CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Proof Editor for Natural Deduction

Bachelor's thesis in Computer science and engineering

FREDDY ABRAHAMSSON, THERESE ANDERSSON,
AXEL FORSMAN, LO RANTA, MICHAEL ÅKESSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Bachelor's thesis 2021

Proof Editor for Natural Deduction

FREDDY ABRAHAMSSON,
THERESE ANDERSSON
AXEL FORSMAN,
LO RANTA,
MICHAEL ÅKESSON

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Proof Editor for Natural Deduction
FREDDY ABRAHAMSSON, THERESE ANDERSSON, AXEL FORSMAN,
LO RANTA, MICHAEL ÅKESSON

© FREDDY ABRAHAMSSON, THERESE ANDERSSON, AXEL FORSMAN,
LO RANTA, MICHAEL ÅKESSON 2021.

Supervisor: Ana Bove, Department of Computer Science and Engineering
Examiner: Thierry Coquand, Department of Computer Science and Engineering

Bachelor's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Screenshot of Logan, with a proof of $\forall x\,(P(x) \to Q(x)), \forall x\,P(x) \vdash \forall x\,Q(x)$

Typeset in LaTeX
Gothenburg, Sweden 2021

Proof Editor for Natural Deduction
FREDDY ABRAHAMSSON, THERESE ANDERSSON, AXEL FORSMAN,
LO RANTA, MICHAEL ÅKESSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In this thesis, we present *Logan*, a proof editor for constructing Fitch-style proofs in first-order logic. This proof editor is intended to be used by students who are taking a course in logic. Compared to other available proof editors Logan aims to be easier to use, both when it comes to getting started and in the continued use of the proof editor. Logan also aims to be more helpful by giving feedback and hints to aid the students in their learning. User testing was conducted on a small sample size of students to evaluate Logan from a usability and user experience perspective to find potential issues. As a result, some usability problems were identified and remedied. The plan is to introduce Logan as a tool in the next instance of the course DAT060/DIT202 *Logic in computer science* at University of Gothenburg and Chalmers University of Technology. If realized, the deployment of Logan can be seen as beta-testing on a larger sample size of students than the conducted user testing. Thus potential missed bugs can be identified and fixed in future development of Logan.

Keywords: Proof editor, Natural deduction, First order logic, PureScript.

Sammandrag

I den här rapporten presenterar vi *Logan*, en beviseditor för att konstruera bevis med fitch form i första ordningens logik. Den här beviseditorn är gjord för att användas av studenter som läser en kurs i logik. Jämfört med andra tillgängliga beviseditors så försöker Logan vara mer lättanvänd, både vad gäller att komma igång och i den fortsatta användningen av beviseditorn. Logan erbjuder också användarna mer hjälp i form av återkoppling och ledtrådar för att främja deras inlärning. Användartester i en liten skala på studenter gjordes för att utvärdera Logans användbarhet och användarupplevelse för att hitta potentiella problem. Under dessa upptäcktes några problem relaterade till användbarhet som löstes därefter. Planen är att introducera Logan som ett verktyg i nästa omgång av kursen DAT060/DIT202 *Logic in computer science* vid Göteborgs Universitet samt Chalmers Tekniska Högskola. Om detta blir realiserat så kommer driftsättningen av Logan kunna ses som en betatestning på en större skala av studenter jämfört med de genomförda testerna. Därigenom kan potentiella buggar som har missats bli identifierade och åtgärdade för vidare utveckling av Logan i framtiden.

Nyckelord: Beviseditor, Naturlig deduktion, Första ordningens logik, PureScript.

Acknowledgements

We would like to thank our supervisor Ana Bove for proposing this project as well as providing stellar support and feedback during the whole process. We would like to acknowledge the students who took their time to help us evaluate our product and provide us with suggestions for improvements. Also, a big thank you to our families and friends, for all the support and understanding.

Freddy Abrahamsson, Therese Andersson, Axel Forsman, Lo Ranta, Michael Åkesson
Gothenburg, June 2021

Glossary

DOM Document Object Model. Representation of a Web page as nodes. 27
DSL Domain-specific Language. 3, 5–7

GUI Graphical User Interface. 3, 6

Kanban is a method for managing, among other things, a development process where progress is visualized using cards on a board. 17

LEM Law of Excluded Middle. 11

MT Modus Tollens. 11

natural deduction is a proof calculus where step-by-step application of inference rules is used to derive conclusions from premises. 1

PBC Proof By Contradiction. 11

TUI Text-based User Interface. 3, 6

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Purpose . . . 1
1.2 Benefits of Using a Proof Editor . . . 2
1.3 Currently Available Software . . . 2
1.3.1 Types of Interface . . . 3
1.3.2 Target Audience . . . 6
1.4 Choice of Programming Language . . . 6
1.5 Limitations . . . 7

2 Theory **9**
2.1 Propositional Logic . . . 9
2.2 Predicate Logic . . . 10
2.3 Natural Deduction . . . 11
2.3.1 Automated Natural Deduction . . . 15
2.4 Usability and User Experience . . . 15

3 Methods **17**
3.1 Developing Logan . . . 17
3.2 Evaluating Logan . . . 17

4 Results **23**
4.1 Application . . . 23
4.1.1 Representation of a Proof . . . 23
4.1.2 Validation of Proofs . . . 25
4.1.3 Implementation of the Web Application . . . 26
4.1.4 The Final Product . . . 27
4.2 User Tests . . . 31
4.2.1 Basic Functionality . . . 31
4.2.2 Constructing Proofs . . . 35
4.2.3 Interviews . . . 36
4.2.4 Usability Improvements . . . 38

5 Discussion **41**

5.1 The Final Product 41
5.2 The User Testing 41
5.3 Future Work 44

6 Conclusion **47**

Bibliography **49**

List of Figures

1.1 Screenshot of fitch-checker, running at `https://proofs.openlogicproject.org/`. In this particular screenshot, a proof of the sequent $R, R \to U \vdash U$ is shown. 3

1.2 Screenshot of the Conan editor, available at `https://github.com/nonilole/Conan`. In this particular screenshot, a proof of the sequent $R, R \to U \vdash U$ is shown. 4

1.3 Proof editors such as Alfa and Lean have textual interfaces. This is a screenshot of Lean, showing an incorrect proof of the sequent $\forall x \forall y\, (x = g(y) \to f(x) = y) \vdash \forall x\, f(g(x)) = x$. 5

1.4 Agda has a textual interface, here seen in the EMACS editor. The screenshot shows a hole, the goal and what is in scope. From the types it is clear that `a` is the only thing in scope that could fit into this hole and represent the goal `A`. 5

2.1 The proof rules for propositional logic as defined in *Logic in computer science* [3]. 12

2.2 A simple proof in natural deduction, proving the sequent $R, R \to U \vdash U$. 13

2.3 Proof of the sequent $A \wedge B \to C, A \vee D, B \vee E, \neg(D \vee E) \vdash C$. Note the use of boxes which make up the arguments for $\vee$-elimination on rows 10 and 16. 13

2.4 Additional inference rules for predicate logic. 14

2.5 Proof of the sequent $\forall x\, (P(x) \to Q(x)), \forall x\, P(x) \vdash \forall x\, Q(x)$. 15

3.1 Test case 1: $P \wedge Q, R \vdash Q \wedge R$. It is indeed possible to construct a proof of this using natural deduction, but the specific proof outlined here is not correct. The rule that is applied on the third line should be $\wedge e_2$, and the operands to $\wedge i$ on line 4 should be flipped. 19

3.2 Test case 2: $P \to (Q \to R) \vdash P \wedge Q \to R$. This is provable using natural deduction, but there are some minor details that are wrong in this particular proof. On line 7 the introduction rule is applied to no arguments at all, but it should be applied to the entire subproof spanning lines 2-6. Furthermore, when going from line 6 to 7 the participants must close the proof box. By default new rows are added in the current proof box, but by pressing the `Shift` and `Enter` keys simultaneously the new row will be inserted after closing the current proof box. 20

3.3 Test case 3: $Q \to R \vdash P \lor Q \to P \lor R$. This proof contains no errors, but it will test the participants intuition when it comes to using nested proof boxes. They are expected to use some rules they have not used in the previous tests. 20
3.4 Test case 4: $P \land \neg Q \to R, \neg R, P \vdash Q$. The errors in this proof are that there is a redundant implication arrow on the first line, and on line 9 the rule is applied to line 88 instead of line 8. The proof editor will always open a new box when an assumption is introduced, but in this proof there is no box. The proof editor will open the box but the participants will not have closed it if they simply entered the proof as it is written here. 21

4.1 Definition of the `Rule` type and a few of its constructors. 24
4.2 Definition of the `ProofRow`, `Scope`, `Proof` and `ND` along with the type signature for the function `runND` which performs the actual validation. 24
4.3 Definitions for the `Variable`, `Term`, `Formula` and `FFC` types used to represent different parts of a formula. 25
4.4 Type signatures for the `proofRef` and `boxRef` functions along with parts of the definition of the `applyRule` function. 26
4.5 Screenshot of the editor, showing a proof of the sequent $R, R \to U \vdash U$. 28
4.6 Screenshot of Logan with an error message. The output from $\land$i does not match the formula on line 3. 29
4.7 Screenshot of Logan, showing an explanation for the $\to$e rule, which the user has already tried to use. Since there is an error present in the proof, the last line is not highlighted in green, even though it matches the conclusion. 30
4.8 The interface of Logan at the time of user tests. 32
4.9 The rule field of the editor with buttons for each rule. When a button is clicked a help text about the rule replaces the text below the buttons. 33
4.10 Time in seconds taken to complete each task in observation 1. Task 2, adding a premise, took a long time for most users. 34
4.11 User ratings for visual design and ease of finding information. 39

List of Tables

2.1 The logical connectives present in propositional logic, with their symbolic representations and example sentences. 10
2.2 The derived proof rules used in this project, as presented in [3]. . . . 12

4.1 Instructions for testing basic functionality. 32

1

Introduction

Logic is one of the most important tools for reasoning about computer software and its correctness, making it an important subject for students in the field of computer science.

In addition to theoretical studies, students need to learn practical applications of logic, such as proving theorems or checking software for correctness. One important technique that students will encounter during university courses in logic is natural deduction. Natural deduction is a system for proving formulas in both propositional and predicate logic. It uses inference rules, based on the syntactic structure of logical formulas, to infer conclusions from existing formulas, called premises. More details about logic and the theory behind natural deduction can be found in chapter 2.

1.1 Purpose

The main goal of this project is to develop a proof editor for natural deduction in both propositional and predicate logic. In the editor, the user should be able to construct proofs step by step, starting with a set of premises and then applying predefined rules to reach some conclusion.

In order to help students learn natural deduction, the editor will provide some assistance such as highlighting errors where they occur and showing information about what went wrong. Concretely, the four goals of the project are

- The proof editor should not require any set-up or installation.
- The proof editor should have an intuitive and easy to use interface and method of user input.
- The proof editor should give feedback when a mistake is made.
- The proof editor should be structured in a way that aids in making correct proofs.

1.2 Benefits of Using a Proof Editor

As students are learning natural deduction, they are frequently asked to construct proofs as exercises, as part of their coursework and as part of their exams [1]. Usually, the students will construct their proofs using pen and paper, as it is a simple and easily accessible method. This method, however, relies on the students' own competence of the subject as no feedback is provided while the proof is constructed. A student who struggles with an inference rule will not be corrected when constructing the proof, with an increased risk of the rule being used incorrectly. Errors like these might not be discovered until the student is ultimately assessed on their competence of natural deduction. Even worse would be if the student is not tested on any rules they do not understand, in which case the error will not be discovered and remedied. If the student had constructed their proofs using a proof editor, the editor could have informed the student that the rule is applied wrongly, and provide the much-needed feedback.

In addition to the feedback, a proof editor can also help the students by providing hints on which steps to take when stuck. For propositional logic, proofs can be constructed automatically, which means that the editor can create a complete proof behind the scenes and use that to point the user in the right direction [2]. In predicate logic, there is a limitation on the level of assistance that can be provided by software due to the undecidability of predicate logic [3]. However, it is still possible to validate each step of a proof in predicate logic and provide feedback to the students.

1.3 Currently Available Software

Using proof editors to write machine checked proofs is not a novel idea. There already exists a number of general-purpose proof editors such as *Agda* [4], *Coq* [5], *Alfa* [6] and *Lean* [7], and more niche editors such as *Conan* [8] and *fitch-checker* [9].

General-purpose proof editors such as Agda can be used to write and check very sophisticated proofs. They come with a lot of functionality and features that allow a user to encode their proofs very efficiently and precisely. This level of abstraction leads to a higher threshold when it comes to the efficiency of the user. To write proofs in Agda it is required that the user is already familiar with e.g functional programming, and preferably also dependent type theory.

A user that is only interested in constructing proofs of a certain kind might be better off using a more niche editor that is designed specifically for those types of proofs. As an example, Conan only offers construction of proofs using natural deduction and does not implement any of the general features and abstractions that Agda does. A user will learn to use Conan very quickly and can start writing proofs faster.

1.3.1 Types of Interface

The interfaces with which a user interacts with an editor is usually one of two kinds. One kind is that of a Graphical User Interface (GUI) and the other is that of a Text-based User Interface (TUI).

As can be seen in figures 1.1 and 1.2, both fitch-checker and Conan are GUI based interfaces. The upside of having such an interface is that the learning curve is not as steep since the user has more visual elements to help guide the learning process. There is no need to study syntax or remember specific commands [10].

Figure 1.1: Screenshot of fitch-checker, running at `https://proofs.openlogicproject.org/`. In this particular screenshot, a proof of the sequent $R, R \to U \vdash U$ is shown.

Proof editors such as Alfa and Lean have textual interfaces where proofs are constructed in a Domain-specific Language (DSL), see figure 1.3. The upside of using a DSL is that the proofs can become very compact and short. The meaning of

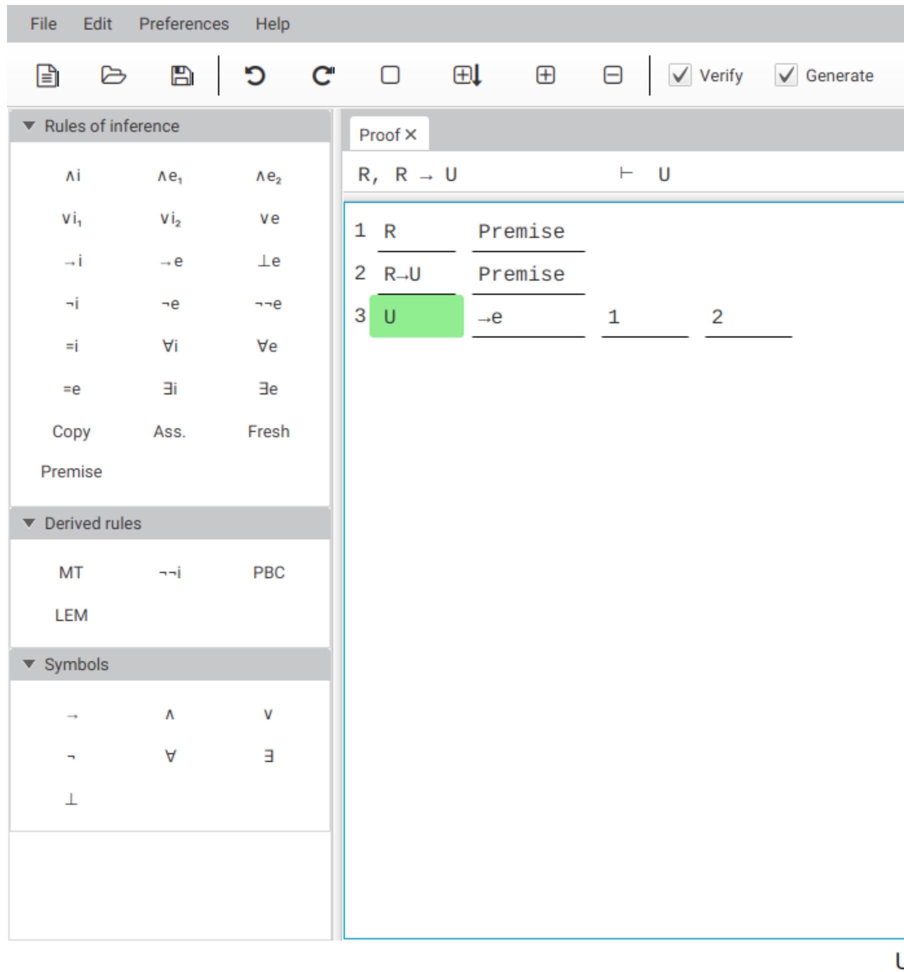

Figure 1.2: Screenshot of the Conan editor, available at `https://github.com/nonilole/Conan`. In this particular screenshot, a proof of the sequent $R, R \to U \vdash U$ is shown.

the components of a proof can become more intuitive as the language is tailored specifically for the domain.

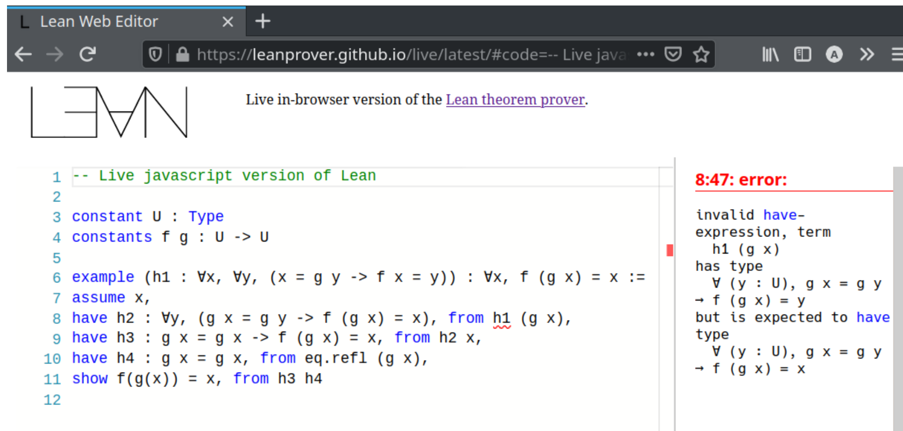

Figure 1.3: Proof editors such as Alfa and Lean have textual interfaces. This is a screenshot of Lean, showing an incorrect proof of the sequent $\forall x \forall y\, (x = g(y) \rightarrow f(x) = y) \vdash \forall x\, f(g(x)) = x$.

Another example of a textual interface can be seen in figure 1.4. Agda, as well as Coq, is a proof *assistant*. Conan and Fitch assist a user by checking the validity of a proof, but e.g. writing proofs in the DSL of Agda and Coq can be done in an environment where users are assisted much more by the editor. For example Agda can make suggestions of how to complete proofs, show information of conclusions that are currently available and help write the actual code. This requires learning not only a DSL but also a set of commands to interact with the environment. The learning curve is steeper, but once a user has grasped all the knowledge of how to construct proofs using the editor, the efficiency in constructing proofs is much higher.

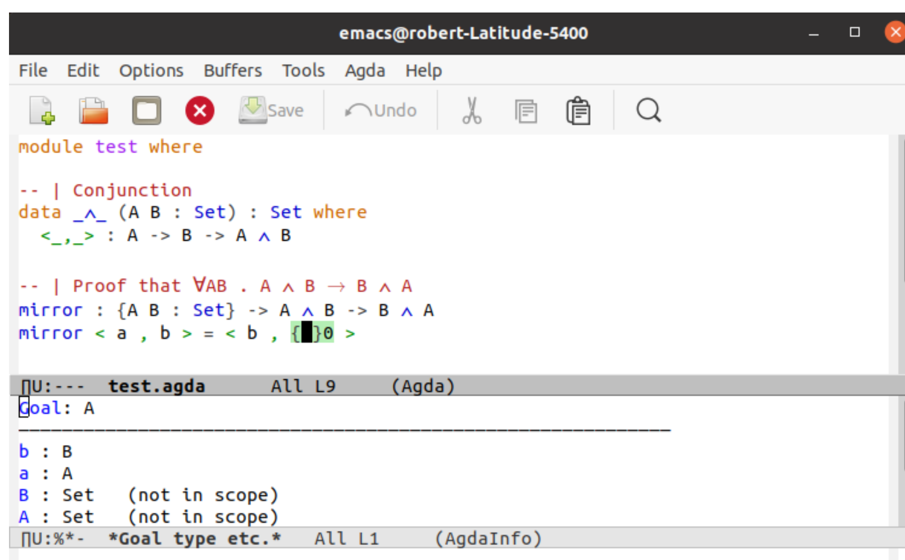

Figure 1.4: Agda has a textual interface, here seen in the EMACS editor. The screenshot shows a hole, the goal and what is in scope. From the types it is clear that `a` is the only thing in scope that could fit into this hole and represent the goal `A`.

1.3.2 Target Audience

GUI-based editors are well suited for people who are in the process of learning natural deduction. Since the user does not need to remember all the rules or be familiar with a language-specific syntax, these editors provide an accessible way of creating proofs. This does not mean that TUI-based editors are not good, but that having to also learn a DSL before being able to actually learn natural deduction costs more time. If time is not a scarce resource, the initial investment of learning to use a proof assistant such as Agda might be more beneficial. The knowledge can later be used to write more extensive proofs.

1.4 Choice of Programming Language

As one of the goals described in section 1.1 is that the editor should not require any installation or setup, the form of a web application is suitable. The term web application is very broad and is applicable to many different types of applications, but in this case, the term refers to a website where everything is executed in a web browser. The website does not need to communicate with a server to fulfil its purpose.

Simple websites can be created with only HTML documents, but for more sophisticated interactive functionality (such as that of a proof editor), scripting support is required. There are many alternatives when it comes to programming languages that provide this.

JavaScript[1] is frequently used to develop web applications [11] and merits consideration. As described in [12], JavaScript is considered the "Language of the Web" but despite its evident popularity, there are some bad parts to JavaScript. The semantics are sometimes not what a developer might expect, so reasoning about what a piece of code means is not always trivial. Furthermore, JavaScript is dynamically typed, meaning that there is no static analysis that guarantees that a program is type-safe and that it will not crash at runtime. There may be a bug in a program that goes undetected until the piece of code that the bug resides in is actually executed.

An alternative to JavaScript is PureScript[2]. PureScript is a strongly typed, purely functional programming language, whose type system and syntax is heavily influenced by the functional language Haskell. A functional language is one that uses the domain of mathematics to describe a computation. A *purely* functional language is one in which side effects are tracked by the type system. Examples of side effects are writing to and reading from files or raising and handling exceptions. In contrast to JavaScript, PureScript is statically typed. After statically determining that a program is type-safe, PureScript code is transpiled to an equivalent JavaScript program that can be run by a browser.

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript`
[2] `https://www.purescript.org/`

Another language common in web development is Elm[3]. Elm is a DSL that was designed specifically for implementing graphical user interfaces for the web [13]. Like PureScript, Elm is purely functional. However, while Elm is a DSL for creating websites and simple web applications, PureScript is designed to be a general-purpose programming language for developing any kind of application. Elm is transpiled to JavaScript code, just like PureScript.

An alternative that is not a functional language is TypeScript[4]. TypeScript is a sequential language like C, C$^\sharp$ and Java, to name a few. A sequential program is defined by a series of statements that are executed in sequence. TypeScript is a superset of JavaScript, which means that any JavaScript program is also a TypeScript program. TypeScript gives the developer the additional ability to annotate a program with type information that is statically verified before execution.

The language chosen for the implementation of this proof editor is PureScript. The choice was motivated by the group's prior knowledge of Haskell, and the more expressive type system which helps ensure correctness.

1.5 Limitations

The editor will provide a certain level of assistance, but it will not offer automated proof construction where a conclusion can be derived automatically from a set of premises. Since the editor targets students learning natural deduction, we believe that the focus should be on the reasoning process rather than providing sample solutions.

The set of inference rules available in the editor and the presentation of the proofs will be taken from the book *Logic in Computer Science* [3], which makes the editor most suited for use by students taking a course that follows this book.

[3] `https://elm-lang.org/`
[4] `https://www.typescriptlang.org/`

2

Theory

The first part of this chapter will give an introductory explanation of propositional and predicate logic, two branches of symbolic logic. Whereas logic is the study of valid rules of inference as a whole, symbolic logic makes use of symbols to capture the logical ideas [14]. Section 2.1 covers the simplest symbolic logic—the propositional logic, with section 2.2 considering the more general predicate logic. After that, in section 2.3, we go over natural deduction which is one way of explaining which logical inferences are valid and why. We remind the reader that we will be using the notation from *Logic in Computer Science* [3]. The remainder of this chapter will then be concerned with the theory behind usability and user experience design, which is used to design usable products.

2.1 Propositional Logic

Propositional logic is a symbolic language that is built by declarative sentences that are either true or false, but not both [15]. These sentences are called *propositions*. Examples of propositions are: "It is raining," "It is sunny," and "Patricia uses her umbrella." These example sentences all make clear statements about something which is either true or false; we can talk about the *truth value* of the proposition. To further strengthen our understanding of what it means to be declarative, let us look at two examples of sentences that are not propositions: The sentence "Go tie your shoes son!" is imperative because it is giving an instruction, while "What time is it?" is interrogative by questioning. No truth values are involved or can be applied to these kind of sentences. For convenience, we shall denote a proposition by a string of lowercase symbols, e.g. we may denote the propositions above as follows:

$$r \triangleq \text{“It is raining,”}$$
$$s \triangleq \text{“It is sunny,”}$$
$$p \triangleq \text{“Patricia uses her umbrella.”}$$

Propositions can either be atomic or compound. *Atomic propositions* are sentences that cannot be further broken down, like r, s or p. *Compound propositions* are sentences that have other sentences joined together in a compositional way using logical operators called *connectives*. The sentences which are joined together can be

either atomic propositions or compound propositions themselves. A simple example of a compound proposition is “It is sunny and Patricia uses her umbrella.” One common set of connectives that are involved in propositional logic, which is the set that we will use, is shown in table 2.1.

Table 2.1: The logical connectives present in propositional logic, with their symbolic representations and example sentences.

Name	Symbol	Example of a declarative sentence
Negation	$\neg r$	It is not raining.
Conjunction	$r \wedge p$	It is raining and Patricia uses her umbrella.
Disjunction	$r \vee s$	It is raining or it is sunny.
Implication	$r \rightarrow p$	If it is raining, then Patricia uses her umbrella.

2.2 Predicate Logic

The statements we have looked at so far have all been simple. Now let us look at a more complex statement such as

$$\text{Every parent is older than some child.} \tag{2.1}$$

Propositional logic will not help in the reasoning here because it is not possible to deconstruct the proposition into elementary propositions composed by logical connectives—through the lens of propositional logic, it must then itself be an elementary proposition. This does however not tell us anything about when this proposition is true, only that it is either true or it is false. We need a way to represent, for example, the relationship “older than” and the fact that it is true for *every* parent. Predicate logic, also known as first-order logic, is an extension of propositional logic where predicates, variables, quantifiers and functions are added to represent more expressive declarative sentences. When doing this, we will talk about objects, living in a specific *domain*.

First, we introduce *terms*. These are either *variables*, *constants*, or *functions* applied to other terms. Variables are used to denote an unknown object in the domain, represented with characters like x. Functions also allow for referring to objects in the domain, by for example $f(x)$, and are used to map objects to each other, such as “the child’s mother”. A *predicate*, on the other hand, is a formula which takes one or more terms and produces a truth value. In other words, a predicate captures the properties of objects living in the domain. Finally, there are also quantifiers in predicate logic. *There exists* ($\exists$) indicates the existence of an object with a certain property, and *for all* ($\forall$) indicates that a certain property applies to all objects in a domain. Using all of the concepts we just introduced, we can convert sentence 2.1 into

$$\forall y\ [P(y) \rightarrow \exists z\ [C(z) \wedge O(y, z)]]$$

where $P(y)$ means y is a parent, $C(z)$ means z is a child and $O(y, z)$ means y is older than z.

2.3 Natural Deduction

One way of constructing proofs in logic is to use a method called *natural deduction*. In natural deduction, one starts with a set of premises and a conclusion to be proven. We denote this as a *sequent*, $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ where ϕ_k represents a premise and ψ is the conclusion we want to prove.

In order to reach the conclusion, we apply inference rules which allow us to prove new formulas from existing formulas. As an example, we have the rule for *implication elimination*, known also as *modus ponens*. Given a statement $\phi \rightarrow \psi$ and the statement ϕ, modus ponens allows you to deduce ψ. This rule is written in the following way

$$\frac{\phi \qquad \phi \rightarrow \psi}{\psi} \rightarrow e$$

Other rules, such as the *implication introduction* rely on assumptions, which are assumed to be valid in a limited scope within the proof, represented by a *box*, and conclusions drawn from these assumptions. Outside of the box, the assumption cannot be used as an argument for inference rules. These boxes can then be used as arguments for various inference rules, such as implication introduction. The rule, written as

$$\frac{\boxed{\begin{array}{c}\phi \\ \vdots \\ \psi\end{array}}}{\phi \rightarrow \psi} \rightarrow i$$

says that given an assumption ϕ and a formula ψ, proven within the scope of ϕ, one can deduce $\phi \rightarrow \psi$.

It is possible to use different sets of inference rules to create a complete set that allows all valid proofs to be proven. In this paper we follow the book *Logic in computer science* [3]; its set of rules is displayed in figure 2.1. It is also possible to derive new rules, such as Modus Tollens (MT), double negation introduction ($\neg\neg e$), Proof By Contradiction (PBC) and the Law of Excluded Middle (LEM). The derived rules used in this project are shown separately in table 2.2. Combining these rules, one can construct both simple proofs, such as the one in figure 2.2, and more complex proofs where we combine several different rules as shown in figure 2.3.

With the introduction of predicates, we need some additional inference rules which are shown in figure 2.4. Apart from formulas and boxes starting with assumptions, these rules also use boxes that start with the introduction of new variables. These variables can be either a *fresh variable*, on which no assumptions are made, or variables introduced along with an assumption. Just like assumptions in propositional logic, the introduced variables are only available within the limits of the box that they open. A box starting with a fresh variable is used in the *for all-introduction* rule,

Introduction rules:

$$\frac{\phi \quad \psi}{\phi \land \psi} \land i$$

$$\frac{\phi}{\phi \lor \psi} \lor i_1 \qquad \frac{\psi}{\phi \lor \psi} \lor i_2$$

$$\frac{\boxed{\begin{array}{c}\phi \\ \vdots \\ \psi\end{array}}}{\phi \to \psi} \to i$$

$$\frac{\boxed{\begin{array}{c}\phi \\ \vdots \\ \bot\end{array}}}{\neg\phi} \neg i$$

Elimination rules:

$$\frac{\phi \land \psi}{\phi} \land e_1 \qquad \frac{\phi \land \psi}{\psi} \land e_2$$

$$\frac{\phi \lor \psi \quad \boxed{\begin{array}{c}\phi \\ \vdots \\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi \\ \vdots \\ \chi\end{array}}}{\chi} \lor e$$

$$\frac{\phi \quad \phi \to \psi}{\psi} \to e$$

$$\frac{\phi \quad \neg\phi}{\bot} \neg e$$

$$\frac{\bot}{\phi} \bot e$$

$$\frac{\neg\neg\phi}{\phi} \neg\neg e$$

Figure 2.1: The proof rules for propositional logic as defined in *Logic in computer science* [3].

Table 2.2: The derived proof rules used in this project, as presented in [3].

Name	**Rule**
Modus tollens	$\frac{\phi \to \psi \quad \neg\psi}{\neg\phi}$ MT
Double negation intro	$\frac{\phi}{\neg\neg\phi} \neg\neg i$
Proof by contradiction	$\frac{\boxed{\begin{array}{c}\neg\phi \\ \vdots \\ \bot\end{array}}}{\phi}$ PBC
Law of excluded middle	$\frac{}{\phi \lor \neg\phi}$ LEM

1.	R	premise
2.	$R \to U$	premise
3.	U	$\to$ e, 1, 2

Figure 2.2: A simple proof in natural deduction, proving the sequent $R, R \to U \vdash U$.

1.	$A \land B \to C$	premise
2.	$A \lor D$	premise
3.	$B \lor E$	premise
4.	$\neg(D \lor E)$	premise
5.	A	assumption
6.	D	assumption
7.	$D \lor E$	$\lor i_1$, 6
8.	$\bot$	$\neg$e, 4, 7
9.	A	$\bot$e, 8
10.	A	$\lor$e, 2, 5–5, 6–9
11.	B	assumption
12.	E	assumption
13.	$D \lor E$	$\lor i_2$, 12
14.	$\bot$	$\neg$e, 4, 13
15.	B	$\bot$e, 14
16.	B	$\lor$e, 3, 11–11, 12–15
17.	$A \land B$	$\land$i, 10,16
18.	C	$\to$ e, 1, 17

Figure 2.3: Proof of the sequent $A \land B \to C, A \lor D, B \lor E, \neg(D \lor E) \vdash C$. Note the use of boxes which make up the arguments for $\lor$-elimination on rows 10 and 16.

Introduction rules:	Elimination rules:
$$\dfrac{\boxed{\begin{array}{l} x_0 \\ \quad\vdots \\ \quad \phi\,[x_0/x] \end{array}}}{\forall x \phi}\ \forall x \text{ i}$$	$$\dfrac{\forall x \phi}{\phi\,[t/x]}\ \forall x \text{ e}$$
$$\dfrac{\phi\,[t/x]}{\exists x \phi}\ \exists x \text{ i}$$	$$\dfrac{\exists x \phi \quad \boxed{\begin{array}{ll} x_0 & \phi\,[x_0/x] \\ & \vdots \\ & \chi \end{array}}}{\chi}\ \exists x \text{ e}$$
$$\dfrac{}{t = t}\ = i$$	$$\dfrac{t_1 = t_2 \qquad \phi[t_1/x]}{\phi[t_2/x]}\ = e$$

Figure 2.4: Additional inference rules for predicate logic.

which says that if we introduce a variable x_0 and deduce $\phi\,[x_0/x]$, within the scope of x_0, then we can use this box to prove $\forall x\, \phi$. Here the notation $\phi\,[x_0/x]$ means ϕ with all occurrences of x substituted with x_0. The $\forall$-introduction rule is written as

$$\dfrac{\boxed{\begin{array}{l} x_0 \\ \quad\vdots \\ \quad \phi\,[x_0/x] \end{array}}}{\forall x \phi}\ \forall x \text{ i}$$

A variable introduced along with an assumption allows us to use *exists-elimination* which says that given a formula $\exists x\, \phi$, a variable x_0, introduced with the assumption $\phi[x_0/x]$, and a conclusion χ, proven within the scope of x_0, we can deduce χ. The variable x_0 may not occur in χ, nor outside the box, which ensures that this rule is sound. Symbolically, it is written in the following way

$$\dfrac{\exists x \phi \quad \boxed{\begin{array}{ll} x_0 & \phi\,[x_0/x] \\ & \vdots \\ & \chi \end{array}}}{\chi}\ \exists x \text{ e}$$

Using the rules listed in table 2.4 in combination with the rules from table 2.1, we can construct proofs for sequents which contain predicates, such as $\forall x\,(P(x) \to Q(x)), \forall x\, P(x) \vdash \forall x\, Q(x)$, as shown in figure 2.5.

1.	$\forall x\,(P(x) \to Q(x))$	premise
2.	$\forall x\, P(x)$	premise
3.	x_0	fresh
4.	$P(x_0)$	$\forall$e, 2
5.	$P(x_0) \to Q(x_0)$	$\forall$e, 1
6.	$Q(x_0)$	$\to$ e, 4, 5
7.	$\forall x\, Q(x)$	$\forall$i, 3–6

Figure 2.5: Proof of the sequent $\forall x\,(P(x) \to Q(x)), \forall x\, P(x) \vdash \forall x\, Q(x)$.

2.3.1 Automated Natural Deduction

For the purpose of providing hints to the user, we implement the algorithm for automated natural deduction by Bolotov, Bocharov, Gorchakov, *et al.* [2]. We give a brief overview here but refer to the original article for details. For simplicity, it uses the minimal set of inference rules, which is different from what is described above. The *goal-directed* searching procedure for a proof proceeds by at each iteration first checking the reachability of the current goal, which can be either a contradiction or some formula. If reached, a corresponding introduction rule is applied. Otherwise, the algorithm searches for an elimination rule to apply. If no such possible elimination rule is found, however, then the structure of the current goal is considered in order to push new goals and potentially add new assumptions. Backtracking is used to explore branching paths, e.g. if the current goal is $a \vee b$ we try to prove either of a, b and $\neg(a \vee b)$. To prevent looping a marking scheme is used. Finally, if at some point we are searching for a contradiction and no more elimination rules are applicable then a new goal gets generated such as to make the exhausted derivation form a so-called *Hintikka set*, which we know would be satisfiable (and so the goal would be unreachable).

2.4 Usability and User Experience

General guidelines exist that can be followed and applied to implement better products and one common variant of this is usability design. The term usability is a qualitative attribute that focuses mainly on how usable a product is, referring to that it is easy to learn, can be used effectively, and that it is enjoyable from a user experience perspective [16]. Objective reasoning about the usability of a product may be done in different ways depending on the product in question or its user. Does the program need to be highly productive at work or is it a learning tool for students that should be motivating and helpful at the same time? Removing any subjective reasonings like individual preferences about the usability of a product, there are six main goals [16] that every interactive product should have:

Effectiveness refers to if a product is doing what it is supposed to do and whether the user can perform expected actions and complete them.

Efficiency relates to speed and whether the product helps users perform actions quickly and sustain a higher level of productivity.

Safety can be seen as how tolerant a product is regarding errors that may arise. Users should be safe from performing unwanted actions and if they accidentally perform such an action, safety mechanisms should be in place for error recovery.

Utility focuses on whether the product has the necessary and correct functionality that is needed when using the product.

Learnability refers to how easily new users can learn to use the product.

Memorability is related to the learnability goal. When a user has learned how to use a product, how easy is it to remember how to use it?

These goals are a good place to start from when reasoning about what functionality and features are needed in a product. The usability goals can also help formulate questions to be used as self-checks by developers or in user testing to assess how well a product works.

Another set of goals that is helpful to identify and specify in order to design a good product is user experience goals. While the usability goals are more divisible and in some sense more objective, the user experience goals are more about the holistic experience and the subjective feelings and reactions of a user.

The International Organization for Standardization (ISO) defines user experience as a user's perceptions and responses that result from the use and/or anticipated use of a system, product or service. ISO defines usability as the extent to which a system, product or service can be used by specified users to achieve its specified goals with effectiveness, efficiency and satisfaction in a specified context of use [17].

From these definitions, it can be reasoned that usability deals with users as a group while user experience deals with users as individuals. To design for usability one has to understand the intended audience as a group, identifying their common needs. To design for user experience one has to try to identify the reactions and emotions people in the target group have in common, and what elements contribute to evoking certain feelings for a user. Elements that affect the user experience include attention, pace, play, interactivity, conscious and unconscious control, style of narrative, and flow [16].

3

Methods

The following chapter gives an overview of the development and evaluation of Logan. The actual implementation is presented in detail in chapter 4.

3.1 Developing Logan

The development of Logan was done in an entirely remote setting, with the team meeting twice per week over video chat. The development of the code was broken into tasks that were coordinated using a Kanban board, where each task was given a card. These cards were then sorted into different columns for upcoming, ongoing and finished tasks, giving the team an overview of the progress. Code modules were continuously submitted for peer-reviewing within the group before being accepted into the main branch of the project repository.

3.2 Evaluating Logan

To get insights into how well the proof editor performs in terms of usability and user experience, user tests were employed.

The participants of the user study were students who had previously taken the course *Logic in Computer Science* at University of Gothenburg or Chalmers University of Technology, with various backgrounds from programs in Computer Science. All the students had some prior experience of using a terminal in their studies, but only one of them had used a proof editor for logic before, and they stated that the extent of their usage had been limited. Three students had constructed proofs in natural deduction within the last year, while the rest had done so within the last two years.

The study consisted of five participants since Nielsen and Landauer have shown that the proportion of identified usability problems in a usability test with five participants is likely to be over 75 percent [18]. While the number of new problems identified initially increases rapidly, after testing five participants the likelihood of identifying additional issues decreases significantly.

The tests were constructed using the *discount usability engineering method*. This method was invented by Jakob Nielsen in the early 90s [19] and emphasizes the use

of informal basic usability testing methodologies that are cheap, fast and easy to perform. Many such methodologies exist, and in this study the *think-aloud protocol* was used. The think-aloud protocol is a method where participants are asked to verbalize their thoughts and feelings while performing some task. Collecting this additional data in the form of a participant's thoughts and emotions contributes to understanding their cognitive process. The ambition was to use this extra information to understand *why* some design decisions were good or bad.

The study itself was qualitative rather than quantitative. A qualitative study emphasizes careful test design to gather as much information as possible from each test, while a quantitative study has participants perform more tests with less data collected from each test. Both methods have their advantages, but a quantitative study is more time consuming and requires a more careful selection of test users, to make sure they are representative of the target audience. Data gathering methods used in this study were making observations during the tests and performing interviews with open-ended questions. To gather data the participants were observed during the test and interviewed with open-ended questions afterwards. Open-ended questions let the participants explore their answers themselves, which often results in more data gathered. As the tests were scheduled to be done in the middle of the COVID-19 pandemic, the tests were performed remotely. The advantages of this format are that the testers use their own equipment and are working in their own environment to test the proof editor, just like they would if they were using it to learn natural deduction [16].

Each test was performed by two moderators, whose task was to guide the tester through the user test. The participants shared their screens and spoke with the moderators during the test. The tests were recorded to let the moderators spend their attention on observing the test users rather than on taking notes.

Each test consisted of the following steps: First, a participant was asked some simple general questions that are relevant to the study. This initial discussion was meant to ease the tension of the tester and let the moderators get a deeper understanding of the participant. After this, the participant got a link to a website that hosted the editor. The moderators provided simple instructions, intended to assess the basic functionality of the editor, such as “Can you add a premise?”. The participant then completed the task while the moderators observed their behavior with the participant executing the think-aloud protocol. Lastly, the participants were sent a file containing natural deduction proofs, illustrated in figures 3.1, 3.2, 3.3 and 3.4. Some proofs are correct while others contain errors that the participants were meant to identify and fix using the proof editor. The participants were given finished problems to solve with Logan as the study intended to test the usability of the proof editor, not the natural deduction competence of the participants.

At the end of the tests, the participants answered questions related to the user experience of using the editor to get additional data that could have been missed by the moderators. The observation focused on identifying potential issues related to the main usability and user experience goals. The functionality of the editor was tested

throughout the observation to see if something was missing or was injudicious, which corresponds and is related to *effectiveness* and *utility*. The moderators checked if some functionality was not easily grasped, missed or could be improved when seeing it from a user perspective. *Learnability* and *memorability* was tested by identifying if the basic functionality was easily learned by the users and if they could use what they had learned to navigate and write proofs in the editor for the first time. The users were exposed to functionality during the early part of the testing which is needed in later parts when writing down the proofs listed in figures 3.1, 3.2, 3.3 and 3.4. The intention was to identify if the basic functionality is easily grasped in the earlier stage, and at the same time test memorability when the test user will need to use the knowledge gathered in the earlier part of the testing process. Lastly, *efficiency* was tested throughout the process by studying if the users were learning to navigate and construct proofs in the editor in a reasonable time. No specific time limits were set, it was up to the moderators to judge from their experience what constituted a reasonable time.

The last part that contained an interview with open-ended questions, was used to gather data relating to the user experience of the tester and the interface design of the editor. The intention during this part was to strengthen and clarify the observed feelings and emotional aspects of the user during the testing process. Moderators might miss details or be biased in how they perceive an observed scenario and thus, to minimize inaccurate conclusions, this part was considered necessary.

After the data was gathered, it was analyzed by the moderators and used to evaluate the usability and user experience of Logan. Usability issues were presented to the project members, where a discussion was held, triaging what issues needed to be improved upon. Potential issues were ordered by severity, where later fixes focused on the most severe ones. The details of the particular solutions for how an individual issue was remedied can be found in subsection 4.2.4.

1.	$P \land Q$	premise
2.	R	premise
3.	Q	$\land e_1$ 1
4.	$Q \land R$	$\land i$ 2, 3

Figure 3.1: Test case 1: $P \land Q, R \vdash Q \land R$. It is indeed possible to construct a proof of this using natural deduction, but the specific proof outlined here is not correct. The rule that is applied on the third line should be $\land e_2$, and the operands to $\land i$ on line 4 should be flipped.

1.	$P \to (Q \to R)$	premise
2.	$P \land Q$	assumption
3.	P	$\land e_1$ 2
4.	Q	$\land e_2$ 2
5.	$Q \to R$	$\to$ e 1, 3
6.	R	$\to$ e 5, 4
7.	$P \land Q \to R$	$\to$ i

Figure 3.2: Test case 2: $P \to (Q \to R) \vdash P \land Q \to R$. This is provable using natural deduction, but there are some minor details that are wrong in this particular proof. On line 7 the introduction rule is applied to no arguments at all, but it should be applied to the entire subproof spanning lines 2-6. Furthermore, when going from line 6 to 7 the participants must close the proof box. By default new rows are added in the current proof box, but by pressing the `Shift` and `Enter` keys simultaneously the new row will be inserted after closing the current proof box.

1.	$Q \to R$	premise
2.	$P \lor Q$	assumption
3.	P	assumption
4.	$P \lor R$	$\lor i_1$ 3
5.	Q	assumption
6.	R	$\to$ e 1, 5
7.	$P \lor R$	$\lor i_2$ 6
8.	$P \lor R$	$\lor$e 2, 3–4, 5–7
9.	$P \lor Q \to P \lor R$	$\to$ i 2–8

Figure 3.3: Test case 3: $Q \to R \vdash P \lor Q \to P \lor R$. This proof contains no errors, but it will test the participants intuition when it comes to using nested proof boxes. They are expected to use some rules they have not used in the previous tests.

1.	$P \land \neg Q \to\to R$	premise
2.	$\neg R$	premise
3.	P	premise
4.	$\neg Q$	assumption
5.	$P \land \neg Q$	$\land$i 3 , 4
6.	R	$\to$ e 1, 5
7.	$\bot$	$\neg$e 6, 2
8.	$\neg\neg Q$	$\neg$i 4-7
9.	Q	$\neg\neg$e 88

Figure 3.4: Test case 4: $P \land \neg Q \to R, \neg R, P \vdash Q$. The errors in this proof are that there is a redundant implication arrow on the first line, and on line 9 the rule is applied to line 88 instead of line 8. The proof editor will always open a new box when an assumption is introduced, but in this proof there is no box. The proof editor will open the box but the participants will not have closed it if they simply entered the proof as it is written here.

4
Results

In this chapter, we give a description of Logan and present the results from our user tests. The entire source code for Logan is publicly available in a GitHub repository[1], with a live version hosted on GitHub Pages[2].

4.1 Application

As mentioned in section 1.5 this project aims to develop a proof editor to be used together with the book *Logic in Computer Science* [3] and will use the set of inference rules presented in tables 2.1, 2.2 and 2.4.

4.1.1 Representation of a Proof

Internally, inference rules are represented as an algebraic data type with separate data constructors for each rule. Each constructor takes a number of arguments, enclosed in the `Maybe` type, corresponding to the number of arguments the rule can be applied on. Individual formulas are represented by the line number on which they occur, while boxes are represented as a two-tuple with the numbers of the first and last line of the box. Figure 4.1 shows the `Box` type and a few of the constructors of the `Rule` type.

A `Proof` is represented as an `Array` of `ProofRow` where each row contains a formula or a fresh variable, along with the justifying inference rule and an error which is present if the rule application fails for any reason. There is also a `List` of `Scope` which implements a stack of scopes to keep track of available rows and boxes with each scope corresponding to a box in the proof. The code for these types is displayed in figure 4.2. In order to avoid having to store variable contexts, shadowing by way of a fresh variable is disallowed. For a pedagogical tool, this has the benefit of avoiding generated error messages such as 'Expected x_0, got x_0,' where one x_0 shadowed the other.

Formulas are represented using the recursive data type `Formula`, with propositions being represented as predicates with no terms, as shown in 4.3. The type `FFC` (Formu-

[1] `https://github.com/datx02-21-16/datx02`
[2] `https://datx02-21-16.github.io/datx02/`

```
type Box = Tuple Int Int

data Rule
  = Premise
  | Assumption
  | AndElim1 (Maybe Int)
  ...
  | OrElim (Maybe Int) (Maybe Box) (Maybe Box)
  ...
  | ExistsElim (Maybe Int) (Maybe Box)
  ...
```

Figure 4.1: Definition of the `Rule` type and a few of its constructors.

```
type ProofRow
  = { formula :: Maybe FFC
    , rule :: Maybe Rule
    , error :: Maybe NdError
    }

type Scope
  = { lines :: Set Int
    , boxes :: Array Box
    , boxStart :: Maybe Int
    , vars :: Map Variable Int
    }

type Proof
  = { rows :: Array ProofRow
    , scopes :: List Scope
    }

newtype ND a
  = ND (State Proof a)

runND :: forall a. Maybe FFC -> ND a -> Tuple Boolean Proof
runND conclusion (ND nd) = ...
```

Figure 4.2: Definition of the `ProofRow`, `Scope`, `Proof` and `ND` along with the type signature for the function `runND` which performs the actual validation.

```
newtype Variable
  = Variable String

data Term
  = Var Variable
  | App String (Array Term)

data Formula
  = Predicate String (Array Term)
  | Not Formula
  | And Formula Formula
  | Or Formula Formula
  | Implies Formula Formula
  | Forall Variable Formula
  | Exists Variable Formula

data FFC
  = FC Formula
  | VC Variable
```

Figure 4.3: Definitions for the `Variable`, `Term`, `Formula` and `FFC` types used to represent different parts of a formula.

laFieldContent) with constructors `FC` (FormulaContent) and `VC` (VariableContent), which is used to allow the introduction of fresh variables.

For substitutions and unification, we implement the definitions and routines from [15]. In our implementation, the data type `Variable` stores only the textual representation of variables, instead of making use of interning or generation numbers to influence equality. While conceptually simpler, it does have some drawbacks: For example, say that we have the two quantified formulas $\exists x\, P(x, z)$ and $\exists y\, P(y, x)$ and we wish to find a necessary substitution $\{z/x\}$ to make them equivalent. Currently, we first need to find a unique variable u, by concatenating all involved variables, before trying to unify the set $\{P(x, z), (P(y, x)\{u/x\})\{x/y\}\}$. This is necessary because the bound and unbound x variables would otherwise be considered equal.

4.1.2 Validation of Proofs

To validate the proofs, each line is added into the state of an `ND` (NaturalDeduction), the definition of which can be seen in figure 4.2. The complete proof is then validated using the `runND` function which uses `runState` to check for any errors and compare the formula on the last row to the entered `conclusion`.

When adding a `ProofRow` to a proof, the row is validated using the `applyRule` function, part of which is shown in figure 4.4. `applyRule` validates a rule application by performing the following steps:

1. Use the functions `proofRef` and `boxRef` to fetch the formulas from any rows or boxes being referenced by the `rule`.
2. Pattern match the referenced formulas against the formulas that the rule can be applied on.
3. Compare the result of applying the `rule` on the argument with the input `formula`.

If any of the steps fail, the function will throw an appropriate error that gets stored in the row, causing `runND` to return `false` when validating the proof.

```
proofRef :: Maybe Int -> ExceptT NdError ND FFC

boxRef :: Maybe Box -> ExceptT NdError ND (Tuple FFC FFC)

applyRule :: Rule -> Maybe FFC -> ExceptT NdError ND FFC
applyRule rule formula = do
  { rows, scopes } <- get
  case rule of
    Assumption -> except $ note BadFormula formula
    AndElim1 i -> do
      a <- proofRef i
      case a of
        FC (And x _) -> pure $ FC x
        _ -> throwError $ InvalidArg BadAndE
    ImplIntro box -> do
      (Tuple a b) <- boxRef box
      case a, b of
        FC f1, FC f2 -> pure $ FC $ Implies f1 f2
        _, _ -> throwError $ InvalidArg ArgNotFormula
    ...
```

Figure 4.4: Type signatures for the `proofRef` and `boxRef` functions along with parts of the definition of the `applyRule` function.

Once the validation is finished, `runND` returns a tuple with a `Boolean` indicator of whether the `conclusion` has been proven.

4.1.3 Implementation of the Web Application

The user interface in Logan is implemented as a web application in Halogen [20], which is a component-based framework, written entirely in PureScript. The choice of using Halogen was influenced by the choice to use PureScript as the programming language for the logic engine. Using a different language to implement the graphical interface would have meant that the two sides of the application, the website and the logic engine, would need some way of communicating. If the graphical interface is written in PureScript it is enough to use normal function application to invoke

the logic engine.

In Halogen, each DOM element is represented as a PureScript function. Static elements can be represented with a single rendering function, while more complex components, requiring dynamic behavior, are implemented as their own `Component` type. In addition to the rendering, the components let the developer specify state, accepted data input to the component, internal actions in the component and queries it can respond to. In components the rendering function states the appearance of the component in a declarative manner. It gets called whenever the state changes, causing Halogen to carry out the necessary modifications to the DOM. The actual interface is put together by nesting elements and components inside each other, just like a static HTML page in which all elements are placed in a tree structure.

The main component in the Logan interface is the proof component which is rendered as the main panel in the interface. This component maintains a state with the target conclusion and all rows of the proof, each of which in turn contains text fields for rules and formulas, which are themselves components. Whenever the user changes the input, an `ND` is built from the new state and validated via the `runND` function as described in subsection 4.1.1. The returned values are then used to print error messages or show the success indicator if the proof is complete.

4.1.4 The Final Product

Logan can be used to construct proofs as shown in figure 4.5. Once the conclusion has been reached, a green frame is shown to indicate that the proof is finished. Boxes are delimited by a black frame from the initial assumption to just below the final line.

If the user makes a mistake, such as feeding an invalid argument to a rule, the editor will highlight the fields where the error occurs and display a message, explaining what the error is. Error messages can be shown for the following reasons

- A line reference or box reference cannot be parsed.
- The user references something which is out of scope.
- A formula or rule cannot be parsed.
- The user makes a reference to a line where a box is expected or vice versa.
- The user tries to refer to a box where the referenced lines are not the borders of the same box.
- A formula does not match the output from the rule application on the same line.
- A user tries to apply an inference rule on a which does not match the expected pattern.

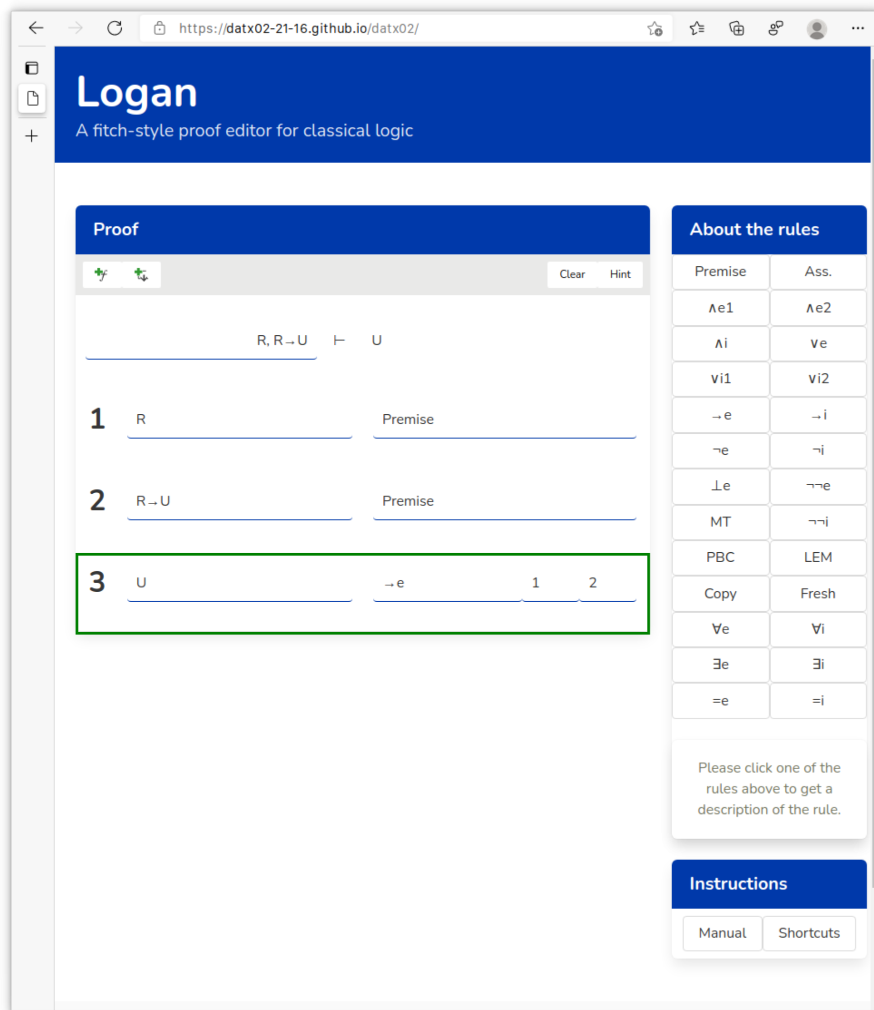

Figure 4.5: Screenshot of the editor, showing a proof of the sequent $R, R \to U \vdash U$.

- A new premise is introduced in the middle of a proof.

As long as there are errors in the proof, the editor will not show the green border around the conclusion, even if it has been reached. Examples of error messages can be found in figures 4.6 and 4.7.

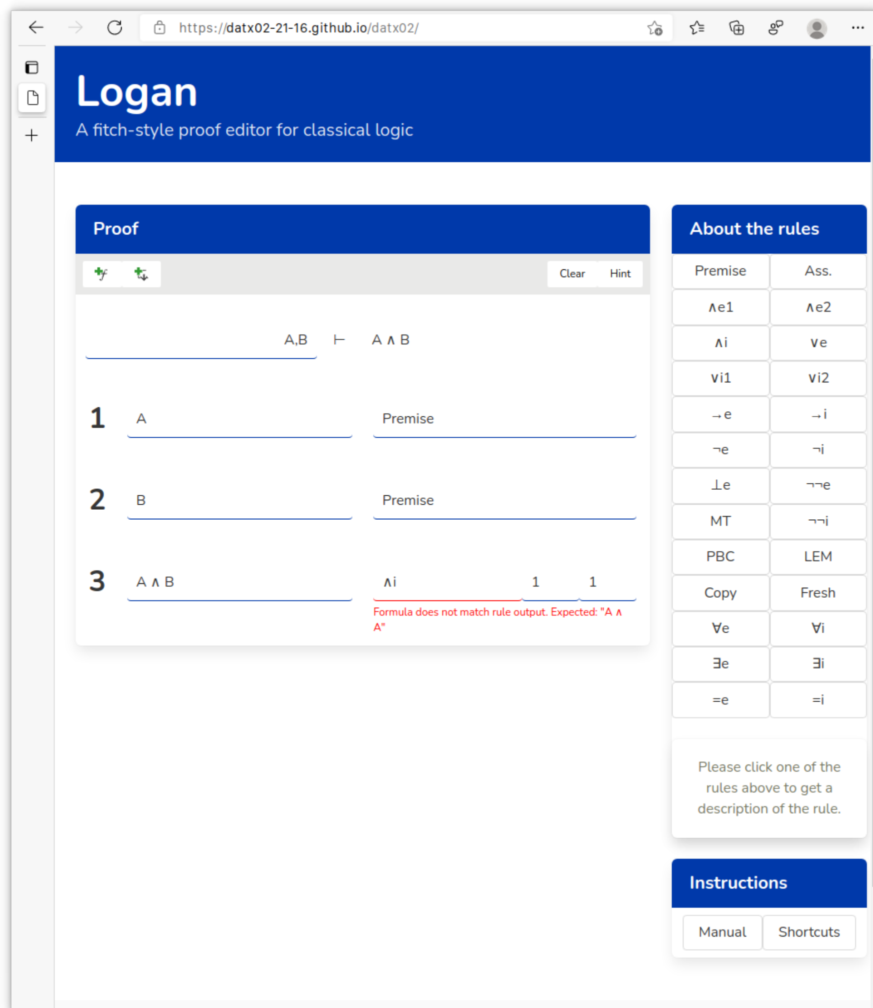

Figure 4.6: Screenshot of Logan with an error message. The output from ∧i does not match the formula on line 3.

The editor can also explain the available rules. By clicking the buttons to the right of the proof, the user can see the syntax of each rule and get a brief explanation, as shown in figure 4.7. By clicking the 'Manual' and 'Shortcut' buttons, the user can get more detailed instructions and see a list of available syntax shortcuts, such as writing 'im' to get an implication arrow.

One noteworthy thing is that boxes in the proof were chosen to be tied to Assumption- or Fresh-rule applications, which is different from how e.g. Conan does it, where boxes are a separate thing that the user can press a button to create anywhere in the proof. Our behavior can be argued to be more streamlined since an assumption

Figure 4.7: Screenshot of Logan, showing an explanation for the →e rule, which the user has already tried to use. Since there is an error present in the proof, the last line is not highlighted in green, even though it matches the conclusion.

cannot go anywhere except at the start of a box. Thus, having the box be created automatically with the assumption detracts less from the user's flow.

In order to be able to give the user rudimentary hints on how to get started on a proof, in case they ever get stuck, we implemented an algorithm for automated natural deduction, as described in section 2.3.1. As described the algorithm can prove sequents in predicate logic, but we have only implemented it for propositional logic. For simplicity, when generating hints the editor only looks at the user-inputted premises and conclusion, and not any of the proof rows that the user may have already filled in. The proof editor will then solve the sequent itself behind the scenes, reconstruct the structure of boxes and tell the user what assumptions were made up to the first PBC. This was motivated by the rules requiring assuming arbitrary formulae - i.e. not necessarily subformulas of previously proved rows - being deemed the hardest given the infinite search space. The path to a proof requiring only, say, the $\wedge/\rightarrow$-rules, on the other hand, is easier because each step may be done by only enumerating what rules can possibly be applied given the formulae that have already been proved.

4.2 User Tests

User tests were performed a few weeks before the end of the project, which meant that some functionality was not yet in place. The interface at the time of testing is presented in figure 4.8. There are no rules for predicate logic included in this interface because at this time one could only use Logan to construct proofs in propositional logic.

4.2.1 Basic Functionality

In this section, we first present an overview of how basic functionality in Logan worked for the users and then go into detail about the users' experience while performing specific tasks.

The basic functionality of the editor as defined by the tasks in table 4.1 overall worked fairly well for the users. There was no task that was impossible for the users to accomplish, even though certain difficulties were encountered by some. All users found the manual and shortcuts popups when clicking the appropriate buttons, but which one they used the most differed from user to user. The users all turned to the manual or shortcuts once they had encountered problems, not before. When using the manual or shortcuts they did not read it through but opted to scroll through it, scanning for relevant information to the problem at hand or search using the browser shortcut `Ctrl + f`.

One difficulty that occurred and affected the users during this part of the testing was related to the buttons and rule field, see figure 4.9. The buttons seemed to confuse and lead the users in the wrong direction when they started to explore the editor. Many users thought that these buttons would have some functionality relating to

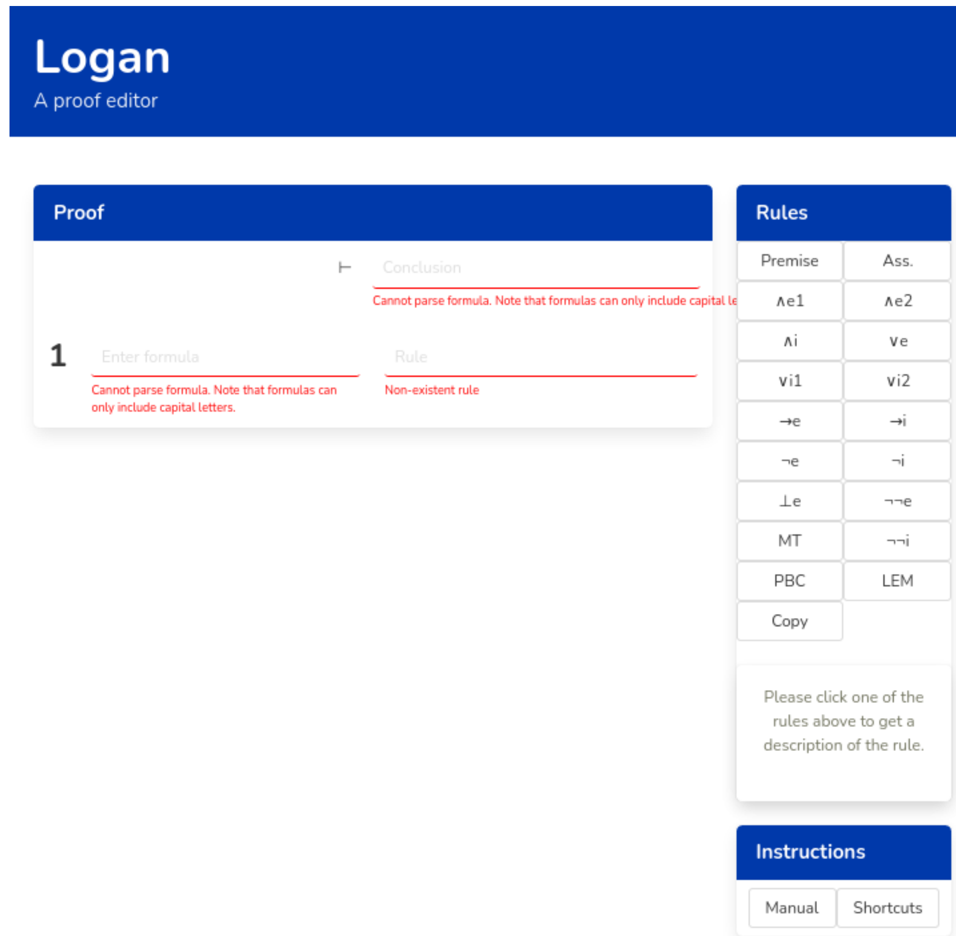

Figure 4.8: The interface of Logan at the time of user tests.

Table 4.1: Instructions for testing basic functionality.

Task	Description
1	Can you add a big capital letter A as a conclusion?
2	Can you add a big capital letter B as a premise?
3	Can you add a new empty row?
4	Can you remove a row?
5	Can you add an assumption with the formula C?
6	Can you remove the assumption with the formula C?
7	Can you change the conclusion to $A \wedge B$?
8	Remove all the numbered rows if you haven't done it yet and then write two premises. One as the letter A, the other as letter B.
9	Can you apply the and introduction rule on the two premises A and B?

applying or inputting rules in a proof. They clicked the buttons multiple times and seemed to not understand what happened when they clicked or what the intended purpose of the buttons was.

Figure 4.9: The rule field of the editor with buttons for each rule. When a button is clicked a help text about the rule replaces the text below the buttons.

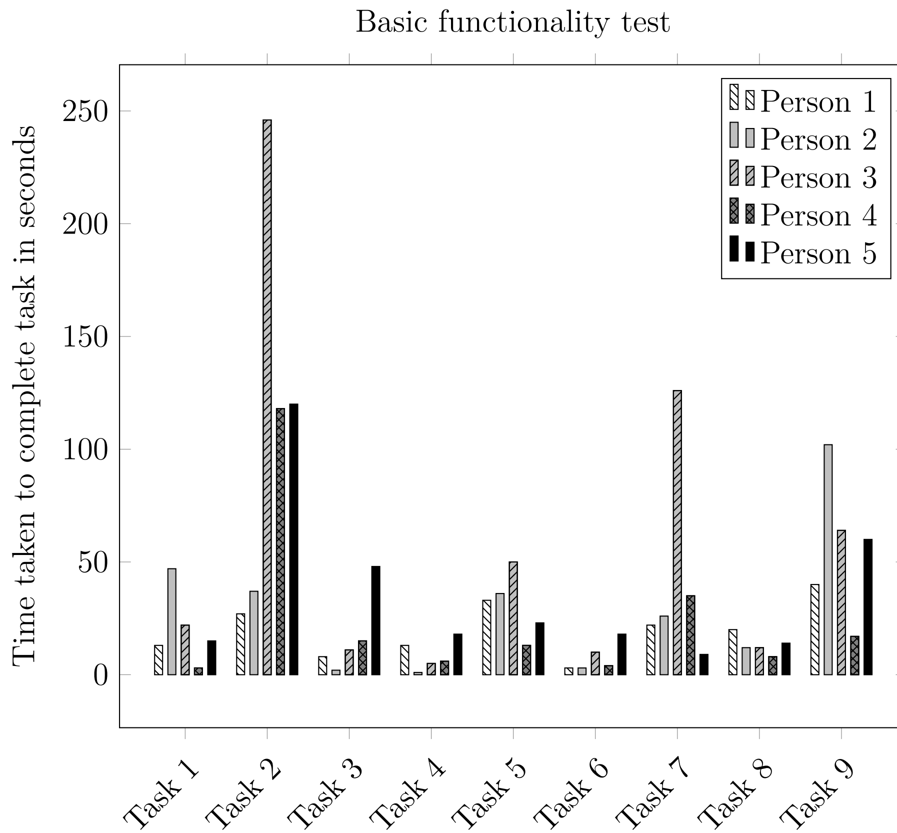

Figure 4.10: Time in seconds taken to complete each task in observation 1. Task 2, adding a premise, took a long time for most users.

Most users experienced some difficulties with adding a premise, which is reflected in the time it took them to complete this task, see figure 4.10. They tried to click on the 'Premise' button in the rule field, expecting that this would enable them to add a premise somehow. Some also searched for a way to add a premise by clicking in the area to the left of the turnstile symbol. In the end, all but one user had to use the manual in order to find out how a premise could be added to the proof.

The shortcuts used to insert a new row and delete a row seemed to be intuitive to use for the users. These actions were done reflexively by most and the users reported that the actions felt natural to them. Two users accessed the shortcuts popup to find out about these commands without trying anything on their own first. It seemed like they were trying to find a button to press before looking it up. The other users all guessed and got it right by themselves. However, they did not all get it right on the first try. Some tried to drag the row away to remove it or press the `Delete` button on their keyboard before trying the correct command.

While adding and deleting assumptions was not too difficult for the users to accomplish, some users still tried pushing the 'Ass.' button expecting to be able to use it to add an assumption. Most users noticed that you can copy the label of the button but some missed the period in 'Ass.' and did not immediately understand why this did not work.

Changing the conclusion to one which included the symbol ∧, was completed reasonably fast by all users except one. All users intuitively understood that the conclusion field can be clicked on to edit it. Most users tried typing in 'and' and got pleasantly surprised by the capabilities of the proof editor to parse the word and output the correct symbol. Some users consulted the manual or shortcut popups to find the correct keyword for inputting the symbol. User 3, who took a longer time than the others, seemed to misunderstand the task and wasted time on adding new rows in the proof and trying to apply rules to get to the conclusion until they realized that the task was simply to update the conclusion in the conclusion field.

Even in the last task of the test, some users still kept clicking buttons when they were trying to apply the ∧i rule. One user looked in the shortcut popup for information on how to write a rule only to find keywords for generating specific symbols and nothing about the rules, which added a bit of confusion. Others understood quickly that they could use the knowledge of generating the ∧ symbol they had used in an earlier task and append the letter 'i' to write the corresponding rule.

It was observed throughout the testing that there were some differences between how the users chose to interact with the interface, using either the mouse or keyboard. Some always used the mouse to move between fields while typing in the formulas and rules, while others used `Tab` to do this. A few users seemed to look for buttons as their first instinct when trying to figure out how to accomplish a task. These users were the same ones that used the mouse to navigate.

4.2.2 Constructing Proofs

After having gone through the basic functionality testing, users did not seem to experience further difficulties when performing the second part of constructing natural deduction proofs of the given sequents. No task was impossible to complete and clear improvements were observed when constructing these proofs using the knowledge gathered in the previous part. The users started to navigate freely, understanding more and more how to use the editor and apply inference rules. During the construction of proofs, the users had to use symbols and inference rules they had not encountered in the basic functionality testing. This presented no problem, however, with users either guessing the commands or quickly looking them up in the manual or shortcut popups.

The commands to input symbols seemed easy for the users to remember. Most users only needed to look them up at most once in order to be able to use them in their proofs. Some symbols had multiple commands available, and there were some variations in which one the user picked. For example, some users used 'neg' as a shortcut to write the ¬ symbol while others used 'not' instead.

At the time of testing, there were some inconsistencies in how many letters were used in each command. Some symbols had commands consisting of two letters while others used three letters. One user said that this made it more difficult to remember the commands since they did not follow a common pattern. This user sometimes lost

flow when writing due to expecting a two-letter command to work when a three-letter command was needed.

The feedback that the editor gives in the form of error messages and color-coding seemed to work well for the users. When a user made a mistake the editor underlined a part of the row in red to show the error and displayed a message and the users reacted to this and fixed the mistakes. However, it seemed that users mostly noticed the red color and did not pay much attention to the error messages. One user remarked on the fact that the editor underlined the rule when the formula and the output from applying the rule to its argument did not match. They felt like this was leading them to believe something was wrong with the rule when in fact the rule was not really the problem.

Test case 4, see figure 3.4, introduced a problematic situation. The assumption row would introduce a new box which would have users keep adding lines inside that box until the end and then have to go back and try to amend this "mistake". As was suspected, this proved somewhat complicated for the users to accomplish. This test was designed specifically to explore the possible difficulties users could encounter due to the fact that the Shift + Enter command, which at this point only worked to close a box when at the last row of the box. If one attempted to close a box with `Shift + Enter` while in the middle of the box a new row inside the box appeared instead. Few users discovered the drag-and-drop functionality that would enable them to drag rows out of the box to amend the mistake easily. Most ended up closing the box at the last inputted row, then rewrote the last rows of the box outside of the box before deleting those rows from inside the box. This way of solving their mistake seemed to distract from the users' flow and annoy them slightly. Many users tried to use the `Shift + Enter` command in the middle of the box multiple times before resorting to this more roundabout fix. The results of this test gave a clear indication that this was a usability issue that needed to be addressed.

4.2.3 Interviews

During the interviews, the users were asked to rate the visual design on a scale from one to five, where one represents the least appealing and five the most. An overview of the ratings can be found in figure 4.11. One user rated the design as a five, with the motivation that they had no problems with it - it was clean, clearly divided and had no clutter. The other users rated the design between three and four but gave similar comments as the user that gave the design a top grade. Something that lowered their rating according to some users was the error messages that were displayed from the start. The issues with these were that the message did not display properly, but also that the users questioned why they were there at all. They reasoned that when they first start using the editor they have not made any mistakes yet and they are already aware that they need to fill in the fields in order to build a proof and thus do not need to be alerted of empty fields. One user thought the placeholder text was impossible to read, due to being a light grey on a white background. Another one said that a dark mode would have been appreciated to feel more comfortable while using the editor.

On the same scale, users were asked to rate the ease of finding necessary information in order to be able to navigate and use the proof editor. These ratings can be found in figure 4.11. Here all the users tested agreed that they could find the necessary information but that it could be made even easier. Their ratings were all between three and four. Some thought the manual and shortcuts had too much information in one place, and that searching for the relevant information could be made easier by introducing some navigational tools or separating out some information to another place. Another suggestion was to split the shortcuts popup into two, one with shortcuts for different actions and one with how to type the symbols. One user also said that they would prefer if a popup with just the symbols could be accessed with a command such as `Ctrl + h` to avoid having to use the mouse while typing the proofs.

During the tests, some difficulties the users encountered were observed and noted by the moderators. To find out more about each person's user experience the participants were asked what they had felt was the most difficult during the test. Their answers corroborated what had been already noted but also added some new insight. As was noted during testing: Adding premises was difficult, the rules buttons were confusing, closing the assumption boxes was not intuitive, and it was hard to discover the drag-and-drop functionality. The new information was mainly that some users said they had problems getting started, they had trouble understanding how to input formulas and rules and remembering the commands for symbols and rules. One said as a beginner you might not know or remember the names of the rules and symbols and that could make it difficult to input them using the commands.

Users were then asked to talk about the positive aspects of using Logan. They said they liked that it was fast and that using it helped them clearly structure their proofs and avoid mistakes. For catching mistakes they said it was especially helpful that each step is validated so that they get immediate feedback during the construction of a proof. Some also enjoyed the possibility to use their keyboard for most user interaction.

While we had already asked about difficulties they had when using Logan we also wanted to know some more about what they felt were the negative aspects - things they disliked or found annoying. Here the interviewees could mention minor things that could be easily be missed during observations but which still impacted their user experience.

Among the things mentioned was that the way to go about deleting a row did not feel intuitive. The user described that if they had their cursor in the rule field and the row was empty they could still not delete it by pressing `Backspace`. They had to move the cursor to the formula field in order for this to work. They also mentioned that they would have liked to have a button to delete a row and that this should work even with non-empty rows.

Another user mentioned that they want to be able to use $\rightarrow$-elimination to conclude bottom from a formula and the negation of the same formula, with the motivation

that $\neg p$ could be interpreted as $p \to \bot$. In Logan this is done using ¬e.

An accessibility problem was brought up by a user who said that they could not read the placeholder text in the input fields due to the placeholders having a light grey color on a white background. The same user also said they much prefer dark mode interfaces in applications.

One user had stumbled on a problem with assumption boxes during testing. They said if they open a box by writing ‘Ass.’ in the rule field of a row and then later go back and edit this rule field again this will break the box and you will have to move everything back into the box if this was done by mistake, the rows will not get back into the box again automatically if you type in ‘Ass.’.

Towards the end of the interview, the users were given the opportunity to suggest improvements to consider for future versions of Logan. Several users suggested adding functionality for exporting their proofs in some format suitable for handing in to an assignment.

Other suggestions were adding an undo button, improving the navigation of the manual by adding chapters or some search functionality, adding support for constructing proofs in different logic systems such as minimal or intuitionistic logic, making the information about how to input symbols more easily accessible - for example by enabling a popup that could be displayed when a user pressed `Ctrl + h` or adding a constantly displayed box with this information somewhere in the interface, and adding information about the shortcut for each rule in the help text that comes up when a rule button is clicked.

Perhaps the most important question for the evaluation of how well Logan performed was asked last: Would you consider using Logan in your studies? The interviewees all said that if this editor had been available to them when they took an introductory course in logic they would have used it. One of the users added the condition that an export functionality would have to be present. The interviewees said they could see themselves using the editor to practice constructing proofs during the course, check their solutions before handing in assignments to catch careless mistakes they might have made and to create a digital version of their proofs for handing in to the assignments.

4.2.4 Usability Improvements

Usability issues that were identified during the testing process were grouped by severity, with priority given to implementing fixes for the most urgent ones that affected the functionality of the proof editor. Many suggestions for improvements were noted by the test users such as adding a dark mode setting or export of proofs written in the editor to a suitable format like PDF or plain text. These suggestions are more quality of life improvements and because of time constraints, a decision was made to fix more urgent issues that impact the functionality of the editor and leave minor inconveniences for later stages. Below is a summary of the fixes that

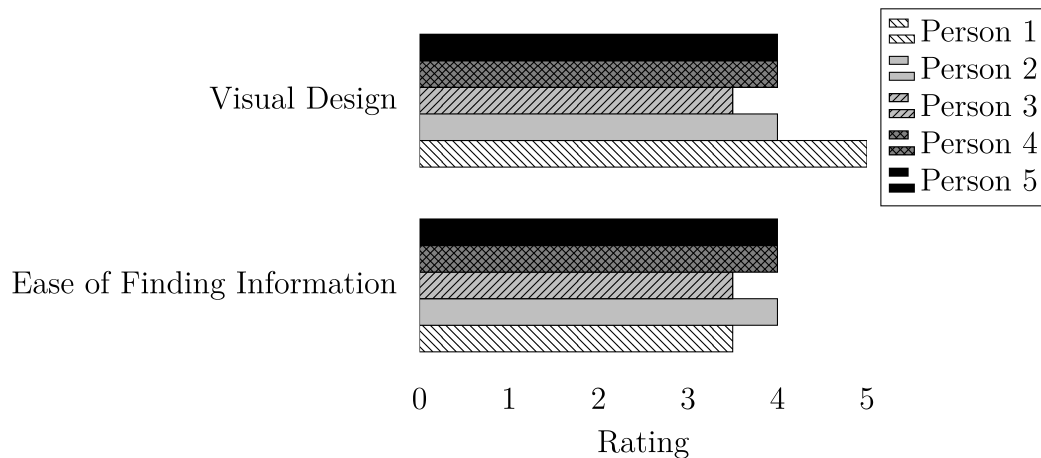

Figure 4.11: User ratings for visual design and ease of finding information.

were implemented after the user tests:

- Let users input premises beside the turnstile symbol.

  This fix aimed at providing a solution for the difficulties users had when adding a premise. It was not intuitive enough that each premise needed to be added manually by writing ‘pr’ in the rule field. Users seemed to hover their mouse beside the turnstile symbol, looking for an input field.
  Solution: Added a text field waiting for user input to the left of the turnstile symbol, indicating that premises can be inserted there.

- Make rule buttons more clear.

  This relates to the users’ initial pressing of rule buttons, expecting functionality that could be used when constructing proofs.
  Solution: The rule column header text ‘Rules’ was changed to ‘About the rules’, while also adding relevant pictures with accompanying text, to clarify and explain what they do.

- Add a button for creating a row.

  This new button provides a solution for the users’ presupposition of adding rows with a button. Users seemed to hover their mouse initially beside already generated rows in the editor, trying to find some functionality to add new rows while learning how to use the editor.
  Solution: Added a clickable button to add a new row in current proof.

- Add a clear button.

  The addition of the clear button aimed to fix a usability issue that was identified by the moderators of the testing process. When users were finished constructing a proof of a sequent and were supposed to start with a new proof, they had different methods of resetting to an initial state. One user refreshed the browser

window, while others removed each row of a proof until the editor contained only empty rows. This is inconvenient and functionality that clears the state of the editor should be implemented.
Solution: A clear button that deletes the current proof and puts the editor in the initial state with an empty proof.

- Add button for creating a new row and jumping outside the current box.

  The third new button clarifies how to jump outside of a box. Using the command `Shift + Enter` to do so is not intuitive if the user has not already found the manual or shortcut section.
  Solution: To clarify and add more possibilities of using this command, a button for the action of jumping outside of a box was added.

- `Shift + Enter` should always jump out of the box.

  When hitting `Shift + Enter` inside a box while not being on the last row, it would add a new row inside the current box instead of jumping out of the box.
  Solution: Made `Shift + Enter` work as expected.

- Change the placeholder text to be more visible.

  It was an issue that the initial placeholder text in the input fields of the proof editor was not visible enough. Namely, the premise, conclusion, formula and rule fields were affected. The text color was initially not specified, causing it to be rendered in a light gray color. This did not affect the developers, and thus it was hard to identify as an issue. During the testing process, one user however could not read the text due to the low contrast against the background, while other users did not seem to complain. After investigation, it was concluded that different web browsers render the proof editor input fields with different contrast and colorfulness. Firefox was the web browser that seemed to render the gray color of the text as almost white, while other web browsers like Microsoft Edge or Chrome used a darker tone.
  Solution: Changed the placeholder text color to a more suitable color that works for a range of web browsers.

- Consistent keyboard shortcuts for symbols.

  One user noted an inconsistency where the shortcuts for generating symbols consisted of either two or three letters. As already discussed in subsection 4.2.2, this inconsistency affected the users' flow when writing proofs. Even though only one user suggested this improvement, the developers of the project agreed that this fix was important.
  Solution: A change was made by updating all keyboard shortcuts for symbols to use two letters.

5

Discussion

In this chapter, we discuss the resulting product of this project, the proof editor Logan, as well as the approach and the results of our user testing. We also present some suggestions for future work.

5.1 The Final Product

While Logan works well for large screens, there is an issue with smaller screens, such as those on laptops, where not all of the interface can be seen at the same time. This does not make the proof editor unusable but could deter from the usability of Logan for those students who primarily use a laptop in their studies. For mobile devices Logan does not work at all, in that some user interface elements are too large and spill out into the margins on cell phone sized displays. We argue that this is not a big issue, since most, if not all, students have access to a desktop either at home or at campus and are thus able to use Logan.

5.2 The User Testing

Evaluating our proof editor with the students lead to interesting insights and suggestions from a user perspective. Functionality that makes sense to us, the developers, may not make sense to regular users, since we have been involved from the beginning both with the design and implementation. This can lead to incorrect assumptions about users in general and how the editor is going to be used. The conducted user testing identified some major issues that needed to be resolved, where some would probably be identified by the developers eventually, while others may have remained undiscovered. These major issues found during the user testing have been addressed already, see subsection 4.2.4. While we believe that the implemented solutions fix these issues we cannot be sure without further user testing. Additional motivation for more iterations of user testing is that any changes made to Logan could introduce new problems, needing to be identified.

The appreciation of the visual design was fairly high, which seems to be in accord with what is indicated by the literature; target users are accepting of fairly simple designs in scientific applications as long as the functionality is there [21].

A possible explanation for why the users seemed to focus on the colored error indications, rather than the error messages, is that the error messages are too long to read or not worded plainly enough. It could also be that the users only needed a quick reminder that they were wrong to realize themselves what went wrong. Other possible factors that could cause this behavior could be the placement or size of the messages. The usefulness and possible improvements of the error messages should probably be explored further in future user tests.

The fact that it had been a while since most users had last constructed proofs in natural deduction could possibly affect their ability to use the editor. While there were noticeable differences between the users in the time it took to complete each task relating to the basic functionality of the editor in table 4.1, these differences do not seem related to how long ago they last constructed proofs. The user who took the longest time to complete task 2, adding a premise, also had the most recent experience in writing natural deduction proofs. It could be argued that comparing the times could be irrelevant to this study anyway since none of the users was currently taking an introductory course in logic and thus this mostly compares their ability to recall and get back into proof construction. No matter how fast any one of the users was, it could still be the case that currently not taking a course in logic negatively affected their speed and ease of use. However, the time taken to complete a task can still be of interest to see if any particular task seemed to take longer for all users. One such task that stands out is adding a premise. This was a disappointing result to see since adding a premise is generally the first thing a user will do and thus something that should be made as easy and fast as possible. With the version of Logan that was tested, instead of the instant gratification that was aimed for, the users were initially confused and frustrated. This problem was addressed in an update to the editor that enabled the adding of premises in the same manner as adding a conclusion, since during testing this was a task none of the users had a problem with and which was accomplished fast as seen in the results for task one in figure 4.10. Further user testing should be done to find out if this was effective in amending the problem.

The outcomes of discount methods for usability testing depend on the moderators' ability to observe users and interpret results. Both moderators conducting the observations and interviews had theoretical knowledge about usability testing in general, but no experience of using these methods in a practical setting. The intention was to have a set script that each user would follow, without interference from the moderators. However, during the testing process, instead of going from one phase to another, spontaneous probing attempts and discussions were initiated by the moderators before the transition between phases which may have impacted the result. As an example, during one test, after a user completed the basic functionality tasks listed in table 4.1, a discussion was initiated where moderators accidentally explained how the editor works when the user was amazed how the editor could input the symbol $\wedge$ by typing in 'and'. Discussions like these should occur when the testing is already done because it may skew the result. This kind of situation could be seen as a consequence of a lack of practical experience when performing an observation.

Using inexperienced moderators could also result in some parts of the observations being misinterpreted or missed completely. It is hard to know whether this was the case and, if so, to what extent this affected the result of the user study. Efforts to counteract this possibility were made by recording the user tests to be able to go back and look at them multiple times, and by conducting interviews to be able to hear what the users themselves had to say about their experience.

The testers were all volunteers, most of whom had a personal relationship with someone in the project group. This could influence the results, since testers may have held back on criticism or given a better rating due to wanting to please their friends. The one test user who did not have any personal relationship with anyone in the project group gave similar criticism and rating, but we can still not discount the possibility that personal relationship could have influenced the results. The fact that all volunteered could mean that the selection was not representative of the target group. It could be argued that someone who volunteers for testing probably has more interest in trying new things and maybe has greater patience than the average student.

Aside from the possibility that the ratings could be influenced by personal relationships, the ratings themselves are quite subjective and it could be argued that it is hard to ascribe meaning to the results. To try to gain more insight than just the number itself could give, we also asked the testers for the motivation of their rating. From their motivations, it became clear that there were subjective views on what a certain rating entailed. Sometimes their reasoning could be similar but they still ended up giving different ratings and sometimes their ratings would be the same but their motivations were focusing on different aspects.

To construct proofs by typing in a given partially finished example is probably not how users generally will work with Logan. More usability problems may be found when users explore and use the proof editor on their own.

Observations performed remotely have the advantage that the testers can use their own equipment and be in their own environment, as noted in section 3.2. The context of use, i.e. the actual conditions in which a product is used, is important to take into consideration because it may affect the result. One could argue that using remote capabilities, where testers can be in their own environment at home instead of in a laboratory setting, adds a sense of comfort and security. However, the awareness of being observed and recorded can still affect how the users behave during testing. The users may feel nervous and change their behavior which could result in different scenarios occurring during the testing, than if they would use the proof editor without being observed. It was not uncommon during the tests that the participants were apologetic and said “I am sorry” or “I am not sure what to do here, I am sorry” which is a behavior that could imply nervousness.

5.3 Future Work

Although the editor has been deemed to meet the minimum requirements, as will be discussed in chapter 6, there is always room for improvement. If the editor is used to construct natural deduction proofs as part of an assignment, it would be convenient for the user to be able to export a proof to LaTeX markup for further editing or directly to a PDF, as mentioned previously. Since the current version of Logan lacks the export functionality, the user would have to first construct the proof using the editor and then copy the proof manually using pen and paper or a text editor in order to produce a document for submission.

Something that has been overlooked in the process of developing Logan is how well it works for students with disabilities, e.g. students who are color blind or students that use a screen reader. Accessibility and inclusiveness design is something that should probably be investigated in the future through targeted user testing or heuristic evaluations to ensure that Logan can be used by all students.

Logan can currently only be used to construct proofs in propositional and predicate logic. In the future, this could be extended to other logic systems such as minimal or intuitionistic logic.

There is not yet any support for mobile devices. This could be implemented to reach an even wider audience.

In addition to technical features, one could also work on making the process of learning natural deduction more appealing through methods such as *gamification*. While our work on the editor has focused mostly on instant feedback and ease-of-use these are only some of the aspects of the idea of gamification that has seen an increase in popularity in recent years [22]. Defined as "[the use of] game-based mechanics, aesthetics and game thinking to engage people, motivate action, promote learning, and solve problems", Llorens-Largo, Gallego-Durán, Villagrá-Arnedo, *et al.* concluded that the fun, motivation and progressiveness components of video games are some of the key factors when incorporating gamification to enhance education. An example of this is Duolingo, which is a language learning tool that has successfully applied gamification in its app [23]. This begs the question of whether gamification could bring advantages even to the field of symbolic logic. A study implementing Duolingo in a Spanish class setting cited the accessibility and persistence of the app as two of its main strengths, with many students completing more lessons than those required for the class. Perhaps in our case, having an effortless way of inputting sequents from coursebook exercises would amount to an easy way to lower the barrier for doing more proofs.

Furthermore, another more closely related example is the logical game *Polymorphic Blocks* [24] which encodes natural deduction as a puzzle game. When evaluating the ease-of-use of Polymorphic Blocks Lerner, Foster and Griswold pitched a group of people using only pen-and-paper to prove sequents - who were given a 30 minute lecture and better help - versus one only using the game, and found that the second

group were consistently faster. This shows that the puzzle game being more fun and engaging spurred learning. However some caveats are that while the game did not teach symbols, it also performed *unification* and *substitution application* for the users, meaning the users did not have to learn these concepts. This is further justification for why it would be interesting to see a game-inspired progression system retrofitted on top of our editor or similar, which has a more efficient and subjectively more "fun" interface than pen-and-paper while lacking many other facets of gamification.

6
Conclusion

Examining Logan against the minimum requirements from section 1.1, we conclude that it does meet all of them. The proof editor allows the user to construct natural deduction proofs by applying inference rules in both propositional and predicate logic, and at the same time validate a given proof.

In addition to the minimum requirement of allowing the user to construct a proof, four goals were set up for the project in section 1.1. When comparing Logan to these goals, we can conclude the following.

- It is built as a web application, making it easily accessible on any platform with access to a modern web browser.
- The interface has a clear resemblance to the way natural deduction proofs are presented in [3].
- It provides feedback to the user by showing error messages describing any errors made.
- In the user testing all users managed to construct correct proofs with the aid of the available information in the proof editor.

The limited user testing conducted suggest that Logan works fairly well for students, and the plan is to introduce it as a tool in the upcoming instances of the course DAT060/DIT201 *Logic in computer science* at University of Gothenburg and Chalmers University of Technology. This presents an opportunity to further evaluate how useful Logan is and find more usability problems that can be addressed in future updates of Logan.

Bibliography

[1] Department of Computer Science and Engineering, Chalmers. (2020). "DAT060 / DIT202 Logic in computer science", [Online]. Available: https://chalmers.instructure.com/courses/10148 (visited on 04/29/2021).

[2] A. Bolotov, V. Bocharov, A. Gorchakov, and V. Shangin, "Automated first order natural deduction", Jan. 2005, pp. 1292–1311.

[3] M. Huth and M. D. Ryan, *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2018.

[4] U. Norell, *Towards a practical programming language based on dependent type theory*. Citeseer, 2007, vol. 32.

[5] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[6] T. Hallgren. (Oct. 26, 2020). "Beviseditorn alfa", [Online]. Available: https://cth.altocumulus.org/~hallgren/Alfa/ (visited on 03/31/2021).

[7] L. de Moura. (2021). "Lean", [Online]. Available: https://leanprover.github.io/ (visited on 03/31/2021).

[8] E. Björnsson, F. Johansson, J. Liu, J. Olsson, H. Ly, and A. Widbom, "Proof editor for natural deduction in first-order logic the evaluation of an educational aiding tool for students learning logic", B.S. thesis, 2017.

[9] OpenLogicProject. (2019). "Natural deduction proof editor and checker", [Online]. Available: https://github.com/OpenLogicProject/fitch-checker (visited on 03/31/2021).

[10] J.-W. Chen and J. Zhang, "Comparing text-based and graphic user interfacesfor novice and expert users", 2007. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2655855/pdf/amia-0125-s2007.pdf (visited on 04/29/2021).

[11] W3Techs. (Apr. 7, 2021). "Usage statistics of javascript as client-side programming language on websites", [Online]. Available: https://w3techs.com/technologies/details/cp-javascript/ (visited on 04/07/2021).

[12] D. Crockford, *JavaScript: The Good Parts: The Good Parts*. O'Reilly Media, Inc., 2008.

[13] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for guis", *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 411–422, 2013.

[14] B. Russel and A. N. Whitehead, *Principia Mathematica*. Cambridge University Press, Jan. 1927, ISBN: 9780521067911.

[15] C.-L. Chang and R. C.-T. Lee, *Symbolic logic and mechanical theorem proving*. Academic press, 2014.

[16] H. Sharp, Y. Rogers, and J. Preece, *Interaction design: beyond human-computer interaction*. Wiley, 2019.

[17] "Ergonomics of human-system interaction — Part 210: Human-centred design for interactive systems", International Organization for Standardization, Geneva, CH, Standard, Mar. 2019.

[18] J. Nielsen, *Why you only need to test with 5 users*. [Online]. Available: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/.

[19] ——, "Applying discount usability engineering", *IEEE software*, vol. 12, no. 1, pp. 98–100, 1995.

[20] (2021). "Github - purescript-halogen/purescript-halogen", [Online]. Available: https://github.com/purescript-halogen/purescript-halogen (visited on 03/26/2021).

[21] J. Tidwell, "The patterns", in *Designing interfaces*, 2nd ed. O'Reilly Media, Inc., pp. 8–23.

[22] F. Llorens-Largo, F. J. Gallego-Durán, C. J. Villagrá-Arnedo, P. Compañ-Rosique, R. Satorre-Cuerda, and R. Molina-Carmona, "Gamification of the learning process: Lessons learned", *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, vol. 11, no. 4, pp. 227–234, 2016.

[23] P. Munday, "The case for using duolingo as part of the language classroom experience", *RIED: revista iberoamericana de educación a distancia*, vol. 19, no. 1, pp. 83–101, 2016.

[24] S. Lerner, S. R. Foster, and W. G. Griswold, "Polymorphic blocks: Formalism-inspired ui for structured connectors", in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 2015, pp. 3063–3072.