Start Proof Page

When entering the home page of the system, the user will see a box with two text fields, a first-order switch and two buttons, as shown in Figure 4.9. This initial page is where the user can either start a new proof or load a previously created one.

4.2.1.1 Creating a new proof

To start a new proof, the user is required to enter a formula into the goal text field, as well as write every premises in the premise text field, which is a multi-field text field from the materialUI [27] package, one per line. Additionally, the user must flick the first-order switch if the proof is of that branch. Since many proofs do not need premises, that box can be left empty, but it is required for the user to provide a non-empty formula for the goal to be able to start the proof, otherwise the system will show an error message to the user, as seen in Figure 4.10. These error messages, as well as every other message shown by the front-end webpages, are generated by the back-end and displayed with the help of the React-Toastify [32] package. To write logic formulas the user needs to be able to type the characters for connectives and quantifiers. These are Unicode characters which are not present in any regular keyboard. To deal with this situation, the system will display a

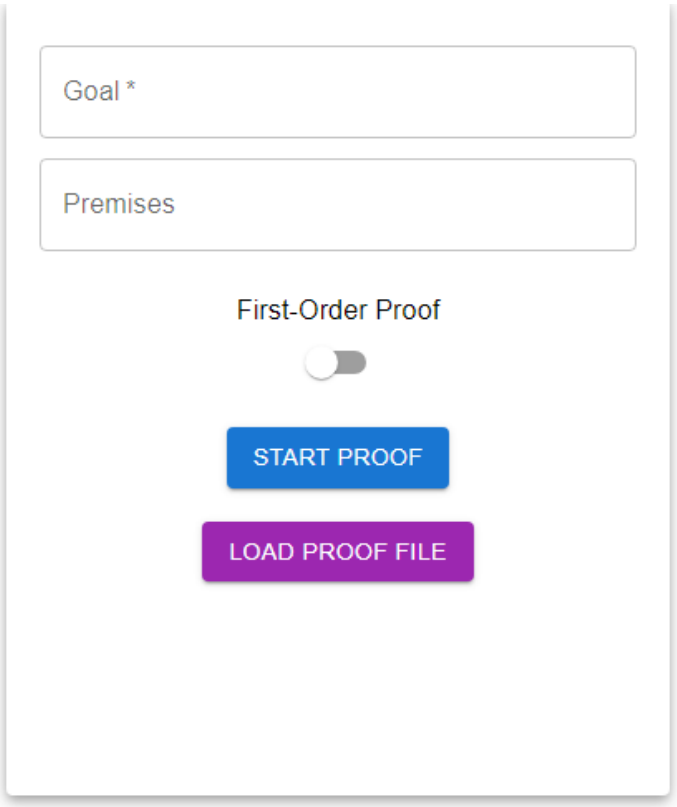


Figure 4.9: The box shown to the user on the home page.

set of buttons, each one corresponding to a connective or a quantifier, when a user clicks either one of the text fields, as shown in Figure 4.11. When clicking on any of the buttons, it will type the corresponding Unicode character into the text field directly above them and is capable of typing the character directly where the cursor was left before the click. When everything has been done, the user can proceed to the proof page by clicking the Start Proof button. Figure 4.12 shows the box with the goal and premise of the proof which will be resolved further down in the tech demo section.

Upon clicking the Start Proof button, the front-end will launch a request containing a CreateProofDTO which consists of a string representing the goal and an array of strings representing the premises. The code for the request is generated automatically using the openapigen [29] tool by providing the API specification of the back-end in a .yaml format. The generator will then create a request method for each of the back-end endpoints. Upon receiving this request, the back-end will process it and attempt to create a new Proof System object with the contents of the provided request body, in this case, the CreateProofDTO. On success, the back-end will send respond by sending a StartProofDTO, which contains the id of the proof, the necessary data (goal, premises, sub-goals) about the ProofSystem to display it and an OK message, which are all stored in a React-Redux store. If, however, the processing of the request fails, the back-end will send an error message instead, which the front-end stores and then displays to the user via the use of the React-Toastify, just as mentioned before.

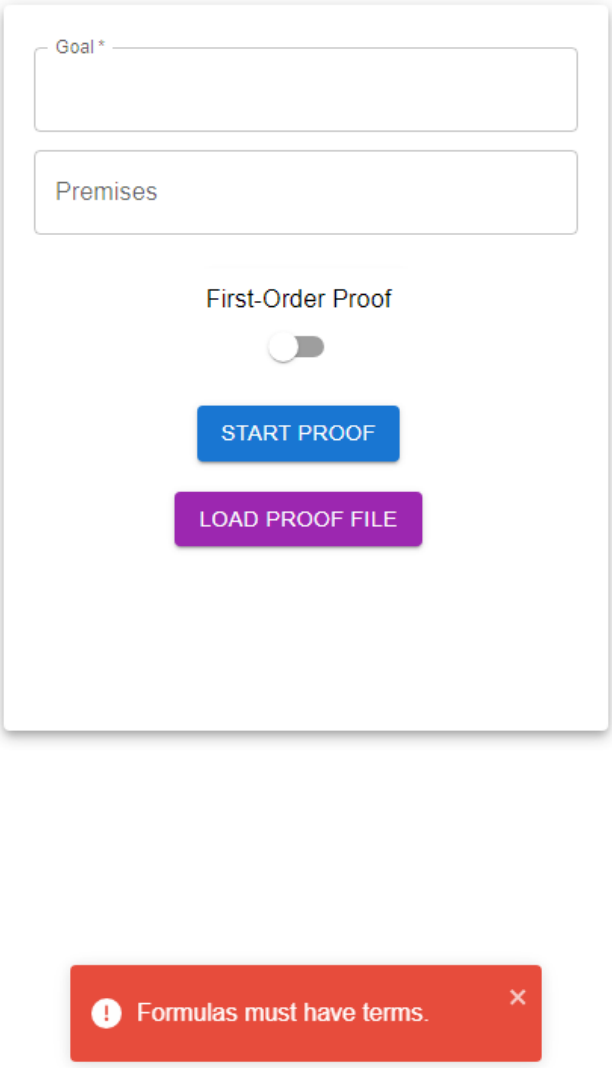


Figure 4.10: Error message shown when attempting to start a proof with an empty goal formula.

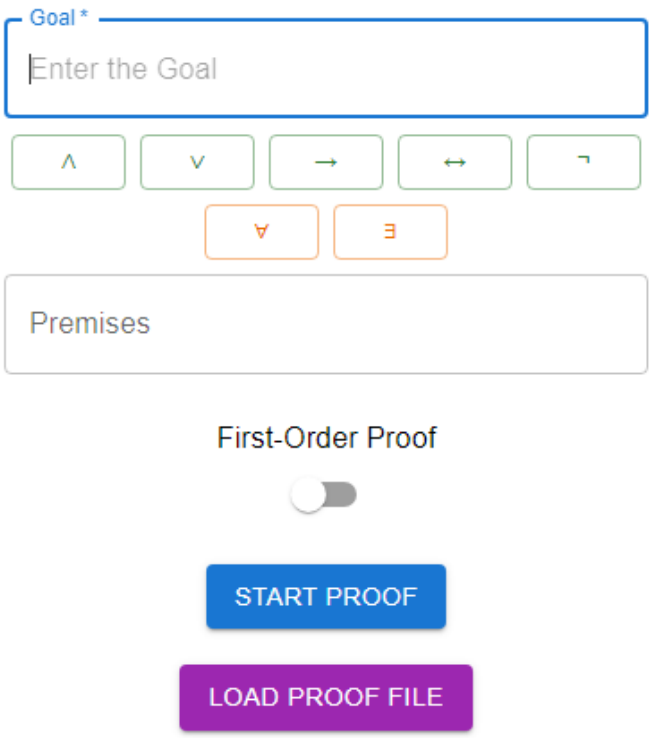


Figure 4.11: Buttons for connectives and quantifiers shown below the goal box. Clicking on the premises box will show the same set of buttons below it.

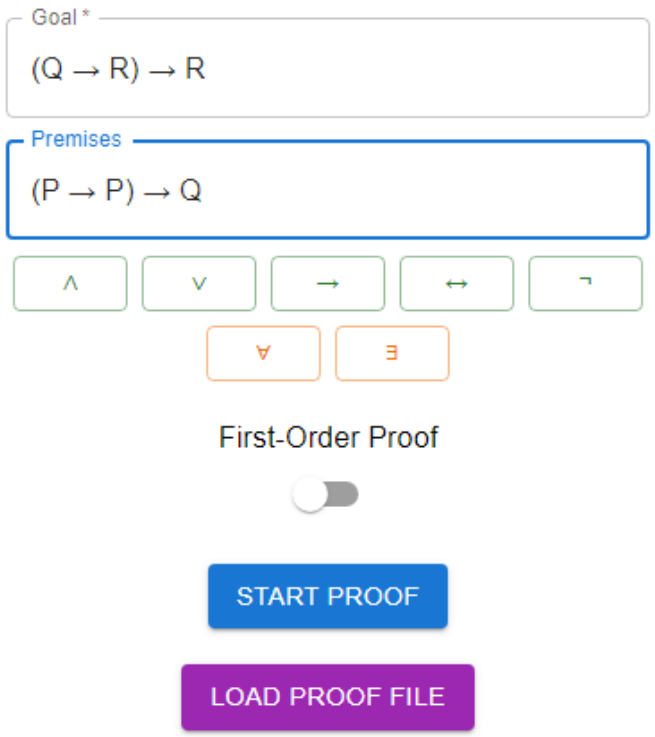


Figure 4.12: Proof box with the goal and premises of the Implication 9 proof taken from the NDPack [28] website.

4.2.1.2 Loading a Previously Created Proof

Now for the other option, loading a previously created proof, finished or not. This can be done by clicking the purple 'Load Proof' button, which will cause the front-end to display a modal, which is a pop-up box that is displayed on top of all other elements, to the user. Figure 4.13 shows the modal generated by the load proof button when there are no proof files stored. At present, the files are stored locally, in a folder contained inside the code of the project. If there are proofs stored, however, the modal displays a list of all the filenames for the available proof files, as can be seen in Figure 4.14. Every time the user enters the home page, the front-end sends a request to the back-end asking for the list of filenames. Once the back-end processes the request and responds with the list, it is stored in the React-redux store and then displayed inside the modal whenever the user clicks the load button. The user can then click on any of the file names to load that specific proof. When a file has been chosen, a request is sent to the back-end to load the corresponding proof which will be answered in the same manner as when creating a new proof, with an error message in case of failure or a StartProofDTO in case of success. The proof file shown in Figure 4.14's modal corresponds to a first-order language proof and will be shown during the next section.

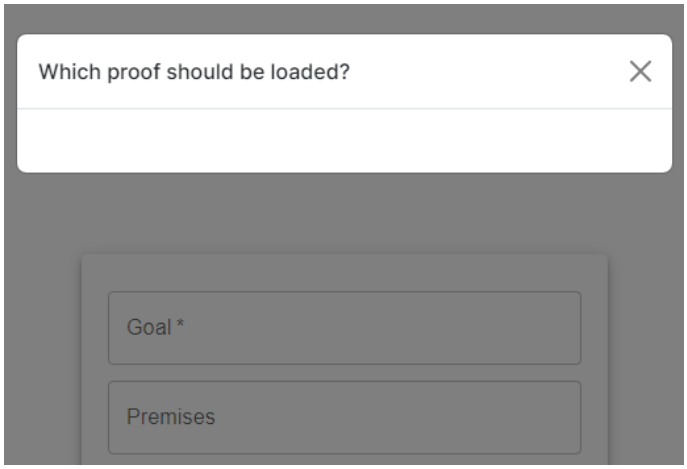


Figure 4.13: The modal shown when clicking Load Proof File, when there are no exercises present.

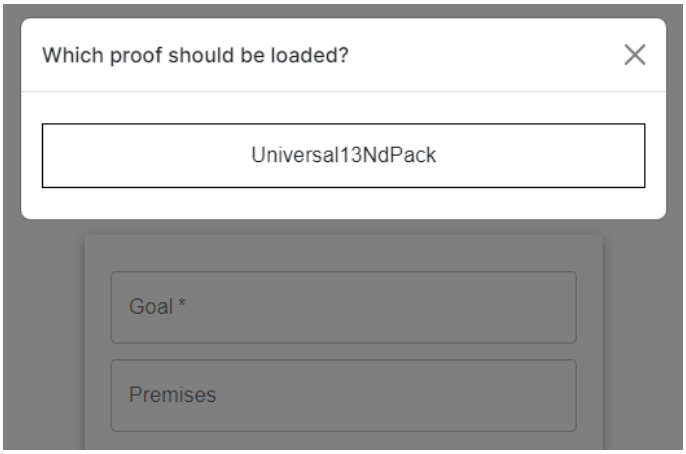


Figure 4.14: The modal shown when clicking Load Proof File, with one exercise present.

4.2.2 Proof Page and Tech Demo

4.2.2.1 Newly Created Proof

This tech demo will use the goal and premises shown in Figure 4.12. When the user clicks the Start Proof button, they are redirected to the proof page, where they will be able to view the proof and manipulate it. Figure 4.15 shows part of the proof page. The proof itself is placed in the middle of the screen, displayed as a Grid from the materialUI package. At the start, it is composed of the goal and a proof line with a dropdown button on its right.

Figure 4.15: The initial state of the proof page once Start Proof is clicked.

Starting with the left of the screen a box is present that shows the user the goal of the proof, the premises, if there are any, a button to reset the proof and another button to save the proof.

In the middle of the screen, we see the proof itself in an initial state where the only sub-goal is the goal itself. Since the back-end only supports backward reasoning, it will be solved entirely from bottom to top. The user can identify any remaining sub-goals by the presence of the dropdown button (the downward arrow button) alongside the proof line. Clicking on the dropdown button will display a long menu with several actions, as shown in Figure 4.16. The dropdown button and its menu are components of the material UI package. The many items on the menu represent actions that can be applied to a

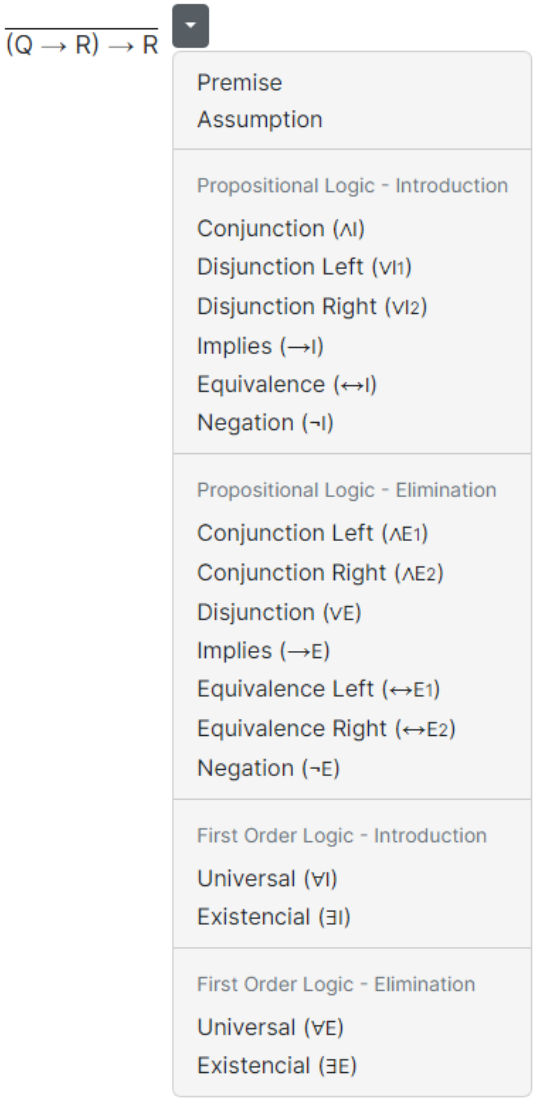


Figure 4.16: The dropdown menu shown upon clicking the button with a downwards arrow. Each item of the menu is an action that can be applied to the formula

formula, which are labeling the formula as a premise, labeling it as an assumption, or the various propositional and predicate rules. The menu separates the rules by their branch of logic and if they are elimination or introduction rules. Clicking any of these items will end up sending a request to the back-end to process that specific action for the chosen sub-goal formula. Every action that requires additional information, as shown previously by table 4.1, will first show a modal with a text field where the user can write the necessary information. Once submitted, the request is sent. The back-end will receive the request and process the action, responding with an OK message and the new state of the proof in case of success, or an error message in case of failure, as can be seen in Figure 4.17.

To proceed with the example proof, the user must apply the Introduction of Implies proof rule to the only sub-goal. Once the request is sent and responded to, the front-end

Figure 4.17: Error message shown when attempting to apply the implies elimination rule with a provided formula that doesn't have an implies connective. Error messages are set to appear on the bottom center of the screen.

updates the store with the received information and then re-renders the new state of the proof, as can be seen in Figure 4.18.

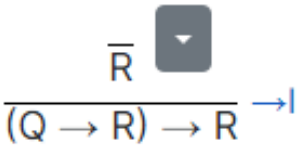


Figure 4.18: The state of the proof after the application of an introduction of implies rule to the only sub-goal.

Now the goal no longer displays the dropdown button alongside its proof line and has the next sub-goal of the proof, originated by the applied rule, on top of it. This new formula is now the one that features the dropdown button, as it is a subgoal. In this proof state, another feature can now be shown in full detail. By clicking on any of the

formulas of the proof, or its dropdown button, a box will appear on the right, as shown in Figure 4.19. This figure shows the menu for the goal formula, showing the assumptions in its domain, alongside a Delete Rule button, which, when clicked, will send a request to the back-end for the deletion of the rule applied to the formula. This button appears only to formulas with formula type 3 (inferred), and will undo everything done above that formula. Figure 4.20 shows the box for the newly generated sub-goal. As this formula does not have an inferred status, it will only show the assumptions that can be applied in its domain, with no Delete Rule button in sight.

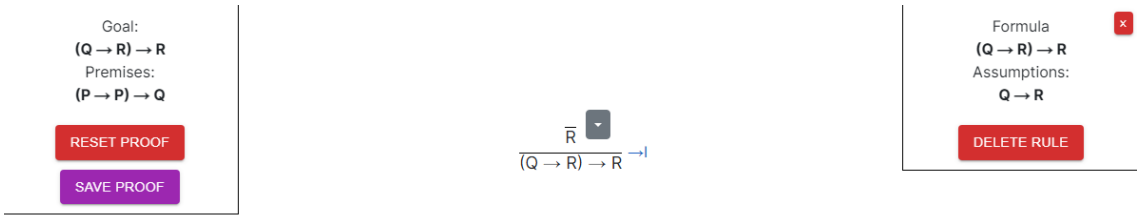


Figure 4.19: Right side box for the goal of the proof. As this is a formula that has had a rule applied to it, the Delete Rule button appears.

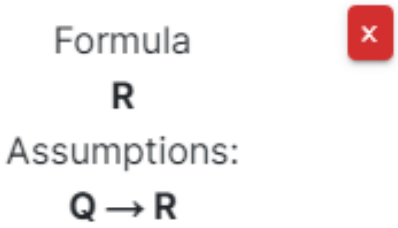


Figure 4.20: Right side box for the newly generated sub-goal of the proof, without a Delete Rule button.

The example proof can be continued by applying the Elimination of Implies rule to the only sub-goal. As this rule requires additional information to be provided by the user, a modal is generated, as can be seen in Figure 4.21. Just like with new proof creation, the user may need to write formulas with connectives or quantifiers in these modals and as such the various buttons for these characters are also present within them. Figure 4.22 shows the same modal with the formula needed to correctly advance the proof. Upon submission, a request is sent to the back-end, which in turn responds with the new proof state, which is stored and re-rendered, just like the previous rule application.

Figure 4.23 shows the proof after the application of the elimination rule, where two different subgoals can now be seen. The user can continue the proof by tackling either one of these subgoals, in any order they choose to, as the request sent from the front-end will contain information regarding which subgoal the user trying to manipulate.

Following the example proof, one of the two new subgoals will serve to show one of the actions that is not applying a rule. In this case, formula Q → R corresponds to an assumption within its domain. This can be verified by clicking the formula and seeing the

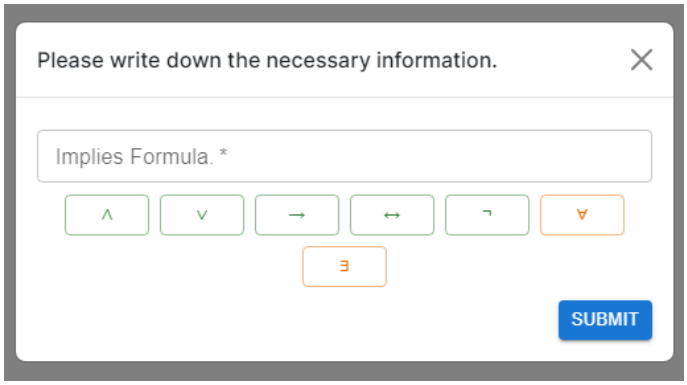


Figure 4.21: The modal shown when pressing the implies item in the dropdown's Propositional Logic - Elimination section.

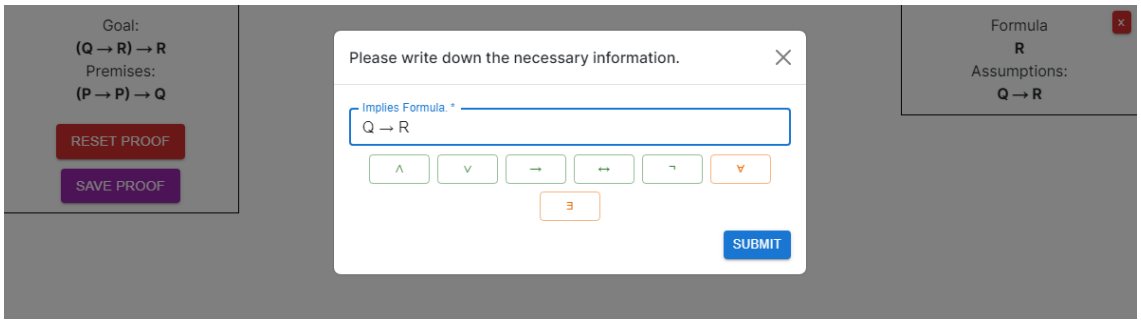


Figure 4.22: The modal filled with a formula. The rest of the proof page can also be seen below the modal, albeit obscured.

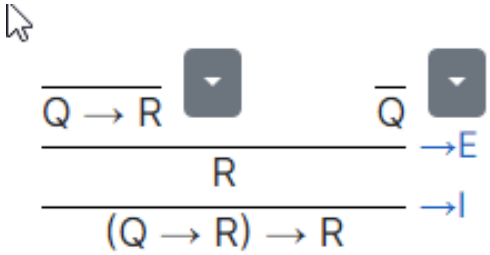


Figure 4.23: The state of the proof after the application of the elimination of implies rule to Q → R.

assumption in the top right box, as shown in Figure 4.24. This means this sub-goal can be labeled as an assumption by opening the dropdown menu and selecting Assumption. Once a response is sent and the state of the proof on the front-end is updated, we can now see the results of the action, as shown in Figure 4.25. As can be seen in this figure, the formula will now have a mark number near it, in this case 2, and the corresponding mark number will appear next to the rule which closes that mark, which is also the rule which generated the assumption.

The current proof state can be saved as a file, and then be reloaded at a later time through the 'Load Proof' button on the home page. To save the proof, the user must click

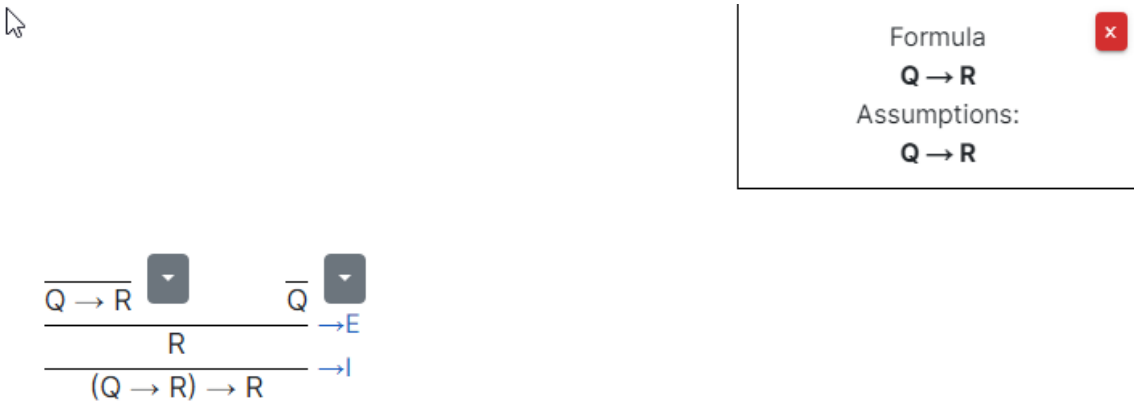


Figure 4.24: The sub-goal Q → R, matches an assumption within its domain, as shown in the top right box.

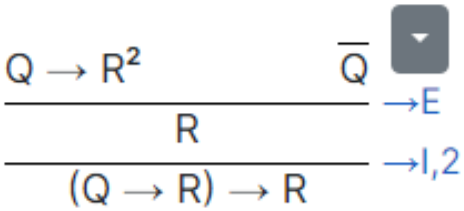


Figure 4.25: The state of the proof after the isAssumption action is applied to Q → R. A mark now appears on both the formula, and the rule which closes that mark.

on the 'Save Proof' button in the left box, which will show the user a modal requesting the user to type a name for the file, as shown in Figure 4.26. Upon providing a name and submitting, a request is sent to the back-end which will process it and create a .json file with the given file name. When reloading the file, it will have the same state as the one seen in Figure 4.25, and the user can proceed with solving the proof, which will be shown in the next section. The user can of course proceed with the original proof after saving it, there is no need to go back and reload it.

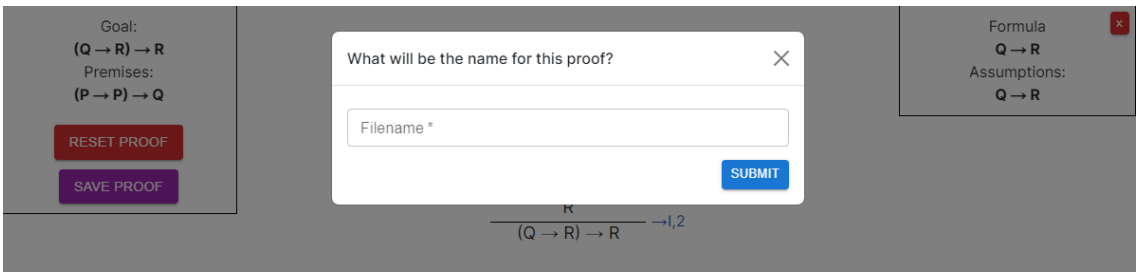


Figure 4.26: The modal shown when clicking the Save Proof button.

Lastly, the user can select the "Reset Proof" button at any time during the proof. This will send a request to the back-end which triggers the proof reset action, resetting it to either the beginning in the case of a newly created proof or from the initial state on the

file for loaded proofs.

With that, all the actions available in front-end have been shown. To end, Figure 4.27 shows the example proof in a finished state, which can be seen by the lack of dropdown menu buttons, as well as the green colored message on the bottom center of the screen.

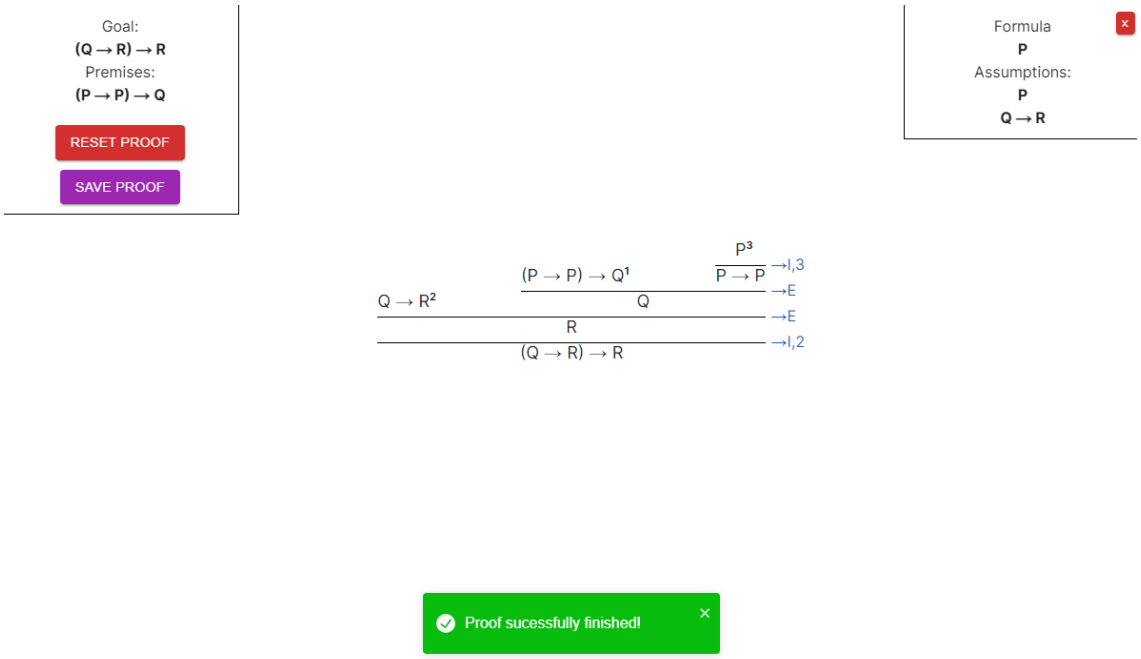


Figure 4.27: The example proof finished, complete with a message shown in the bottom center of the screen indicating the end.

4.2.2.2 Loaded Proof

This section shows a proof loaded from a file, in this case, the file seen in Figure 4.14. The proof is loaded in the state that it was saved in, in this case, having already had some actions applied to it, but not yet finished, as shown in Figure 4.28. The user can continue the proof as normal from this state, that is applying an action to one of the sub-goals or even deleting previous actions despite them being present from the initial state of the proof file, as shown in Figure 4.29. Pressing the reset button from either of these states will bring the proof back to the initial state of the proof. Figure 4.30 shows the full, completed proof.

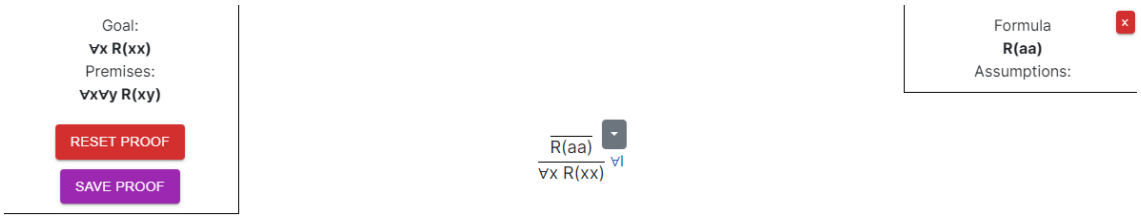


Figure 4.28: The predicate proof loaded from the 'Universal13NdPack' file. It is loaded in the same state that it was when it was saved.

Figure 4.29: The proof's state after the deletion of the introduction of universal rule.

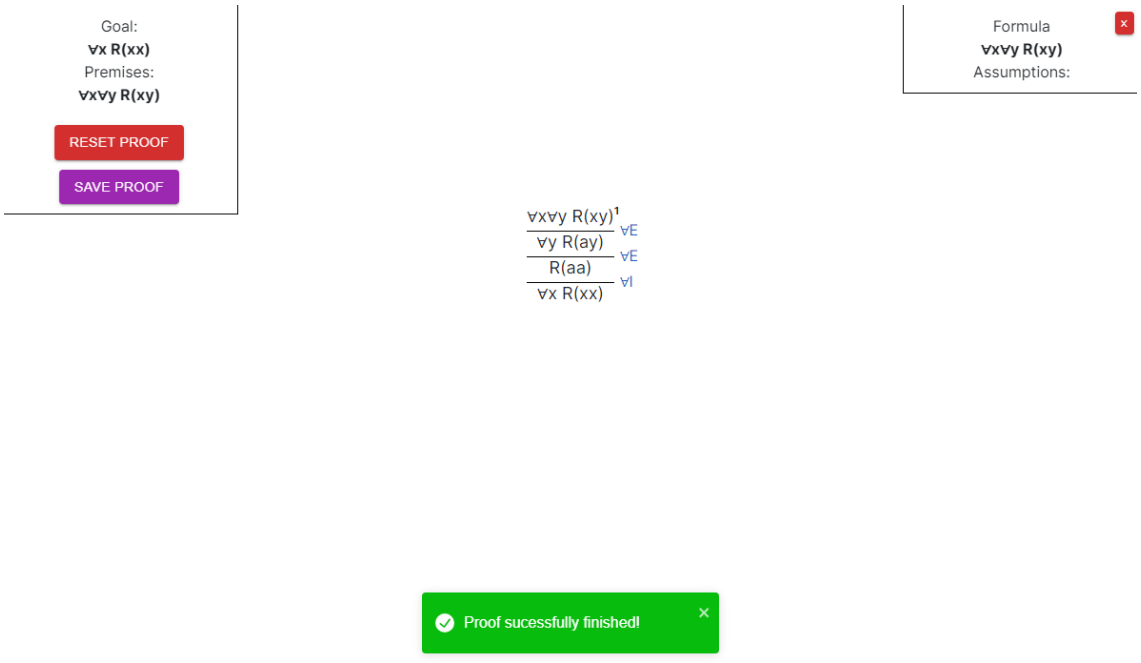


Figure 4.30: The predicate example proof finished.

4.2.2.3 Feedback Examples

To end this section here is a showcase of a few error messages that can be shown to a user.

Figure 4.31 shows a message telling the user that the action failed due to the formula the action was applied to not being part of the set of formulas in its assumptions domain.

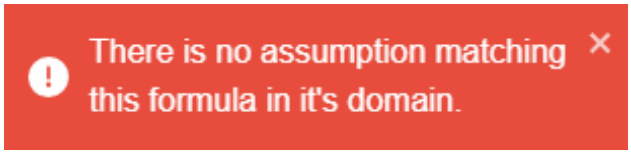


Figure 4.31: A message describing an error that can occur when the user applies the Assumption action.

Figure 4.32 shows a message telling the user that the application of the Elimination of Implies rule failed due to the right side of the user's provided formula not matching the formula to which the rule is being applied.

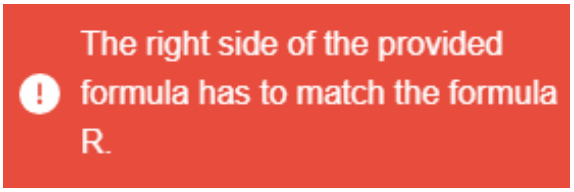


Figure 4.32: A message describing an error that can occur due to backward reasoning when the user applies the Elimination of Implies rule.

4.3 Final thoughts

Overall, the final version of the tool is a proof assistant that supports propositional and predicate logic, provides some feedback to the user when an error is committed and supports the saving and loading of proofs as files. While the system can only support backward reasoning, this still allows any proof to be constructed from bottom to top. As such, students can utilize the tool to test and verify potential solutions that they have for an exercise.

5

System Testing

Testing of the system included both the back-end's functionalities as well as the front-end's interface and has been done throughout its development whenever any changes were made. Additionally, testing heavily intensified upon completion of certain milestones, which are detailed in the next section. Additionally, there were some attempts to get users to test a complete version of the tool, however, due to various factors, we were unable to find enough users to properly conduct a user testing session.

This chapter is divided into two sections. The first section deals with the system's testing, detailing how it was done and what kind of problems and errors were found and fixed. The second section concerns the topic of user testing, the problems that appeared when attempting it and an explanation of what information was expected to be gained from it.

5.1 System Testing

System testing was done alongside its development, to make sure any new additions to the code were working correctly. This constant testing served to find any errors or issues that were more obvious, such as exceptions that were thrown or outputs that deviated from the expected. At the end of development for a new feature, there would be a period of intense testing to find less obvious errors, as well as to test the new features for various formulas. The milestones at which more intensive testing occurred were:

- Formula Parser sub-component working for propositional logic.
- Logic Reasoner sub-component working for propositional logic, in conjunction with the propositional proof rules implemented in the Proof System component.
- The two points above, but for first-order logic.
- Functional Proof Tree in the front-end's interactive interface, without connection to the back-end.
- Connection established between the back-end and the front-end.

- Functional Exercise Manager component
- Feedback and error messages added to front-end's interactive interface

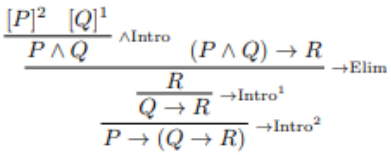
Once the formula parser was capable of parsing propositional logic formulas, testing was done by inputting several different formulas, both unary and binary, with all different types of connectors, as well as various placements of parentheses. This helped find very niche cases where the parser wouldn't correctly process the given string, mostly caused by the positioning of parentheses.

Next, for the Logic Reasoner and the propositional proof rules, testing was done by applying each rule to several different formulas, in a way that would test both the reasoning, checking whether the reasoning was correct and, if so, applying it then checking if the output of the rule application was the one that was expected. Upon conclusion of this round of testing, all of the rule applications were acting as expected for that specific isolated application. Some issues would still occur with the generation of children's formulas from rule applications and labeling as assumptions or premises, which at this stage were hard to spot due to the lack of a visual interface.

With propositional logic tested, the same steps were then done for first-order formulas, specifically if the formulas were being correctly parsed and if the rules were being correctly applied with the expected output. Additionally, during this phase some JUnit [25] tests were written most of them being complete resolutions of proofs, one unit test per proof. The remaining tests involved testing reasoning, specifically attempting to apply rules in cases where they should be invalid, and having the reasoner detect the issue and stop the rule application. These unit tests served as a baseline for any future changes to the system, ensuring that the system still functioned as expected and, if not, that something else had to be changed. Figure 5.1 shows one of these JUnit unit tests, as well as the proof it is based on.

For the next milestone, testing was done by utilizing pre-made trees and altering their values. This was useful for finding issues regarding positioning and other visual errors that the proof tree could have. During this process, it was found that the branches of the tree would eventually overlap each other once the tree grew large enough, which was fixed by turning the tree into a materialUI grid. It also helped with correctly positioning every element of the proof generated by the front-end components, such as the dropdown menu button and the marks for premises and assumptions.

Next was the connection between the back-end and front-end, a very important milestone to test since it was when it was finally possible to test all the components of the front-end's webpages, both the home page and the front page. Not only that, but it also allowed the observation of fully built proofs, which ended up being extremely important in correcting errors that remained in the back-end's functionalities that were hard to spot in previous tests. These tests consisted in attempting to solve various proofs, propositional and predicate, utilizing the interactive interface provided by the front-end. Some of the errors found and corrected during this round of testing include:




```java
@Test
public void TestImplication15() throws InvalidFormulaException {
    String goal = "P→(Q→R)";
    String[] premises = new String[1];
    premises[0] = "(P∧Q)→R";
    ProofSystem ps = new ProofSystem(goal, premises);

    assertGoalAndPremises(goal, premises, ps);
    ps.applyIntroImplies();
    assertEquals(ps.showCurrentFormula(), actual: "Q → R");
    assertTrue(ps.showAssumptions()[0].equals("P"));
    ps.applyIntroImplies();
    assertEquals(ps.showCurrentFormula(), actual: "R");
    assertEquals(ps.showAssumptions().length, actual: 2);
    assertTrue(ps.showAssumptions()[0].equals("P"));
    assertTrue(ps.showAssumptions()[1].equals("Q"));
    ps.applyElimImplies( implyFormula: "(P∧Q)→R");

    assertEquals(ps.showCurrentFormula(), actual: "(P ∧ Q) → R");
    assertTrue(ps.showAssumptions()[0].equals("P"));
    assertTrue(ps.showAssumptions()[1].equals("Q"));
    ps.isPremise();


    assertEquals(ps.showCurrentFormula(), actual: "P ∧ Q");
    assertTrue(ps.showAssumptions()[0].equals("P"));
    assertTrue(ps.showAssumptions()[1].equals("Q"));
    ps.applyIntroConjunction();
    assertEquals(ps.showCurrentFormula(), actual: "P");
    assertTrue(ps.showAssumptions()[0].equals("P"));
    assertTrue(ps.showAssumptions()[1].equals("Q"));
    ps.isAssumption();
    assertEquals(ps.showCurrentFormula(), actual: "Q");
    assertTrue(ps.showAssumptions()[0].equals("P"));
    assertTrue(ps.showAssumptions()[1].equals("Q"));
    ps.isAssumption();

}
```

Figure 5.1: The implication 15 proof from NDPack [28] on the left, with a Junit test of the proof on its right.

- Strings sent from front-end to the back-end had additional characters, confusing the parser.
- Wrongful creation of new sub-goals leading to issues with application of rules as well as problems with Spring repositories.
- Sub-goal switching mechanisms sometimes failing when different formula objects with the same formula were present in the list of sub-goals.
- Certain assumptions being available outside of their domain.

Thanks to this round of testing, all of these issues, and more, were fixed.

With the proof system itself built, tested and corrected, the next milestone involved the exercise manager, specifically when a functional version was reached. This involved

both its functionalities of the back-end as well as the corresponding buttons and other logic in the front-end. Tests for this section were done by creating proofs in the front-end, and then saving them, which tested the serialization of the manager, followed by loading the resulting JSON file and testing the deserialization. As initially expected, all tests regarding serialization went smoothly with no major errors to point out. However, this was not the case with deserialization, which revealed various errors. Errors first came from inconsistencies regarding Java objects, as previously explained in Chapter 4. These errors were fixed, however, issues started occurring with ID conflicts on the database when trying to duplicate a proof and its objects. The issue being that an ID must be stored so that the manager can establish links between what should be references to the same object. This was fixed by first assigning the objects utilizing the stored id, and then randomizing the id of the objects. Overall, this round of testing was somewhat messy due to being done via trial and error methods but resulted in all issues seemingly fixed.

Finally, the last milestone was the implementation of the feedback mechanism in the back-end's Proof System component, as well as the necessary additions to support this in the front-end. Testing for this milestone was on the shorter side and was done following these steps:

1. Attempting to create a proof with various invalid parameters.
2. Creating valid proofs and then attempting to use the wrong actions for the sub-goals, before progressing with the correct action.
3. Attempting to save the valid proofs multiple times, with and without the same name.
4. Loading the saved proofs, deleting a rule application and attempting again to use the wrong actions before finishing with the correct action.

This session of testing involved doing these steps and observing the error messages to check if the text size, color and positioning were correct, if the message shown was the correct one for the error made and if the messages were being displayed at all. The only error that seemed to occur during this session was that a message of a kind would only be displayed once until a different message was shown instead. For example, attempting to, incorrectly, apply a rule would show the error message for the first attempt, but attempting to do it again would no longer show the message until another, different, rule application was attempted. This was fixed by attaching the timestamp when the message was received to it, and splitting them when the message was to be displayed, showing only the text.

5.2 User Testing

Two sessions of user testing were planned to occur, a first, more casual session and then a more formal second session. The first session would feature a small group of individuals who would be given a tutorial video on how to utilize the tool and then pick several

exercises from the NDPack [28] to solve. The purpose of this first session was to get a "second opnion"regarding the features of the tool, as well as an attempt at finding errors that could've been missed. The feedback given in this session would then serve to ensure the tool had enough quality for a formal user test with a larger group. The second user session would then be comprised of a larger group of university students who had recently completed the computational logic course that would be asked to solve some proofs using the tool while timing their attempt, and after would be given a questionnaire to answer.

5.2.1 First Session - Casual

The first group was composed of the three university students currently in the process of developing their own master's thesis. As a more casual session, the members chosen were students known to us who made themselves available to participate. All three of these students had completed the computational logic course three years before this, where they were taught natural deduction utilizing the Fitch proof format, instead of the Gentzen Tree format used by the tool. To patch the possible knowledge disparity, they were provided with a video containing an explanation of not only the features of the tool but also a complete demo of a proof resolution, from beginning to end. With the video in hand, the group was told to complete at least five proofs, both propositional and first-order, as well as a time limit of a few days.

Once the task was completed, each one gave feedback regarding the functionality and usability of the tool, such as:

- Add a way to filter and order the proof list shown when clicking Load Proof File.
- Give feedback when a proof is successfully saved.
- Add connector and quantifier buttons to rule applications that ask for provided information.
- Add tooltips explaining certain features.

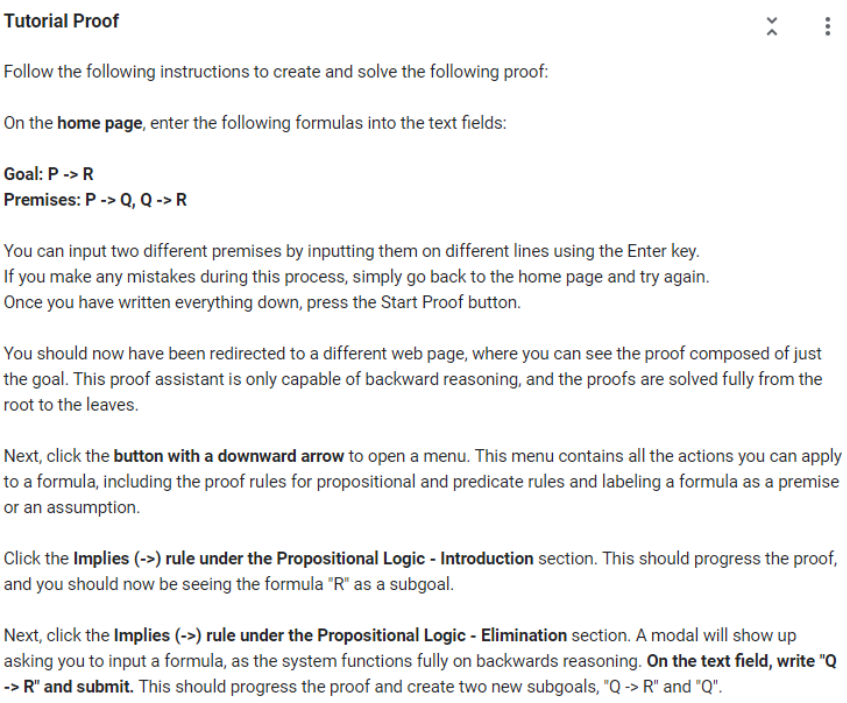
All of these suggestions are fair and would be good additions to the tool, with the third one having been added already as it was planned as such. In addition to these suggestions, the overall feedback regarding usability and visuals of the tool was good, and it seems the tool was well received with this group.

5.2.2 Second Session - Formal

As previously said, this session was to be held with a larger group of students who had recently completed the computational logic course and, as such, would have a better recollection of natural deduction concepts, how they learned them and how helpful the tool would've been to them while learning.

Unfortunately, due to the user testing for the tool only happening once all previously mentioned milestones were completed, which happened at the end of July, the attempts at finding students happened in the same period as the summer holidays, which made it extremely hard to find anyone willing to help. This would've still been the case even if it happened during the time frame of the first session. Additionally, at the beginning of September, when the students were back in classes at university, another invite was sent to all the students that frequented the computational course last year, however not even a single one accepted the invite. In the end, only one student out of the many who were asked volunteered to test the tool. While this student completed the testing and provided helpful feedback, a sample size of one is nowhere near enough to provide any conclusive data. As such, this section will instead be focused on the questionnaire that was given, the questions contained within it and what was expected out of it.

The questionnaire is divided into sections, with the first section containing a consent form for the user to read, detailing its purpose, privacy concerns and how the data would be used. The second section contained a tutorial on the usage of the tool. The user is given a tutorial proof and a step-by-step guide on how to solve most of the proof. The tutorial proof was chosen based on being capable of showing most of the tool's features. Figure 5.2 shows part of the step-by-step guide.



Tutorial Proof

Follow the following instructions to create and solve the following proof:

On the **home page**, enter the following formulas into the text fields:

Goal: P -> R
Premises: P -> Q, Q -> R

You can input two different premises by inputting them on different lines using the Enter key.
If you make any mistakes during this process, simply go back to the home page and try again.
Once you have written everything down, press the Start Proof button.

You should now have been redirected to a different web page, where you can see the proof composed of just the goal. This proof assistant is only capable of backward reasoning, and the proofs are solved fully from the root to the leaves.

Next, click the **button with a downward arrow** to open a menu. This menu contains all the actions you can apply to a formula, including the proof rules for propositional and predicate rules and labeling a formula as a premise or an assumption.

Click the **Implies (->) rule under the Propositional Logic - Introduction** section. This should progress the proof, and you should now be seeing the formula "R" as a subgoal.

Next, click the **Implies (->) rule under the Propositional Logic - Elimination** section. A modal will show up asking you to input a formula, as the system functions fully on backwards reasoning. **On the text field, write "Q -> R" and submit.** This should progress the proof and create two new subgoals, "Q -> R" and "Q".

Figure 5.2: Part of the tutorial proof section of the user testing questionnaire.

After completing all the steps, the proof is mostly finished and the user is asked to finish it. Once finished, the user is presented with two questions, which were:

- Were you able to finish the proof? (Yes/No)

- Were the instructions too hard or confusing? (1 (Easy) - 5 (Hard))

The purpose of these questions, and most of the ones in the next section are to gather data on the quality of the questionnaire for possible future user testing sessions with other groups.

The third section contains four proofs, 2 propositional and 2 first-order, and the user is asked to attempt solving each proof, while timing the attempt, if possible. Figure 5.3 shows the first of these proofs.

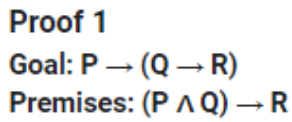


Figure 5.3: The first proof in the questionnaire.

After solving one of the proofs, the user will have at least four questions to answer. These four questions are the same for all four proofs and are shown in figure 5.4. The first three, which require an answer, serve both to improve the quality of future questionnaires as well as to gather data regarding the user's knowledge of natural deduction. The last one is optional as it required the user to make an error at some point, and was meant to gather data regarding the quality of the feedback and error messages displayed by the tool. One of the four proofs additionally requested the user to save the proof and then, later, attempt to load it. For that proof there are an additional two questions, asking the user how easy it was to save and load the proof, on a scale of 1 to 5.

The fourth section contains several questions regarding the tool, which are independent of any proof. These questions are:

1. Is the tool easy to use? (1 - 5)
2. Is it easy to learn how to use the tool? (1 - 5)
3. Are the web pages visually confusing? (Agree, Neutral, Disagree)
4. Is the tool more complex than what it should be? (1 - 5)
5. Do you think you would require help from an expert on the tool to be able to use it? (Yes, Depends on proof, No)
6. Does the tool have every feature that is necessary for a tool of its type? (1 - 5)
7. Does the lack of forward reasoning make it harder to solve proofs? (Yes, No but it would be helpful, No)
8. Is the exercise management component easy to use? (1 - 5)
9. Is this a helpful tool to have when learning Natural Deduction? (1 - 5)

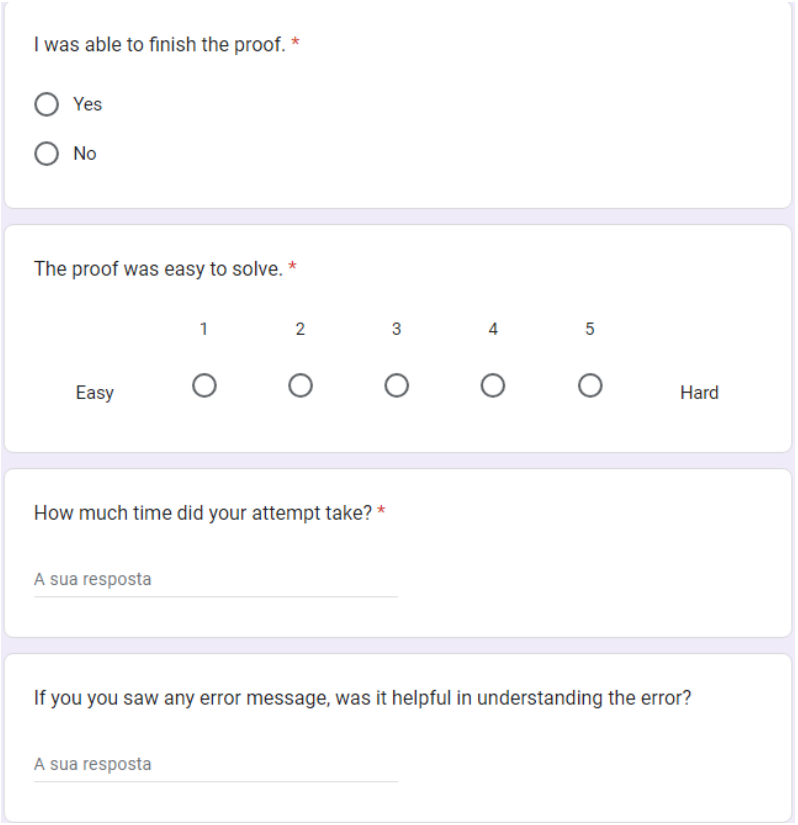


Figure 5.4: Four questions of the third section of the questionnaire, which are repeated for each of the proofs.

In general, these questions touch on the quality of features and the usability of the system. Questions 1, 2, 4 and 5 all serve to gather data on the usability of the tool, and help in understanding if the current state of the tool fits what is expected from a student, or if there are parts that need to be changed or improved. Questions such as 2 and 4 can initially be seen as variations of the same question, as it could be assumed that a tool that is too complex would be hard to use, but this is not the case vice-versa, as a non-complex tool can still be hard to utilize. For all four of these, we would ideally be looking for lower values on the scales or positive answers. A good amount of answers on the other end of the scale would indicate a need for change.

Question 3 asks the user for feedback regarding the visual side of the tool. While this is not seen as a priority, it would be an issue if the user experience was impacted by poor visuals and confusing elements.

Question 6 asks the user if they agree that the tool has every feature necessary for a proof assistant. Should the user disagree, they would be able to write down what was missing at the end of the questionnaire.

Question 7 was initially part of the group of questions in section three but was thought to be a better fit for this section. It questions the user regarding the lack of forward reasoning in the tool and if it affected the usability of the tool. As this is the most notable feature missing from the tool, it was preferred to have a question about it rather than have

it noted at the end of the questionnaire.

Question 8's topic was the exercise manager component. While this was already asked in section 3, those questions referred to a specific proof, while this one referred to the mechanism as a whole, independent of proof, as it was encouraged for the user to attempt to use the mechanism after first coming in contact with it.

Finally, question 9 asked the user for their opinion, on a 1 to 5 scale, of the use of a tool like this as a complement to the task of learning natural deduction. While it is standard to use proof assistants in the learning process, we still wanted the users in this session to provide their own value of it.

The fifth section has a couple of questions that establish the demographic of the user group, such as age, gender and native language.

Lastly, the sixth section has a small thank you note to the user as well as a text box where the user can write any feedback they have for the tool.

6

Conclusion

This thesis focused on developing an online tool whose function is to help students learn natural deduction by acting as a complementary tool to a computational logic course. Its design follows several criteria that are considered to be beneficial to its goal, such as providing feedback and supporting exercise management features. After a search was done to find a previously available tool that features all of these criteria, none was found and it was decided that the tool had to be developed from scratch, inspired by the work done by other tools of this kind.

The tool that was developed consists in a front-end, built with React, that features an interactive proof assistant that displays proofs in the Gentzen Tree format, as well as a Java back-end that processes the actions done by users in the interactive proof assistant. Users are allowed to either create a new proof from the start or load a previously saved proof. They can then solve the proof from bottom to top while receiving feedback whenever an error is committed. Unfortunately, a user testing session featuring a large group of students couldn't be conducted due to the lack of volunteers, mostly due to the time period where it occurred. The few users who tested the tool had a positive reaction to it. Still, the system was tested often during development, including periods of intensive testing that occurred when a milestone was complete, which revealed several errors and problems that were corrected.

Still, there is work that could be done in the future to improve the tool. First, the system would benefit from having forward reasoning added to it, as the presence of both types of reasoning is consistent with how a student would solve a proof on paper. This could be done in a couple of ways, one of them being how the proof assistant Panda [6] did it, as shown in chapter 3. Second, integration with an online platform such as Moodle, which could come with other helpful features such as control over who can save and load a proof, anonymous data collection and even automatic grading. Lastly, a refactoring of the formula parser component, turning it into a true token-based system, instead of the current character-by-character implementation. This would allow the capability to, for example, write the name of the connector or quantifier and have the system understand what that is, for example, writing forall instead of the character ∀. The character-by-character parser

was the first component to be developed, originally as a test after the formula objects were established. However, after the many days lost attempting to connect the external reasoner, eventually leading to its discarding, there were concerns over taking too much time rebuilding the parser and work proceeded with the character-by-character parser instead.

Bibliography

[1] F. ABRAHAMSSON et al. *Proof Editor for Natural Deduction*. 2021. URL: https://odr.chalmers.se/server/api/core/bitstreams/e3cadeaa-efab-4e66-9a18-a41af5617d3e/content (cit. on pp. 3, 27).

[2] Alrubyli and Yazeed. "Natural Deduction Calculus for First-Order Logic". In: *CoRR* (2021). DOI: 10.48550/ARXIV.2108.06015. URL: https://arxiv.org/abs/2108.06015 (cit. on p. 14).

[3] E. Bartzia, A. Meyer, and J. Narboux. "Proof assistants for undergraduate mathematics and computer science education: elements of a priori analysis". In: *INDRUM 2022: Fourth conference of the International Network for Didactic Research in University Mathematics*. Ed. by M. Trigueros. Reinhard Hochmuth. Hanovre, Germany, 2022. URL: https://hal.science/hal-03648357 (cit. on pp. 14, 19).

[4] J. Barwise et al. *Language, proof and logic*. CSLI publications, 2000. ISBN: 157586374X (cit. on pp. 20, 21).

[5] J. M. Fitzpatrick et al. "Lessons Learned in the Design and Delivery of an Introductory Programming MOOC". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '17. Seattle, Washington, USA: Association for Computing Machinery, 2017, pp. 219–224. ISBN: 9781450346986. DOI: 10.1145/3017680.3017730. URL: https://doi.org/10.1145/3017680.3017730 (cit. on p. 2).

[6] O. Gasquet, F. Schwarzentruber, and M. Strecker. "Panda: A proof assistant in natural deduction for all. A Gentzen style proof assistant for undergraduate students". In: *Tools for Teaching Logic: Third International Congress, TICTTL 2011, Salamanca, Spain, June 1-4, 2011. Proceedings*. Springer. 2011, pp. 85–92. ISBN: 3642213499 (cit. on pp. 3, 24, 25, 75).

[7] G. Geck et al. "Iltis: Learning Logic in the Web". In: *CoRR* abs/2105.05763 (2021). DOI: 10.48550/ARXIV.2105.05763. URL: https://iltis.cs.tu-dortmund.de (cit. on pp. 2, 17, 18).

[8] G. Geck et al. "Introduction to Iltis: An Interactive, Web-Based System for Teaching Logic". In: ITiCSE 2018 (2018). DOI: 10.1145/3197091.3197095. URL: https://doi.org/10.1145/3197091.3197095 (cit. on pp. 2, 17, 18).

[9] S. Hedman. *A First Course in Logic: An introduction to model theory, proof theory, computability, and complexity*. OUP Oxford, 2004. ISBN: 0198529813 (cit. on p. 5).

[10] D. Lebron and H. Shahriar. "Comparing MOOC-based platforms: Reflection on pedagogical support, framework and learning analytics". In: *2015 International Conference on Collaboration Technologies and Systems (CTS)*. 2015, pp. 167–174. DOI: 10.1109/CTS.2015.7210417 (cit. on p. 1).

[11] S. Lovrenčić and M. Čubrilo. "OVERVIEW OF ONLINE TEACHING RESOURCES FOR LOGIC PROGRAMMING". In: *The Seventh International Conference on eLearning* (2016) (cit. on pp. 2, 17).

[12] B. Machin and L. Sierra. "Yoda: a simple tool for natural deduction". In: *Tools for Teaching Logic: Third International Congress, TICTTL 2011, Salamanca, Spain, June 1-4, 2011. Proceedings*. 2011. ISBN: 3642213499 (cit. on p. 24).

[13] P. D. Magnus and T. Button. *Forall X: Calgary: An introduction to formal logic*. 2021. ISBN: 9798527349504. URL: https://forallx.openlogicproject.org/forallxyyc.pdf (cit. on p. 26).

[14] J.-F. Monin, C. Ene, and M. Périn. "Gentzen-Prawitz Natural Deduction as a Teaching Tool". In: (2009). DOI: 10.48550/ARXIV.0907.3599. URL: https://arxiv.org/abs/0907.3599 (cit. on p. 2).

[15] L. O'Connor and R. Amjad. "Holbert: Reading, Writing, Proving and Learning in the Browser". In: *arXiv preprint arXiv:2210.11411* (2022-10). DOI: 10.48550/arxiv.2210.11411. URL: https://arxiv.org/abs/2210.11411v1 (cit. on p. 20).

[16] B. Rognier and G. Duhamel. "Présentation de la plateforme edukera". In: *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*. Ed. by J. Signoles. Saint-Malo, France, 2016. URL: https://hal.science/hal-01333606 (cit. on pp. 10, 18, 19).

[17] P. Sands and A. Yadav. "Self-Regulation for High School Learners in a MOOC Computer Science Course". In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 845–851. ISBN: 9781450367936. DOI: 10.1145/3328778.3366818. URL: https://doi.org/10.1145/3328778.3366818 (cit. on p. 2).

[18] E. Z. Yang. "Academic software reuse, an experience report". In: *6.UAP report advised by Adam Chlipala* (2012) (cit. on pp. 23, 24).

[19] A. Yushkovskiy. "Comparison of Two Theorem Provers: Isabelle/HOL and Coq". In: *arXiv preprint arXiv:1808.09701* (2018). DOI: 10.48550/ARXIV.1808.09701. URL: https://arxiv.org/abs/1808.09701 (cit. on pp. 14, 15).

Webography

[20] J. Avigad. *Classical and constructive logic*. (Visited on 2023-01-28) (cit. on pp. 5, 8).

[21] *Fitch-Checker*. URL: https://proofs.openlogicproject.org/ (visited on 2023-01-16) (cit. on p. 26).

[22] *Internet Encyclopedia of Philosophy - Natural Deduction*. URL: https://iep.utm.edu/natural-deduction/ (visited on 2023-01-15) (cit. on p. 8).

[23] *Jackson*. URL: https://github.com/FasterXML/jackson (visited on 2023-06-01) (cit. on p. 47).

[24] *JsCoq*. URL: https://coq.vercel.app/ (visited on 2023-01-14) (cit. on p. 15).

[25] *JUnit*. URL: https://junit.org/junit5/ (visited on 2023-07-01) (cit. on p. 66).

[26] *Logan*. URL: https://datx02-21-16.github.io/datx02/ (visited on 2023-01-18) (cit. on p. 27).

[27] *MaterialUI*. URL: https://mui.com/ (visited on 2023-04-15) (cit. on p. 51).

[28] *Natural Deduction Pack*. URL: https://users.ox.ac.uk/~logicman/carr/NDpack.pdf (visited on 2023-01-15) (cit. on pp. 12, 15, 16, 54, 67, 69).

[29] *Openapi Generator*. URL: https://github.com/OpenAPITools/openapi-generator (visited on 2023-05-01) (cit. on p. 52).

[30] *React Redux*. URL: https://react-redux.js.org/ (visited on 2023-05-01) (cit. on p. 51).

[31] *React Router*. URL: https://reactrouter.com/en/main (visited on 2023-04-15) (cit. on p. 51).

[32] *React Toastify*. URL: https://fkhadra.github.io/react-toastify/introduction (visited on 2023-07-10) (cit. on p. 51).

[33] *ReactJs*. URL: https://reactjs.org/ (visited on 2023-02-05) (cit. on pp. 33, 51).

[34] *Spring*. URL: https://spring.io/ (visited on 2023-04-01) (cit. on pp. 32, 48).

[35] *Spring Boot*. URL: https://spring.io/projects/spring-boot (visited on 2023-04-01) (cit. on p. 48).

[36] *Tactics*. URL: https://coq.inria.fr/refman/proof-engine/tactics.html (visited on 2023-01-23) (cit. on p. 15).