Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

A proof assistant for first-order logic

Bachelor's Thesis

2018 Zoltán Onódy

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

A proof assistant for first-order logic

Bachelor's Thesis

Study programme:	Applied Computer Science
Study field:	Applied Informatics
Department:	Department of Applied Informatics
Supervisor:	Mgr. Ján Kľuka, PhD.
Consultant:	RNDr. Jozef Šiška, PhD.

Bratislava 2018

Zoltán Onódy

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Zoltán Onódy
Study programme:	Applied Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Applied Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: A proof assistant for first-order logic

Annotation: The learning outcomes of mathematics courses at our faculty include acquisition of the mathematical style of writing and reasoning. Achieving this goal requires exercises in formulating definitions, propositions, and proofs with timely feedback. Mathematical language and reasoning use established constructs whose theoretical models are the language of first-order logic and formal systems of axioms and derivation rules (Hilbert calculus [1], sequent calculus [1], semantic tableaux [2], valuation tress [3], etc.). There are, however, also practical software systems for composition of theories and proofs in readable formal languages similar to the language of mathematical writing, and for intelligent automated checking of proofs written in these languages (Mizar, Isabelle/Isar [4]). Their use in teaching practice is, however, discouraged by the need to learn the formal language, the mode of interaction with the software, opaque proof checking (it is not quite clear which leaps of thought the machine accepts and why), and the need to install software. We thus aim at developing a web-based environment for development of theories and interactive composition of proofs from derivation rules expressed in natural language with easily understandable verification of correctness of each proof step.

Aim: The goal of the thesis is to design, implement, and test a prototype interactive assistant for composition of proofs of propositions in first-order logic as described in Annotation. Correctness will be ensured by basing the design on a selected suitable formal system [1,2,3]. The assistant should then be incorporated into a development environment allowing to specify the language and axioms of a theory and state and prove propositions, using earlier-proved propositions as lemmata. The implementation will use client-side web technologies in order to facilitate deployment.

Literature:

[1] Barwise, J. (ed.) (1982): Handbook of Mathematical Logic. North Holland.
[2] Smullyan, R.M. (1968): First-Order Logic. Springer.
[3] Kľuka, J. (2011): Modulárne programovanie a verifikácia v druhorádovej aritmetike. Dizertačná práca. Univerzita Komenského.
[4] Markus Wenzel, Freek Wiedijk (2002): A Comparison of Mizar and Isar. J. Autom. Reasoning 29(3–4): 389–411.

Keywords: proof assistant, first-order logic, formalized mathematics, client-side web application

69142122

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Supervisor:	Mgr. Ján Kľuka, PhD.	
Consultant:	RNDr. Jozef Šiška, PhD.	
Department:	FMFI.KAI - Department of Applied Informatics	
Head of department:	prof. Ing. Igor Farkaš, Dr.	
Assigned:	12.10.2017	
Approved:	16.10.2017	doc. RNDr. Damas Gruska, PhD. Guarantor of Study Programme

....
Student

....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Zoltán Onódy
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: aplikovaná informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: A proof assistant for first-order logic
Dokazovací asistent pre logiku prvého rádu

Anotácia: K cieľom výučby matematiky na našej fakulte patrí osvojenie si matematického spôsobu vyjadovania sa a argumentácie. Skutočné osvojenie vyžaduje precvičovanie formulovania definícií, tvrdení a ich dôkazov s včasnou spätnou väzbou. Matematický jazyk a argumentácia využívajú ustálené konštrukty, ktorých teoretickými modelmi sú jazyk logiky prvého rádu a formálne systémy axióm a odvodzovacích pravidiel (hilbertovský kalkul [1], sekventový kalkul [1], sémantické tablá [2], valuačné stromy [3], atď.). Vznikli však aj praktické systémy na zápis teórií a dôkazov tvrdení v čitateľných formálnych jazykoch blízkych jazyku matematického textu, a na inteligentnú strojovú kontrolu správnosti takto zapísaných dôkazov (Mizar, Isabelle/Isar [4]). Od ich použitia pri výučbe však odrádza nutnosť naučiť sa formálny jazyk, spôsob interakcie, netransparentná kontrola dôkazov (nie je zrejmé, aké skoky v úvahách stroj akceptuje a prečo) a nutnosť inštalácie softvéru. Naším zámerom je preto vyvinúť webové prostredie pre zápis teórií a interaktívnu tvorbu dôkazov zo slovne vyjadrených argumentačných pravidiel s ľahko pochopiteľnou kontrolou správnosti postupu.

Cieľ: Cieľom práce je navrhnúť, implementovať a otestovať prototyp interaktívneho asistenta na tvorbu dôkazov tvrdení v logike prvého rádu v zmysle anotácie. Korektnosť zabezpečí založenie návrhu na vybranom vhodnom formálnom systéme [1,2,3]. Následne by mal byť asistent zapracovaný do prostredia, v ktorom je možné špecifikovať jazyk a axiómy teórie, definovať pojmy a formulovať a dokazovať tvrdenia, pričom skôr dokázané tvrdenia sú využiteľné ako lemy. Pre jednoduché nasadenie implementácia využije webové technológie na strane klienta.

Literatúra:
[1] Barwise, J. (ed.) (1982): Handbook of Mathematical Logic. North Holland.
[2] Smullyan, R.M. (1968): First-Order Logic. Springer.
[3] Kľuka, J. (2011): Modulárne programovanie a verifikácia v druhorádovej aritmetike. Dizertačná práca. Univerzita Komenského.
[4] Markus Wenzel, Freek Wiedijk (2002): A Comparison of Mizar and Isar. J. Autom. Reasoning 29(3–4): 389–411.

Kľúčové slová: dokazovací asistent, logika prvého rádu, formalizovaná matematika, webová aplikácia na strane klienta

Vedúci: Mgr. Ján Kľuka, PhD.

69142122

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Konzultant:	RNDr. Jozef Šiška, PhD.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	prof. Ing. Igor Farkaš, Dr.
Dátum zadania:	12.10.2017
Dátum schválenia:	16.10.2017

doc. RNDr. Damas Gruska, PhD.
garant študijného programu

....
študent

....
vedúci práce

Acknowledgments: I'd like to thank especially Mgr. Ján Kľuka, PhD. for his guidance, explanation of mathematical background needed for this work and for thoughtful comments and useful suggestions. Thanks also goes to RNDr. Jozef Šiška, PhD. for mentoring me in the programming part of this work. I would like to acknowledge everyone who tested the proof assistant, namely, in no particular order: Alexandra Nyitraiová, Michal Kováč, Jozef Kubík, Daniel Kyselica, Soňa Senkovičová, Dávid Šuba, Tomáš Velich, Emil Zvarík, Matej Rychtárik and Kryštof Kiss. Many thanks goes once again to my girlfriend Alexandra Nyitraiová who gave me a lot of feedback during the tool development.

Abstrakt

Študenti na univerzite majú často problémy osvojiť si matematický štýl uvažovania. Na zmiernenie tohto problému sme sa rozhodli vytvoriť interaktívny webový dokazovací asistent. Náš dokazovací asistent podporuje tri typy dôkazov: priamy dôkaz, dôkaz analýzou prípadov a dôkaz sporom. Dôkaz je predovšetkým založený na reťazení tvrdení. Dokazovací asistent overuje, či nové tvrdenia sú logickými dôsledkami predchádzajúcich tvrdení a nejakého inferenčného pravidla, ktoré asistent pozná. To by malo študentom umožniť sústrediť sa na dôkaz samotný namiesto nepremysleného skúšania inferenčných pravidiel. Používateľ do asistenta píše sformalizované prvorádové formuly. Dokazovací asistent je postavený nad hybridným formálnym systémom pozostávajúcim z hilbertovského a sekventového kalkulu a pozná aj ďalšie korektné inferenčné pravidlá. Asistent bol naprogramovaný vo funkcionálnom programovacom jazyku Elm. Je funkčným prototypom, ktorý sa bude v budúcnosti ďalej vyvíjať. Jednoduchosť používania sme testovali so študentmi. Tí nám poskytli pozitívnu a informatívnu spätnú väzbu, ktorú sme čiastočne zapracovali.

Kľúčové slová: dokazovací asistent, logika prvého rádu, formalizovaná matematika, webová aplikácia na strane klienta

Abstract

University students often struggle to master mathematical style of reasoning. In order to alleviate this problem, we have decided to build an interactive, web-based proof assistant. Our proof assistant supports three types of proofs: direct proof, proof by cases, and proof by contradiction. The proof is built primarily by chaining claims. The proof assistant checks whether a new claim logically follows from previous claims by some sound basic rule of inference known to it. This should allow students to focus on intuitively determining logical consequences rather than on finding or blindly trying inference rules. The user writes the claims as first-order formulas. The proof assistant is based on a hybrid formal proof system mixing Hilbert calculus and Sequent calculus. The proof assistant also implements many common additional sound rules of inference. The application has been implemented in the functional programming language Elm. It is a working prototype and we expect further development of the project in the future. The ease of interaction with the proof assistant was tested by students who gave us positive and informative feedback some of which we have implemented.

Keywords: proof assistant, first-order logic, formalized mathematics, client-side web application

Contents

Introduction

Mathematical courses at our faculty include teaching the mathematical style of writing and reasoning. Teaching this subject requires exercises in the formulation of definitions, propositions, and proofs.

Giving feedback to students is time-consuming for the teacher. The problems in proofs are often in details in some steps rather than the idea behind the proof [7].

One of the possible solutions is using automatic tools – proof assistants – to check the correctness of the proofs. There are several proof assistants, for example, Isabel/Isar, Clausal Language (CL) or Fitch. These proof assistants have various shortcomings from the teaching perspective They are either too powerful, focused on inference rules of a particular formal system, or they require installation.

The goal of the thesis is to design, implement, and test an interactive prototype of a proof assistant which is accessible to students and easily extendable in the future.

We aim to develop a web-based application to better support portability and usability. The application should be interactive and check the correctness of each proof step. In order for a step to be correct, it must be derived from the previous formulas by some rule of inference. The proof assistant must be sound and complete. These properties will be ensured by using a suitable formal system such as Hilbert calculus or Sequent calculus. These formal systems, however, have only fundamental rules which makes them impractical. We will, therefore, add more sound rules to alleviate the problem.

This thesis is structured as follows: Section 1 introduces the problem. It discusses the mathematical background required to build a proof assistant. In Section 2, we go through the application design and clarify the decisions we have made. Section 3 shows the implementation of the proof assistant, what were the main challenges and how they were solved. In Section 4, we describe the process and discuss the results of a usability evaluation of our tool. The thesis concludes with ideas for future work.

Chapter 1

Background

To fully understand the content of this thesis it is necessary to understand the mathematical background behind it. In this chapter, we explain what mathematical logic is, define the first-order logic, show the types of frequently used mathematical proofs and show a formal logical argumentation. We describe the tools we are going to use to build our application and what applications, similar to the one we intend to develop in this thesis, already exists.

1.1 Mathematical logic

Mathematical logic is a field exploring the power of formal proof systems close to theoretical computer science and foundations of mathematics.

1.1.1 First-order logic

Syntax of first-order logic. The *language* of first-order logic is a set L of words which are built from non-logical symbols of L and logical symbols. The *non-logical symbols* of L are function symbols, relation symbols and constant symbols [6]. Each function symbol has a number $\#(f) = n$, called its *arity*; f is then called an n*-ary* function symbol. Each relation symbol also has an arity $\#(R) = n$; R is then called an n*-ary* relation symbol.

For example let's have a language $L = \{\ \}$ of set theory. It has no function or constant symbols and has one relation symbol for which $\#(\ ) = 2$ holds.

To illustrate another example, let's have $L = \{+, 0\}$ from group theory. It has one function symbol (where $\#(+) = 2$), one constant symbol (0) and no relation symbols.

The *logical symbols* are , , $\neg, =,$, , variables $x, y, z, \ldots$ and parentheses) and (to keep the symbols readable. Each word from L is called an *expression*. From all the expressions we want to select those which we can assign a meaning.

The ***terms*** of L form the smallest set of expressions containing the variables x, y, z, ..., all constant symbols of L and closed under the formation rule: if $t_1, \ldots, t_n$ are terms of L and if $f \quad L$ is an n-ary function symbol, then the expression $f(t_1, \ldots, t_n)$ is a term of L [6]. A ***closed term*** is a term in which no variable appears.

For instance, if $L = \{+, 0\}$ then, the terms are expressions like

$$+(x, y) \quad \text{and} \quad +(0, +(x, 0))$$

We naturally agree to use

$$x + y \quad \text{and} \quad 0 + (x + 0)$$

instead. Thus moving the function symbol inside and leaving the parentheses off, if there will be no confusion. We use abbreviation nx for $(\cdots((x + x) + x) + \cdots + x)$, n times for $n \geq 1$.

An ***atomic formula*** of L is one of the following two expressions: $(t_1 = t_2)$ where t_1 and t_2 are terms from L and $R(t_1, \ldots, t_n)$ where R is an n-ary relation symbol from L and $t_1, \ldots, t_n$ are terms from L [6].

For example, in the language of group theory $L = \{+, 0\}$ atomic formulas are $(x = 0)$, $(x + y = y + z)$, $\ldots$. Another example is from set theory. In the language $L = \{ \quad \}$ where all terms are variables, the only atomic formulas are those of the form $(v = w)$ and $\quad (vw)$ for variables v and w. We write the latter as $v \quad w$.

The first-order ***formulas*** of L form the smallest set of expressions containing atomic formulas and closed formulas under the formation rules:

(i) If φ, ψ are formulas so are $\neg\varphi$, $(\varphi \quad \psi)$, $(\varphi \quad \psi)$, $(\varphi \to \psi)$;

(ii) if φ is a formula and v is a variable, then $\quad v\varphi$ and $\quad v\varphi$ are formulas as well [6].

Whenever the same symbol is repeated, we omit the parentheses and we associate them to the right. Thus

$$\varphi \quad \psi \quad \theta \quad \text{is} \quad (\varphi \quad (\psi \quad \theta))$$

and

$$\varphi \to \psi \to \theta \quad \text{is} \quad (\varphi \to (\psi \to \theta))$$

For example, let $L = \{+, 0\}$. The following are formulas:

$$(x + y = 0)$$

$$y(x + y = 0)$$

$$x \quad y(x + y = 0)$$

Notice that the first formula contains 2 free variables. In the second the variable x is free but y is bound up by existential quantifier ( ) and in the last formula both x and y are bound up. Only the last formula makes any intuitive sense as an axiom.

Next, we will define the exact notion of ***free variable***. It is defined by induction on the length of the formula φ.

The set FreeVariables(φ) of a formula φ is defined as follows:

(i) If φ is an atomic formula, then FreeVariables(φ) is a set of variables found in φ,

(ii) FreeVariables($\neg\varphi$) = FreeVariables(φ),

(iii) FreeVariables($\varphi \quad \psi$) = FreeVariables($\varphi \quad \psi$) =
FreeVariables($\varphi \rightarrow \psi$) = FreeVariables(φ) FreeVariables(ψ),

(iv) FreeVariables($v\varphi$) = FreeVariables($v\varphi$) = FreeVariables(φ) − $\{v\}$ [6].

A first-order ***sentence*** of L is a formula without any free variables [6]. A ***theory*** is a set of sentences.

Formulas with free variables often occur in proofs, free variables can be substituted for in order to specialize the formula. We define ***substitution*** as follows: if $\varphi(v)$ is a formula and t is a term then $\varphi(t/v)$ (or $\varphi(t)$) means replacing all the occurrences of variable v by the term t, assuming that none of the variables in t is bound in φ. If there is any variable bound, we can simple rename the bound variable.

Semantics of first-order logic. Given a language L we have a concept of a structure for L. The structure $\mathfrak{M}$ assign interpretations of the relation, function and constant symbols of L which is precisely defined as follows:

A ***structure*** for L is a pair $\mathfrak{M} = \quad M, F \quad$ where M is a nonempty set and F is an operation with domain L such that, writing $x^{\mathfrak{M}}$ for $F(x)$,

(i) if $R \quad L$ is an n-ary relation symbol, then $R^{\mathfrak{M}} \quad M^n$;

(ii) if $f \quad L$ is an n-ary function symbol, then $f^{\mathfrak{M}}\colon M^n \rightarrow M$;

(iii) if $c \quad L$ is a constant symbol then $c^{\mathfrak{M}} \quad M$ [6].

For example, if $L = \{+, 0\}$ is a language in group theory, then a structure for L has the form $\mathfrak{M} = \quad M, +^M, 0^M \quad$ where M is a nonempty set, $+^M\colon M \times M \rightarrow M$ and $0^M \quad M$.

So far the terms, formulas and sentences are simply a finite strings of symbols. We need to assign the intended meaning to our logical symbols. This is done by defining the ***satisfaction relation*** $\mathfrak{M} \quad \varphi$ between structures and sentences.

Let $\mathfrak{M} = \quad M, F \quad$ be a structure for language L. An ***assignment*** in $\mathfrak{M}$ is a function s with domain the set of variable of L and range a subset of M. An assignment s thus assigns the meaning $s(v)$ to the variable v. We can then define, $t^{\mathfrak{M}}$ for each term t from L, which maps assignments to elements of M.

Let M be a given structure. For t, a term of L define *the value* $t^{\mathfrak{M}}$ *of the term* t as follows.

(i) If t is a constant symbol c, then $t^{\mathfrak{M}}(s) = c^{\mathfrak{M}}$ for all s;

(ii) if t is a variable v, then $t^{\mathfrak{M}}(s) = s(v)$ for all s;

(iii) if t is the term $f(t_1, \ldots, t_n)$ then, for all s, define $t^{\mathfrak{M}}(s) = f^{\mathfrak{M}}(t_1^{\mathfrak{M}}, \ldots, t_n^{\mathfrak{M}}(s))$ [6].

In (iii), since each of $t_1, \ldots, t_n$ is simpler than t we can assume by induction on (the complexity of) terms that $t_1^{\mathfrak{M}}, \ldots, t_n^{\mathfrak{M}}$ are already defined. $f^{\mathfrak{M}}$ is defined since $\mathfrak{M}$ is a structure for L and $f \quad L$. The reader should note that if $s_1(v) = s_2(v)$ agree on all variables v appearing in t, then $t^{\mathfrak{M}}(s_1) = t^{\mathfrak{M}}(s_2)$. Thus $t^{\mathfrak{M}}$, as a function, depends on only a finite number of values of its argument s.

Let $\mathfrak{M}$ be a structure for L. We define the relation $\mathfrak{M} \quad \varphi[s]$ for all assignments s and all formulas φ as follows [6].

(i) $\mathfrak{M} \quad (t_1 = t_2)[s]$ iff $t_1^{\mathfrak{M}}(s) = t_2^{\mathfrak{M}}(s)$,

(ii) $\mathfrak{M} \quad R(t_1, \ldots, t_n)[s]$ iff $(t_1^{\mathfrak{M}}(s), \ldots, t_n^{\mathfrak{M}}(s)) \quad R^{\mathfrak{M}}$,

(iii) $\mathfrak{M} \quad \neg\varphi[s]$ iff not $\mathfrak{M} \quad \varphi[s]$,

(iv) $\mathfrak{M} \quad (\varphi \quad \psi)[s]$ iff $\mathfrak{M} \quad \varphi[s]$ and $\mathfrak{M} \quad \psi[s]$,

(v) $\mathfrak{M} \quad (\varphi \quad \psi)[s]$ iff $\mathfrak{M} \quad \varphi[s]$ or $\mathfrak{M} \quad \psi[s]$ or both,

(vi) $\mathfrak{M} \quad (\varphi \rightarrow \psi)[s]$ iff either not $\mathfrak{M} \quad \varphi[s]$ or else $\mathfrak{M} \quad \psi[s]$,

(vii) $\mathfrak{M} \quad (\quad v\varphi)[s]$ iff there is an $a \quad M$ such that $\mathfrak{M} \quad \varphi[s(v/a)]$,

(viii) $\mathfrak{M} \quad (\quad v\varphi)[s]$ iff for all $a \quad M$, $\mathfrak{M} \quad \varphi[s(v/a)]$.

A formula is called *valid* if it is satisfied in every structure and under every assignment. A formula φ is a *logical consequence* of a theory T if every structure and every assignment that satisfy the theory T also satisfy the formula φ.

These definitions should cover everything we will need, from first-order logic, for our proof assistant.

1.2 Mathematical proofs

Proof in mathematics is a demonstration that some statement holds using only valid and logical steps. In mathematics, there are several types of proofs. In this section, we are going to show proof examples which include some amount of natural language. Later, in Section 1.3 we also show formal proofs which completely omit the natural language.

Direct proof. The most common method of mathematical proofs is *direct proof*. The conclusion is built by combining axioms, definitions and earlier theorems or lemmata. Let's see an example of direct proof. Prove that the sum of two even integers is always even. Let the integers be x and y. As the x and y are even, some x' and y' exists such as $x = 2x'$ and $y = 2y'$. Therefore, $x + y = 2x' + 2y' = 2(x' + y')$. As we can see, the sum of $x + y$ has a factor of 2, and therefore the sum of any even numbers is an even number.

Proof by contradiction. In *proof by contradiction*, we assume that the statement is false and show that our assumption leads to nonsense. We are therefore led to conclude that our assumption was wrong and the statement must be true. Let's see an example. Prove that there are infinitely many prime numbers. To prove this statement by contradiction, let's assume there are finitely many prime numbers $p_1 < p_2 < ... < p_n$ where p_n is the highest prime number. Let's construct a new number $k = p_1 \quad p_2 \quad ... \quad p_n + 1$. This number is not divisible by any prime $p_1, ..., p_n$, as there always be a remainder of 1. Therefore k is either a prime number or there exists a prime number p such as $p_n < p < k$ which divides k without a remainder. In both cases, there is a prime number higher than p_n which is in contradiction with the assumption that p_n is the highest prime number.

Proof by induction. *Proof by induction* is the type of proof where one proves the base case and shows that if $P(n)$ holds so does $P(n + 1)$. Let's prove

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^n} = \frac{2^n - 1}{2^n}$$

As the base case for $n = 1$ we have

$$\frac{1}{2} = \frac{2^n - 1}{2^n} = \frac{2^1 - 1}{2^1} = \frac{1}{2}$$

Our induction hypothesis is

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^n} = \frac{2^n - 1}{2^n}$$

We need to prove our induction step

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^n} + \frac{1}{2^{n+1}} = \frac{2^{n+1} - 1}{2^{n+1}}$$

by applying the induction hypothesis we get

$$\frac{2^n - 1}{2^n} + \frac{1}{2^{n+1}} = \frac{2^{n+1} - 1}{2^{n+1}}$$

which we can transform

$$\frac{2}{2} \quad \frac{2^n - 1}{2^n} + \frac{1}{2^{n+1}} = \frac{2^{n+1} - 2}{2^{n+1}} + \frac{1}{2^{n+1}} = \frac{2^{n+1} - 1}{2^{n+1}}$$

Proof by exhaustion. *Proof by exhaustion* also known as *proof by cases*, is the type of proof where one divide the proof to multiple cases and proves all of them separately. Let's see a proof by exhaustion. Prove that if $n \quad \mathbb{Z}$ then n^2+3n+4 is even. We know that every integer is odd or even. We thus consider two cases: Case 1) n is even. We know that $n = 2k+1$ for some k. Therefore, $n^2+3n+4 = (2k+1)^2+3(2k+1)+4 = (4k^2+4k+1)+(6k+3)+4 = 4k^2+10k+12 = 2(2k^2+5k+6)$. Case 2) n is odd. We know that $n = 2k$ for some k. Therefore, $n^2+3n+4 = (2k)^2+3(2k)+4 = 4k^2+6k+4 = 2(2k^2+3k+2)$. In both cases we see that the formula n^2+3n+4 has the factor 2, so the formula is even for any number n.

1.3 Formal proofs

In Section 1.2, we showed proof types which included some amount of natural language. In this section we will show two formal proof systems - Hilbert calculus and the sequent calculus.

1.3.1 Formal systems

A *rule of inference* is a function which takes premises, analysis their syntax and makes a conclusion. A rule of inference can, for example, look like

$$X_1\colon \frac{\psi \to \varphi \quad \neg\psi \to \varphi}{\varphi}.$$

An arbitrary formula φ is *proven* if we could prove the formulas $\psi \to \varphi$ and $\neg\psi \to \varphi$. This rule of inference takes two premises and makes one conclusion. It is evident that to prove those two premises we eventually need a rule of inference which does not take any premise. That means, different rules of inference can take different numbers of premises. A rule of inference, which does not take any premise is called a *logical axiom* and can be chosen for example as $X_2\colon \overline{\varphi \to (\varphi \quad \psi)}$ and $X_3\colon \overline{\varphi \to (\psi \quad \varphi)}$. With these three rules we can easily prove in three steps the formula $\varphi \quad \neg\varphi$.

1) $\varphi \to (\varphi \quad \neg\varphi)$; from X_2
2) $\neg\varphi \to (\varphi \quad \neg\varphi)$; from X_3
3) $\varphi \quad \neg\varphi$; from X_1 by using 1 and 2

A rule of inference is *sound* if its conclusion is a logical consequences of its premises.

A *formal system*, therefore, is a set of rules of inference. The system is called *sound* if every formula we can prove from a set of assumptions is a logical consequence of those assumptions. The system is called *complete* if from a set of assumptions we can prove every formula that is their logical consequence.

1.3.2 Hilbert calculus

Hilbert calculus is a formal proof system with one propositional rule of inference and two quantifier rules and several axioms.

Propositional logic. We chose the axiom system from the book [16]. These axioms along with modus ponens (MP) describe classical propositional logic.

$$\begin{aligned}
MP\colon &\quad \frac{\varphi \quad \varphi \rightarrow \psi}{\psi} \\
A_1\colon &\quad \varphi \rightarrow (\psi \rightarrow \varphi) \\
A_2\colon &\quad (\varphi \rightarrow (\psi \rightarrow \xi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \xi)) \\
A_3\colon &\quad (\neg\varphi \rightarrow \neg\psi) \rightarrow ((\neg\varphi \rightarrow \psi) \rightarrow \varphi) \\
A_4\colon &\quad (\varphi \quad \psi) \rightarrow \varphi, (\varphi \quad \psi) \rightarrow \psi \\
A_5\colon &\quad \varphi \rightarrow (\psi \rightarrow (\varphi \quad \psi)) \\
A_6\colon &\quad \varphi \rightarrow (\varphi \quad \psi), \psi \rightarrow (\varphi \quad \psi) \\
A_7\colon &\quad (\varphi \rightarrow \xi) \rightarrow ((\psi \rightarrow \xi) \rightarrow ((\varphi \quad \psi) \rightarrow \xi))
\end{aligned}$$

First-order logic. To have the first-order logic, we add two more axioms and two rules of inference for quantifiers [16].

$$\begin{array}{rcl}
B_1\colon & \quad x(\varphi) \rightarrow \varphi(x/t) & \text{where } t \text{ may be substituted for } x \text{ in } \varphi \\
B_2\colon & \varphi(x/t) \rightarrow \quad x\varphi & \text{where } t \text{ may be substituted for } x \text{ in } \varphi \\
\text{Gen-A:} & \dfrac{\psi \rightarrow \varphi}{\psi \rightarrow \quad x\varphi} & \text{where } x \text{ is not a free variable of } \psi \\
\text{Gen-E:} & \dfrac{\varphi \rightarrow \psi}{\quad x\varphi \rightarrow \psi} & \text{where } x \text{ is not a free variable of } \psi
\end{array}$$

Proof in Hilbert calculus. Let T be a theory. A finite sequence of formulas in Hilbert calculus is called a *proof from* T if every formula is either an axiom or it derives from the previous formulas and a rule of inference or is a member of T. A *proof of a formula* φ *from* T is a proof from T whose last element is φ.

Soundness and completeness. Let T be a theory and φ a formula in the following theorems:

If φ is provable in Hilbert calculus from T then φ is a logical consequence of T.

If φ is a logical consequence of T then φ is provable in Hilbert calculus from T.

These two theorems were proven by Švejdar [16].

1.3.3 Sequent calculus

The sequent calculus is a formal proof system with only one axiom and many rules of inference. In this calculus, in comparison to Hilbert calculus, one does not prove formulas but sequents.

A:	$\overline{\Gamma, \varphi \qquad \Delta, \varphi}$	
W:	$\dfrac{\Gamma \qquad \Delta}{\Gamma \qquad \Delta, \varphi}$	$\dfrac{\Gamma \qquad \Delta}{\Gamma, \varphi \qquad \Delta}$
-r:	$\dfrac{\Gamma \qquad \Delta, \varphi}{\Gamma \qquad \Delta, \varphi \quad \psi}$	$\dfrac{\Gamma \qquad \Delta, \varphi}{\Gamma \qquad \Delta, \psi \quad \varphi}$
-l:	$\dfrac{\Gamma, \varphi \qquad \Delta}{\Gamma, \varphi \quad \psi \qquad \Delta}$	$\dfrac{\Gamma, \varphi \qquad \Delta}{\Gamma, \psi \quad \varphi \qquad \Delta}$
-r:	$\dfrac{\Gamma \qquad \Delta, \varphi \ , \ \Gamma \qquad \Delta, \psi}{\Gamma \qquad \Delta, \varphi \quad \psi}$	
-r:	$\dfrac{\Gamma, \varphi \qquad \Delta \ , \ \Gamma, \psi \qquad \Delta}{\Gamma, \varphi \quad \psi \qquad \Delta}$	
$\neg$-l:	$\dfrac{\Gamma \qquad \Delta, \varphi}{\Gamma, \neg\varphi \qquad \Delta}$	
$\neg$-r:	$\dfrac{\Gamma, \varphi \qquad \Delta}{\Gamma \qquad \Delta, \neg\varphi}$	
$\rightarrow$-r:	$\dfrac{\Gamma, \varphi \qquad \Delta, \psi}{\Gamma \qquad \Delta, \varphi \rightarrow \psi}$	
$\rightarrow$-l:	$\dfrac{\Gamma \qquad \Delta, \varphi \ , \ \Pi, \psi \qquad \Lambda}{\Gamma, \Pi, \varphi \rightarrow \psi \qquad \Delta, \Lambda}$	
Cut:	$\dfrac{\Gamma \qquad \Delta, \varphi \ , \ \Pi, \psi \qquad \Lambda}{\Gamma, \Pi \qquad \Delta, \Lambda}$	

Table 1.1: Rules in the sequent calculus

Sequent. A *sequent* is a pair of finite sets of formulas. We write the *sequent* containing sets Δ and Γ as $\Gamma \qquad \Delta$. The intended meaning of the sequent is: if all the formulas in Γ hold, then at least one formula from Δ holds as well. The set Γ, in the sequent, is called *antecedent* and the set Δ is called *succedent*. Both antecedent and succedent may be empty sets. To simplify the notation of set union, we write Γ, φ instead of $\Gamma \quad \{\varphi\}$.

Propositional rules. The sequent calculus has rules show in table 1.1.

We see that in comparison to Hilbert calculus, where there is only one rule of inference and many axioms, in the sequent calculus, there is only one axiom A and many rules of inference. A is sometimes also called the *initial sequent*.

First-order logic. In order to have the first-order logic, we must have new rules for eliminating and introducing quantifiers. Let φ and ψ be first-order formulas and Γ and Δ be sets of first-order formulas. We can add the following rules of inference:

$$\text{-r:}\quad \frac{\Gamma \qquad \Delta, \varphi(t)}{\Gamma \qquad \Delta, \;\; x\varphi(x)},$$

$$\text{-l:}\quad \frac{\Gamma, \varphi(t) \qquad \Delta}{\Gamma, \;\; x\varphi(x) \qquad \Delta},$$

$$\text{-l:}\quad \frac{\Gamma, \varphi(y) \qquad \Delta}{\Gamma, \;\; x\varphi(x) \qquad \Delta},$$

$$\text{-r:}\quad \frac{\Gamma \qquad \Delta, \varphi(y)}{\Gamma \qquad \Delta, \;\; x\varphi(x)},$$

where t can be substituted for x in φ and y can be substituted for x in φ and y is not a free variable in $\Gamma \quad \Delta \quad \{ \;\; x\varphi\}$.

Proof in the sequent calculus. A *proof* in the sequent calculus is defined as a finite sequence of sequents where each sequent derives from the previous sequents and a rule of inference. A proof can be also defined as an oriented tree where each node is represented by a sequent. The leaf nodes represent the initial sequents and the remaining nodes derives from one or both children. The root of this tree is the final sequent.

Soundness and completeness. Sequent $\Gamma \quad \Delta$ is called *valid* if in every structure and assignment where all formulas from Γ are satisfied at least one formula from Δ is satisfied. In particular, if T is a finite theory and φ is a formula then φ is a logical consequence of T if and only if the sequent $T \quad \{\varphi\}$ is valid.

Let's state now two theorems. The proofs of these theorems are shown in Švejdar's textbook [16].

Every sequent provable in the Sequent calculus is valid.

Every valid sequent is provable in the Sequent calculus.

Cut elimination. Every sequent provable in the sequent calculus can be proven without the cut rule. The proof of this theorem can be find in Švejdar's textbook [16].

1.4 The programming language Elm

Elm is a powerful functional language which compiles to JavaScript. Elm has strong type checking and a well-tested compiler. Compared to JavaScript, pure Elm code is thus practically free of runtime exceptions. The key features of Elm are mentioned in the list below.

Immutability Everything in Elm is immutable, which means no value can be changed after assignment. When developers try to modify, for example, a record, Elm creates a copy of the record with the updated value.

Static types Elm is statically typed. However, type annotation is not required. The compiler can infer the type of a function by the arguments it receives.

Ports Elm and JavaScript can communicate by subscribing to ports for sending and receiving messages. This allows Elm programs to access the entire JavaScript ecosystem without losing the benefits of Elm.

Compiler The compiler has very useful and friendly error messages with hints. That makes compilation errors very easy to fix.

Packages Elm packages have strict version rules. If a declaration of an existing interface is changed, the package's major release number must change. Developers can thus safely upgrade package versions without the fear of breaking something.

In the following sections, we will demonstrate Elm, on a running example, which can evaluate boolean expressions.

1.4.1 The features of Elm

Data types. In most programming languages there are a couple of predefined data types which developers can use. In Elm, however, developers can easily define their own types, even recursive ones (by using keywords *type* and *type alias*) [2]. This encourages the creation of clean, type-safe code. A well-designed program then requires fewer unit-tests as invalid model states are forbidden by definition therefore they are impossible to occur. If a type error occurs, it is caught at compile time, unlike in dynamically typed languages such as JavaScript where type errors occur at runtime. Type alias is there to simplify the type annotations and make them easier to read. It is usually used to define the type of records. During compilation, *type alias* is simply replaced where it is used. Therefore, recursive annotation is forbidden as it would cause an infinitely large data type. Recursive types can be written by using *type* similarly to mathematics. The keyword *type* also brings the possibility of union types. A union type can connect a finite number of different types.

In Listing 1.1, we show the definition of the recursive union type – *Boolean*. Boolean is an expression built up by *conjuctions*, *disjunctions*, *negations* and constants *true* and *false*. We also show the non-recursive union type `Msg` which will be used later in the update function. Lastly, we show the definition of a record which will be used as our model. This record must contains only a boolean expression.

```
type Boolean
    = T
    | F
    | Not Boolean
    | And Boolean Boolean
    | Or Boolean Boolean

type Msg = Reset | Simplify
type alias Model = { expression : Boolean }
```

Listing 1.1: Definition of the used types

Pattern matching. There are two conditionals in Elm; `if` and `case`. Conditional `if` is used for conditions which evaluate to `Bool` and `case` is for matching union types. Union types are taken apart and their constituents can be treated separately. This enables the invocation of various functions based on the type we work with.

Listing 1.2 shows both types of conditionals. Function `boolToBoolean` converts the `Bool` to `Boolean` and function `evaluate` evaluates the boolean expression.

```
boolToBoolean : Bool -> Boolean
boolToBoolean bool = if bool then T else F

evaluate : Boolean -> Bool
evaluate expression =
    case expression of
        T -> True
        F -> False
        Not sub -> not <| evaluate sub
        And sub1 sub2 -> evaluate sub1 && evaluate sub2
        Or sub1 sub2 -> evaluate sub1 || evaluate sub2
```

Listing 1.2: Example of pattern matching

Higher-order helpers. Elm implements many higher-order helpers. Four of them are worth explicitly mentioning as they will be frequently used in our work.

Name	Syntax	Example	Meaning
Function composition	( )	$F \quad G$	$G(F(x))$
Function composition	( )	$F \quad G$	$F(G(x))$
Forward function application	(|>)	x |> F	$F(x)$
Backward function application	(<|)	F <| x	$F(x)$

In Listing 1.3, we show the composition of functions `evaluate` and `boolToBoolean`.

```
simplify : Boolean -> Boolean
simplify = evaluate >> boolToBoolean
```

Listing 1.3: Function composition example

1.4.2 The Elm Architecture and lifecycle

Elm has a Model-View-Update architecture. This architecture encourages modularity, simple code reuse, easy testing and is widespread in current modern web-development [2].

Model. Model is the state of the application. It stores all the data and is the only source of truth. The model is defined by types and the compiler verifies and enforces them.

In Listing 1.4, we show the initial state of our application containing the expression $(T \quad F) \quad \neg(T \quad F)$. The initial model's type was defined in Listing 1.1.

```
initialModel : Model
initialModel = { expression = And (Or T F) (Not <| And T F) }
```

Listing 1.4: Initial model of the application

View. View is the application renderer. A function which renders the whole application as HTML based on the data stored in the model. Elm views are just plain elm functions; every HTML element has its function which means the renderer has type checking. Whenever the update function is executed, elm lifecycle invokes the view function. In this function, we can set what messages should be dispatched on certain user action.

In Listing 1.5, we show how to render the HTML elements and how to show the value of our boolean expression. We also set up the buttons to dispatch `Simplify` and `Reset` messages when the buttons are clicked.

```
view : Model -> Html Msg
view model =
    div []
        [ text <| toString model.expression
        , button [ onClick Simplify ] [ text "Simplify" ]
        , button [ onClick Reset ] [ text "Reset" ]
        ]
```

Listing 1.5: Function rendering the HTML

Update. Update function modifies the application state based on a message it receives.

In Listing 1.6, we match the type of the message and either reset the initial application state or simplify the expression stored in our application.

```
update : Msg -> Model -> Model
update msg model =
    case msg of
        Reset -> initialModel
        Simplify -> { model | expression = simplify model.expression }
```

Listing 1.6: Update function simplifying the expression

Elm application lifecycle. The lifecycle is very simple. During initialization, the provided initial model is used by the view function to render the website. Whenever Elm notices any interaction, either from a user or by the port it had subscribed to, the update function is invoked. The update function receives the message type and the current model. Based on the message type, it updates and returns the model, which causes elm to rerender the website.

In Listing 1.7, we set up the Elm application lifecycle by defining the initial model and the functions which render and update the model. The code from the previous listing builds up a simple, working application shown in Figure 1.1.

```
main : Program Never Model Msg
main = Html.beginnerProgram
    { model = initialModel
    , view = view
    , update = update
    }
```

Listing 1.7: Initial program setup

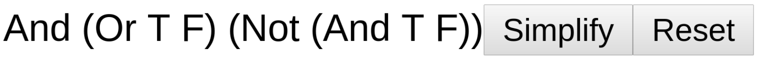


Figure 1.1: Application built in Section 1.4.

1.4.3 Formula parser

Computers have difficulties to understand natural languages. Therefore, in most cases, only formal languages are used. A formal language can be defined by an alphabet and a set of rules called a grammar. These rules define how the string can be formed from the language's alphabet with respect to the language's syntax. Syntactic analysis of a formal language is called parsing. To parse a formal language in elm the module `elm-parser` is used. It parses the formal language to abstract syntax tree. We need a parser to parse first-order formulas (1.1.1). The parser for this purpose was already implemented in Tableaux editor [15]. The definition of the grammar it can parse, in Backus-Naur form, is shown in Listing 1.8. To keep the listing readable, the self explanatory non-terminal symbols `digit` and `letter` are omitted.

```
symbol_and  ::= "&" | " " | "/\"
symbol_or   ::= "|" | " " | "\/"
symbol_impl ::= "->" | "→"
symbol_neg  ::= "-" | "¬" | "~"
symbol_universal   ::= " " | "\A" | "\forall" | "\a"
symbol_existential ::= " " | "\E" | "\exists" | "\e"

ident_character ::= <letter> | <digit> | "_"
identifier      ::= <letter> | <identifier> <ident_character>
spaces          ::= "" | " " <spaces>

atom ::= <identifier> | <identifier> <args>
term ::= <identifier> | <identifier> <args>
args ::= "(" <spaces> <term> <spaces> <nextArgMore> ")"
nextArgMore ::= "" | <nextArg> <nextArgMore>
nextArg     ::= "," <spaces> <term> <spaces>

sfs        ::= <spaces> <formula> <spaces>
quantified ::= <spaces> <identifier> <spaces> <formula>

formula ::= <atom>
          | <sfs>
          | "(" <sfs> ")"
          | "(" <sfs> <symbol_and> <sfs> ")"
          | "(" <sfs> <symbol_or> <sfs> ")"
          | "(" <sfs> <symbol_impl> <sfs> ")"
          | <symbol_neg> <spaces> <formula>
          | <symbol_universal>  <quantified>
          | <symbol_existential> <quantified>
```

Listing 1.8: Parser definition in Backus-Naur form

1.4.4 Zipper

Pure languages like Elm require special handling of certain problems. One of them, for example, is traveling in a tree.

Traveling in a tree. Traveling down the tree is straightforward. We access and return the child node. Can we travel back to the root? In pure languages it is not possible. When we travel down, we throw away the parent and at that point, we have only the child. To solve this problem, the child node needs to have a reference to the parent. Elm, however, has immutable data structures, so it is not possible to create such a structure. Let's illustrate the problem in a few steps:

1. Create a child with `Nothing` for its parent.

2. Create a parent that points at the child.

3. Update the child to point at the parent.

Step three is, however, impossible. When you have immutable data structures, modification means, creating a new child with an updated value, which has a new reference. The parent, therefore, still points to the original child. It means we need some other way to move down in the tree.

Keeping a path to the node. Instead of moving down the tree and throwing away the parent, we will remember how to get to the node. That means we can access arbitrary nodes in the tree at any time.

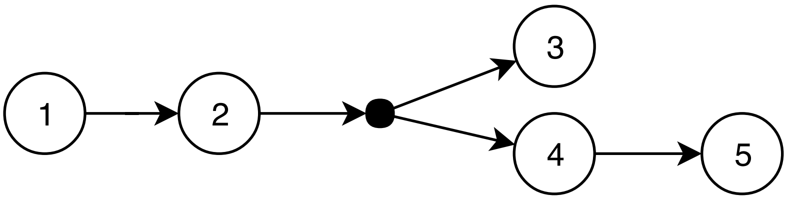


Figure 1.2: Example of a tree

Let's demonstrate this in Figure 1.2. When we want to modify the node 3, we store a path how to access it. So we store the whole tree and the sequence “next step, next step, case1”. To modify the node 5, we must travel there. We can go up by dropping the last step of our sequence (now we have “next step, next step”). At this point, we point to the cases node. Adding “case2, next step” to the previous sequence we get the path to node 5. We can see this allows us to travel the tree in any direction. In case of sub-trees, we would need another type of step, which we could call “visit sub-tree”.

This approach allows us to access any element in $O(n)$ time. A faster way for moving around would be using a zipper.

Zipper. Zipper, basically, introduces a data state. The data state allows us to point to arbitrary node in the tree. It means we can access the current node seamlessly. A zipper for a tree is created in a way, that it is possible to move to any direction and therefore to point to any node. Moving around takes $O(n)$ time, however, once we point to the node we want to modify, we can do so in $O(1)$ time. Zipper has two basic building blocks. The first one is the node (with the sub-tree) which the zipper points at and the second is information about reconstructing the rest of the tree [10]. A piece of information about reconstruction is called a breadcrumb.

1.5 Related work

There are a couple of proof assistants similar to our vision. However, none of them meets all of our requirements. Let's introduce those proof assistants and demonstrate them on a proof of first-order De Morgan rule.

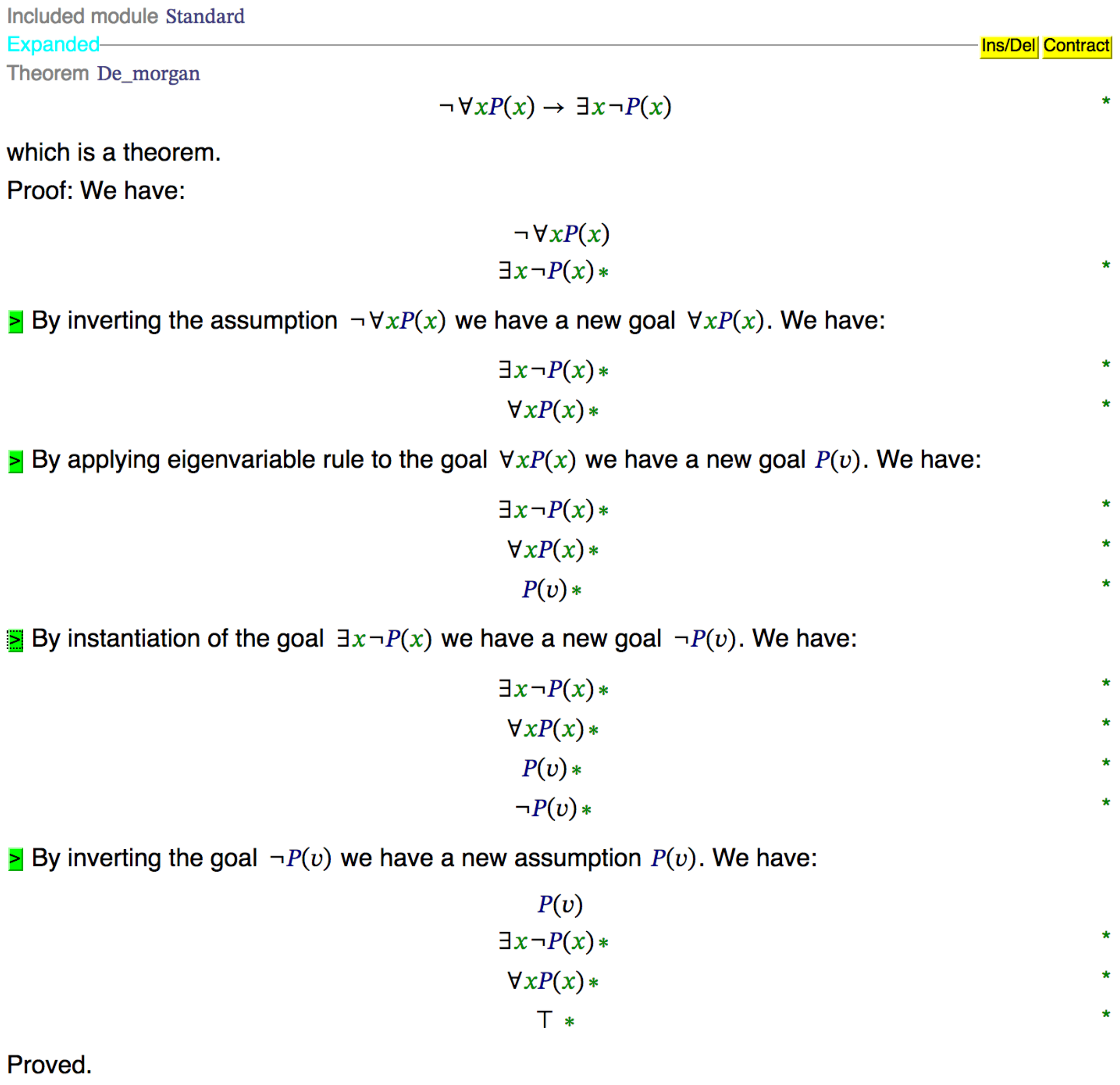
1.5.1 Isabelle/Isar

Isabelle/Isar is a semi-automated proving system [14] supporting proof development. The tool is very powerful and it has no graphical user interface, but comes paired with an IDE. One can make proofs by programming them in a special syntax called Isar. The learning curve for its syntax is quite steep. Therefore this tool is not very suitable for education, especially in the first year at a university. The following listing shows the proof of a De Morgan rule in Isar.

```
lemma de_Morgan:
  assumes "¬ (  x. P x)"
  shows "  x. ¬ P x"
proof (rule classical)
  assume "¬  x. ¬ P x"
  have "  x. P x"
  proof
    fix x show "P x"
    proof (rule classical)
      assume "¬ P x"
      then have "  x. ¬ P x" ..
      with ‹¬  x. ¬ P x› show ?thesis by contradiction
    qed
  qed
  with ‹¬(  x. P x)› show ?thesis by contradiction
qed
```

1.5.2 Clausal Language (CL)

CL is a declarative programming language used since 1997 on Comenius University [7], built on a logical basis of Peano arithmetics. The CL proof assistant uses a variation of semantic tableaux with signed formulas and several derived rules. It is a desktop application which uses web-browser for rendering the proof. Similarly to Isabelle,



Included module Standard

Theorem De_morgan

$$\neg\forall x P(x) \rightarrow \exists x \neg P(x)$$

which is a theorem.

Proof: We have:

$$\neg\forall x P(x)$$
$$\exists x \neg P(x)*$$

> By inverting the assumption $\neg\forall x P(x)$ we have a new goal $\forall x P(x)$. We have:

$$\exists x \neg P(x)*$$
$$\forall x P(x)*$$

> By applying eigenvariable rule to the goal $\forall x P(x)$ we have a new goal $P(v)$. We have:

$$\exists x \neg P(x)*$$
$$\forall x P(x)*$$
$$P(v)*$$

> By instantiation of the goal $\exists x \neg P(x)$ we have a new goal $\neg P(v)$. We have:

$$\exists x \neg P(x)*$$
$$\forall x P(x)*$$
$$P(v)*$$
$$\neg P(v)*$$

> By inverting the goal $\neg P(v)$ we have a new assumption $P(v)$. We have:

$$P(v)$$
$$\exists x \neg P(x)*$$
$$\forall x P(x)*$$
$$\top *$$

Proved.

Figure 1.3: Proof of the De Morgan rule in CL proof assitant

the CL proof assistant is also very powerful and uses a special syntax to command the assistant to apply inference rules.The tool was used in teaching for almost two decades. However, it is based on outdated technologies and there was no significant development in recent years. Also, students reportedly [7] used to have difficulties understanding the automatic inferences made by the assistant and some students used to force it to solve their homework, without understanding what they were doing. Proof are also difficult to edit (e.g., no insertion between two existing steps, deleting a step deletes everything bellow it). These reasons make it difficult to use it in new courses.

Figure 1.3 shows the proof of De Morgan rule in the CL proof assistant. To prove this rule it is necessary to write to input boxes the following commands:

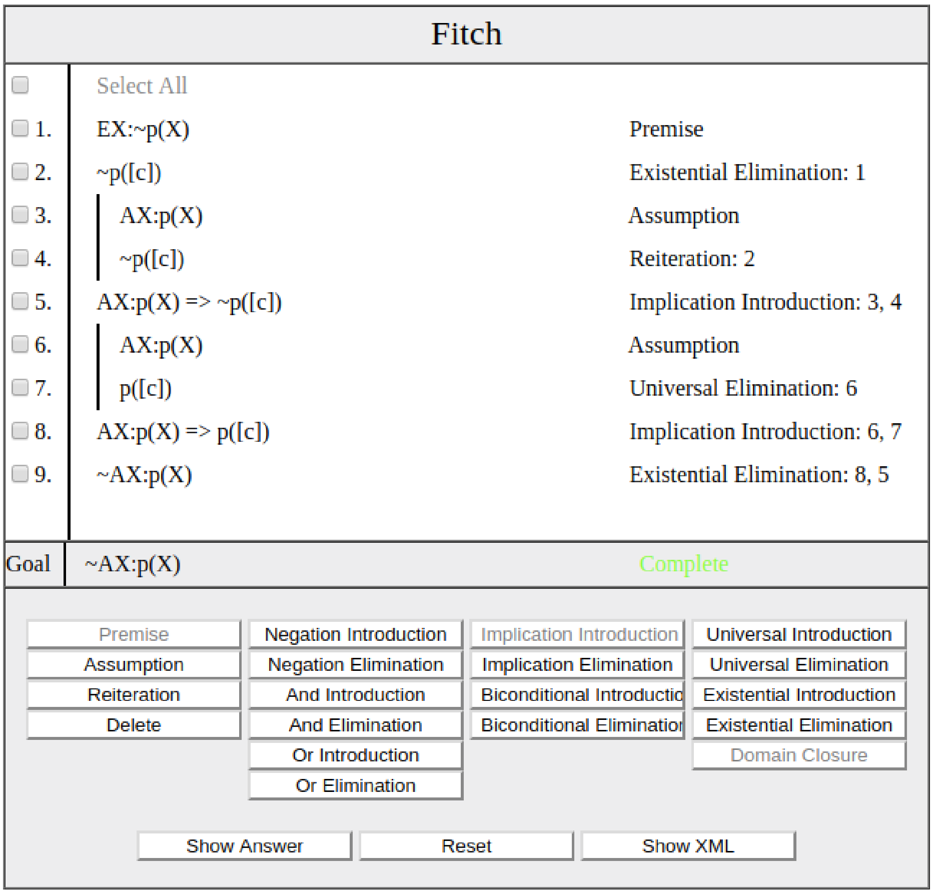
```
thm De_morgan
  ~\a xP(x) -> \e x~P(x)
```

```
proof
 inv ~\a xP(x)
  eigen* \a xP(x); v
   inst* \e x~P(x); v
    inv* ~P(v) proved.....
```

1.5.3 Fitch

The Fitch system [13] is a proof assistant used in logic course from Stanford University. It is based on the Fitch calculus. The rules of inference are the main building block of the proofs. The user selects premises and a rule and the assistant derives its conclusion. This lead to students focusing on the rules instead of the desired conclusion. We would also like to have a more intuitive and interactive user interface.

Figure 1.4 shows the proof of a De Morgan rule in this proof assistant.



Fitch

□ Select All

□ 1. EX:~p(X) Premise

□ 2. ~p([c]) Existential Elimination: 1

□ 3. | AX:p(X) Assumption

□ 4. | ~p([c]) Reiteration: 2

□ 5. AX:p(X) => ~p([c]) Implication Introduction: 3, 4

□ 6. | AX:p(X) Assumption

□ 7. | p([c]) Universal Elimination: 6

□ 8. AX:p(X) => p([c]) Implication Introduction: 6, 7

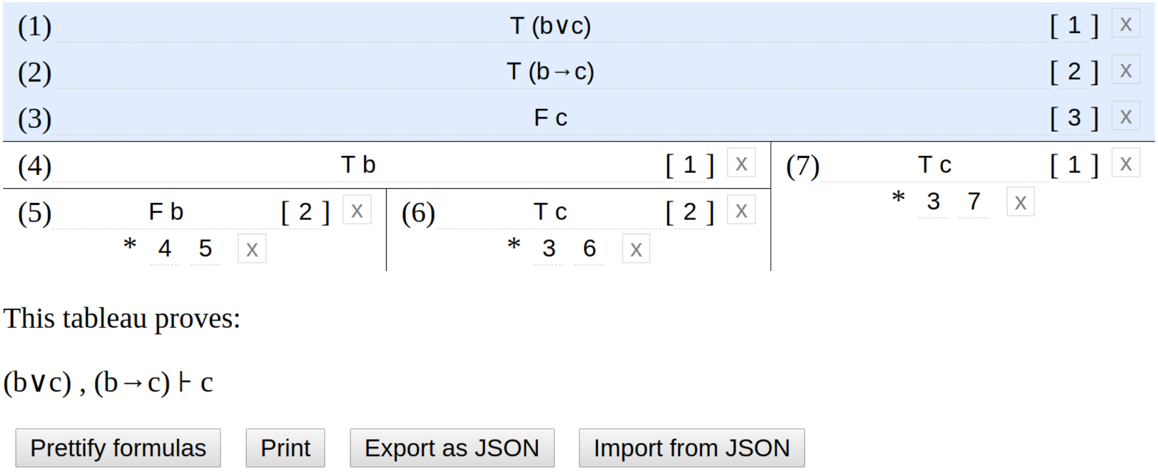
□ 9. ~AX:p(X) Existential Elimination: 8, 5

Goal ~AX:p(X) Complete

Figure 1.4: Proof of the De Morgan rule in Fitch

1.5.4 Tableaux editor

Tableaux editor [15] is a proof assistant for building Smullyan's tableau proofs. It was developed at Comenius University and is used for teaching Mathematics (4) – Logic for Computer Science. The editor works with signed formulas and checks the reasoning between steps. Currently, it does not support first-order logic. A simple proof that shows $\{(b \quad c), (b \to c)\} \quad c$ is shown in Figure 1.5.



(1) T (b∨c) [1] x

(2) T (b→c) [2] x

(3) F c [3] x

(4) T b [1] x

(5) F b [2] x
* 4 5 x

(6) T c [2] x
* 3 6 x

(7) T c [1] x
* 3 7 x

This tableau proves:

(b∨c) , (b→c) ⊦ c

Figure 1.5: An example of a proof in Tableau editor

Analytic tableaux are not the way we would like to approach our problem so this tools as is, is not very relevant for us. Expect for the fact that it is a client-side web application written in Elm.

Chapter 2

Design

This chapter describes the overall design of the proof assistant. Section 2.1 sets the objectives and the scope of this thesis. Section 2.2 describes the proof representation. Section 2.3 describes how the proof validation works and Section 2.4 shows the user interface.

2.1 Requirements analysis

The proof assistant which we are going to build has a couple of requirements. Some requirements were clear from the beginning but some were not so obvious and were shaped later during consultations.

The proof assistant must be easily accessible and deployable. Preferably, it should be a client side web application so students can access it directly in their browsers. A client side web application would fulfill both requirements and we could easily deploy and host our application on GitHub Pages [4]. The proof assistant must be reasonably fast and interactive.

The assistant should support linear proofs similar to Hilbert calculus but instead of applying modus ponens to the previous formulas we want to write the logical consequences of those formulas.

We also want to be able to state a lemma which we can prove in its sub-proof. This would be equivalent to the *cut* rule in Sequent calculus. When we have a formula which is a disjunction, we want to prove the consequences of the disjuncts separately. This is called proof by cases and it would be equivalent to -r rule in Sequent calculus. We also want to be able to make a proof by contradiction. This would be a variation to the *cut* rule in Sequent calculus.

The assistant should not be automated, students must make elementary steps which the assistant must check and inform about the correctness of those steps.

Students should be able to easily interact with the proof assistant, modify the proof, save the current state, load it and be able to return to the previous states.

2.2 Proof representation

In order to support the required types of proof (Section 2.1) we represent the proof as a tree. The tree has two types of nodes: formula node and cases node.

A ***formula node*** has zero or one child nodes. The child node represents the next step in the proof. A formula node can contain a sub-proof which helps to prove the given formula. A sub-proof does not fork the tree.

A ***cases node*** forks the tree. It has exactly two child nodes.

A formula nodes requires an ***explanation***. The explanation gives reason why the formula is true. The reasons can be that the formula is

- a premise,
- a logical consequence of other formulas on the branch, derivable by a single built-in rule,
- a goal (or lemma) which will be proven in a sub-proof,
- proven by contradiction in a sub-proof,
- is a first-order formula, created by generalization, which will be proven in a sub-proof.

The formulas proven in sub-proofs will be available to use in the main branch.

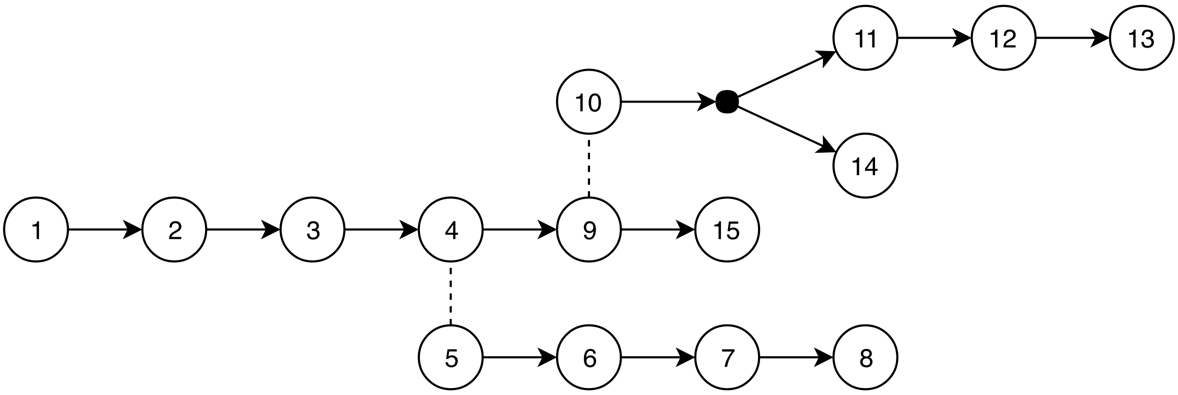


Figure 2.1: Proof tree example

An example of the structure of a proof is shown in Figure 2.1. On the primary level, there is only one branch 1 − 2 − 3 − 4 − 9 − 15. Nodes 4 and 9 have sub-proofs. Node 4 therefore can be proven by branch 1 − 2 − 3 − 5 − 6 − 7 − 8. The black circle represents a cases node. The sub-branch of node 9 is therefore forked in the second step and must be proven in both 1 − 2 − 3 − 4 − 10 − 11 − 12 − 13 and 1 − 2 − 3 − 4 − 10 − 14 branches.

2.3 Validation

The main task of the proof assistant is proof validation. Validation means checking the syntax of a formula and checking the logical correctness of each step. Every explanation type mentioned in Section 2.2 need to have a well-formed formula. Therefore, firstly we check the syntax of the formula by parsing it. If the formula is well-formed, we can parse it to an abstract syntax tree and run additional validators. If the formula cannot be parsed, we show the parsing error message and do not run any additional validator. Different explanation types have different additional validators. The validators for every explanation type are shown in the list below.:

Premise Premise does not have any additional validation as it is assumed to be correct. Problem assignments will usually be given as a set of premises.

Rule The rule validator checks that the formula is a consequence of one of the inference rules and the formulas in the current branch above it.

Goal The goal validator checks whether the goal is proven in its sub-tree. The goal is proven if we show, by deductive reasoning, that it is valid in every branch of its sub-tree. The goal node is valid when we either find its formula in the branch or when there is a contradiction in the branch.

Contradiction The contradiction validator adds the contradicted formula to the branch and removes the original one. Then, the validator check whether the explanation is correct the same way as the goal validator.

Cases The cases validator checks that the cases node derives from a formula of type $(A \quad B)$. One formula in cases node must be A and the other B, otherwise the explanation is not valid.

Soundness and completeness. These two properties are required in our proof assistant. Completeness can be achieved by implementing the Hilbert system we described in Section 1.3.2. To fully support propositional logic, we must implement the seven basic axioms and the rule *modus ponens*.

To support first-order logic, we decided to implement the axioms B_1 and B_2 as rules of inference:

$$B_1\colon \frac{x(\varphi)}{\varphi(x/t)}, \qquad B_2\colon \frac{\varphi(x/t)}{x\varphi}.$$

These rules will also check that variable x can by subtituted by t in φ.

The rules of inference Gen-A and Gen-E will be treated specially. Gen-A introduces a universal quantifier. In order to introduce a universal quantifier $x\varphi$, the users will have to prove, in a sub-proof, the formula $\varphi(x/y)$ where y is a new free variable. This is similar to the sequent calculus: Formulas on the branch where we want to introduce the universal quantifier can be considered as the antecedent of a sequent and the formula $x\varphi$ can be considered a member of the succedent. Our treatment of the quantifier is then analogous to the rule -r.

Gen-E eliminates an existential quantifier. It will be another special rule of inference. The rule will receive a premise and a list of free variables in the current branch. The existential quantifier can be removed and the bound variable can be substituted with some variable if this variable does not occur free anywhere in the branch. This is analogous to the rule -l.

We can now implement this hybrid formal system which should be sound and complete.

On the one hand, this basic system brings soundness and completeness, on the other hand, it is not very practical for real-life use. We, therefore, implement the rules of inference from table 2.1 to extend our tool's usability. Adding new rules of inference to our system does not change the completeness, however, it can break the soundness. All the rules we implement can be proven to be sound and, therefore, our system will remain sound. We also add the rules derived from equivalences in Table 2.2.

2.4 User interface & user interaction

The main purpose of the user interface is to build a tree representing a proof. This includes: adding new nodes, removing nodes, changing the explanation type of formulas and modifying the formulas. The proof assistant will assist in this process by giving feedback about the validity of the proof, validity of the formula's syntax, traveling back in time, persistence etc. Let's now describe some qualities, which are required for our proof assistant to be effective and easy to use.

Simple to use The learning curve for most of the tools mentioned in Section 1.5 is very steep. Our goal is to make a good UX so the users can use our application without reading manuals. Anyone should feel comfortable using it, just by checking out some basic examples.

Minimalist Minimalism is key for a good UIs. Too many buttons or options would distract the user from the proof, and it would, therefore, consume their time.

Powerful Minimalist does not mean idle. The users should be able to do most of the work they need, such as deleting an incorrect proof branch, changing the proof type, etc.

Now let's go through the user interface and specify how the user will interact with the proof assistant.

Law of the excluded middle	$\dfrac{}{P \quad \neg P}$
Addition	$\dfrac{P}{P \quad Q}$
Conjunction	$\dfrac{P \quad Q}{P \quad Q}$
Simplification	$\dfrac{P \quad Q}{P}$
Implication introduction (2)	$\dfrac{Q}{P \rightarrow Q}$
Implication introduction (3)	$\dfrac{\neg P}{P \rightarrow Q}$
Modus Ponens	$\dfrac{(P \rightarrow Q) \quad P}{Q}$
Modus Tollens	$\dfrac{(P \rightarrow Q) \quad \neg Q}{\neg P}$
Disjunctive Syllogism	$\dfrac{\neg P \quad (P \quad Q)}{Q}$
Hypothetical Syllogism	$\dfrac{(P \rightarrow Q) \quad (Q \rightarrow R)}{(P \rightarrow R)}$
Constructive Dilemma	$\dfrac{((P \rightarrow Q) \quad (R \rightarrow S)) \quad (P \quad R)}{(Q \quad S)}$
Destructive Dilemma	$\dfrac{((P \rightarrow Q) \quad (R \rightarrow S)) \quad (\neg Q \quad \neg S)}{(\neg P \quad \neg R)}$
Proof by contradiction	$\dfrac{(\neg P \rightarrow Q) \quad (\neg P \rightarrow \neg Q)}{P}$
Proof by cases	$\dfrac{(P \rightarrow R) \quad (Q \rightarrow R)}{((P \quad Q) \rightarrow R)}$

Table 2.1: Table of the rules of inference to be implemented

2.4.1 User interaction

The proof assistant must have a minimalist and modern user interface.

All of the action buttons directly modifying the proof will be located next to the nodes themselves. This will lead to simpler user interaction but will have negative impact on user experience. For that reason, the action buttons will be by default shown only on the last added node and others will be hidden. It will be, however, possible to show all the action buttons for a node if the user needs it.

Implication introduction	$(\neg p \quad q) \Leftrightarrow (p \rightarrow q)$
Double negation	$\neg\neg p \Leftrightarrow p$
Commutative	$(p \quad q) \Leftrightarrow (q \quad p)$
	$(p \quad q) \Leftrightarrow (q \quad p)$
Idempotency	$(p \quad p) \Leftrightarrow p$
	$(p \quad p) \Leftrightarrow p$
De Morgan rule	$\neg(p \quad r) \Leftrightarrow (\neg p \quad \neg r)$
	$\neg(p \quad r) \Leftrightarrow (\neg p \quad \neg r)$
Associativity	$(p \quad (q \quad r)) \Leftrightarrow ((p \quad q) \quad r)$
	$(p \quad (q \quad r)) \Leftrightarrow ((p \quad q) \quad r)$
Distributivity	$(p \quad (q \quad r)) \Leftrightarrow ((p \quad r) \quad (p \quad r))$
	$(p \quad (q \quad r)) \Leftrightarrow ((p \quad r) \quad (p \quad r))$

Table 2.2: Rules derived from equivalences

When the user clicks on an action button or changes the formula, the changes will be made immediately without confirmation. This will lead to better user interaction and give immediate user feedback.

Formulas. The user will write the formulas to input boxes. This means that we will have all the formulas in plain text and we will have to parse them. The parser details are explained in Backus-Naur form in Section 1.4.3. There are multiple ways to write logical connectives in plain text therefore we will not have any buttons for them.

Type of explanations. As there will be several types of explanations, the user should be able to change and see the chosen explanation easily. This could be achieved using a dropdown menu, however it would require two clicks to change the explanation. Therefore, we will implement this functionality using radio buttons. The options, for the radio buttons, are premise, rule, goal, and contradiction.

Adding next step. Since proofs consist of two types of nodes, there need to be two types of buttons to add a next step to the existing node. First one will add a formula node. This button should be available in every node and should insert an empty node below the current one. The second type of button will add cases node. Inserting a cases node in the middle of the tree would require to remove the current node's entire sub-tree or to set the entire sub-tree as one of the cases. Cases node is however valid only if it derives from a formula of type $A \quad B$. For this reason, the cases node button will be available only in leaf nodes.

Deletion We not only need to add nodes, but we have to delete them as well. Deletion is a little more tricky. We must determine what behavior would the user expect to happen. We defined our delete action as follows: when the user deletes a regular node, the node along with its sub-proof, if there is one, is deleted but the successor nodes remain untouched. On the other hand, when the user deletes a cases node, the entire node along with both its sub-trees are deleted.

Hide sub-proof. Some proof types such as goal or contradiction have a sub-proof – where the formula is proven. This allows the user to state lemmata and prove them separately. However, this introduces a lot of GUI complexity as the whole proof is visible and reduces the focus from the original problem. An easy solution would be to allow the user to hide the sub-proofs.

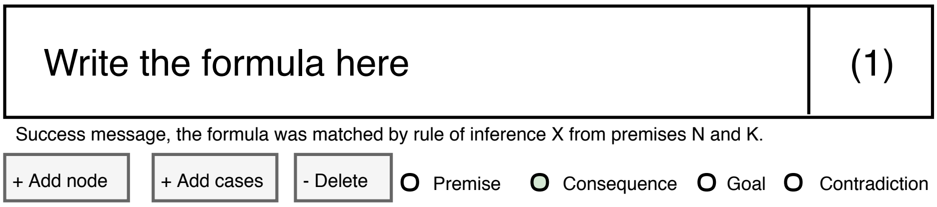
2.4.2 User feedback

The secondary purpose of the UI is to give the students feedback on their proofs. This includes success and error messages. Success messages are information about the correctness of the explanation in individual nodes. Error messages are messages like parsing errors or information that the formula is not yet proven.

Numbering. Each proof step should have a unique identifier, for example, an integer. This will make possible for our proof assistant to show which rule of inference with which premises were used for validation of the current node. Showing this information will make our assistant more educational.

Error and Success messages. There are many possible approaches where and how to show the error or success messages. We were deciding between the following: The first option was to aggregate them on the top of the page. This would make the user interface less messy but hard to read in huge proofs, with tens of steps. The second option was to show every message next to the formula it belongs to. To make the user interface minimalist, the messages must be compact. We decided to combine these two options. We will show every message next to the node it belongs to and show a success or error message on the top of the page base on whether everything is valid or not. The reason is that the messages next to the node are within the node's context and on the top of the page we want to have the information that everything is valid therefore the goal was proven.

The wireframe in Figure 2.2 shows how a node in the proof will look like. The figure contains both elements for interaction with the tree and the given feedback by the proof assistant.



Success message, the formula was matched by rule of inference X from premises N and K.

Figure 2.2: One node in the proof

2.4.3 History and Persistence

State history As some of the user actions are destructive, such as deleting a goal node or changing its type to a regular node, we need to have some mechanism to prevent the user from loosing half of the proof by accident. We can create an "are you sure" dialog window which would force the users to confirm their action. This dialog would alleviate the problem. However, it would make the interaction with the proof assistant less fluent, interrupted by confirmation dialogs. For this reason, we decided to create a state history. In case the user makes some destructive action by accident, he can undo the action, returning to the previous state. We will show the user undo and redo buttons on the top of the page.

Persistence A proof assistant without persistence would not be helpful. If we closed the window, we would lose our proof. This is a problem for a couple of reason. We cannot come back to our proof later or send the proof to the teacher. We will, therefore, serialize our application state to a JSON object. We decided for JSON format because it is widely supported among programming languages. Serializing and deserializing the state will allow us to save and load the application state at any stage. We will add save and load button on the top of the page. The save functionality will allow the teacher to create an assignment with premises and goals and give it out to students. The students then could load the assignment, solve it, save it and return it to the teacher.

Chapter 3

Implementation

We have decided to implement our proof assistant in Elm. This decision was made for a couple of reasons. First, mathematical logic is close to functional programming. Second, some parts of the proof assistant, such as the formula parser, have already been implemented in this programming language.

In this chapter, we describe the code structure of our proof assistant (Section 3.1) and the parser we use for parsing formulas (Section 3.2). We show the structure of the proof tree and explanation along with the zipper, which is heavily used (Section 3.3). We also describe the validation and how the rules of inference are implemented (Section 3.4). How we implemented state history (Section 3.5) and persistence (Section 3.6) and lastly, what framework we used for the user interface (Section 3.7). The source code of our proof assistant can be found in Appendix A on page 43.

3.1 Code structure

We designed our code to have the structure shown in Listing 3.1.

```
index.html
styles.css
src/
- Core/
- - Matcher.elm
- - Proof.elm
- - Types.elm
- - Validator.elm
- Exporting/
- - Json/
- - - Decode.elm
- - - Encode.elm
- - Ports.elm
- Editor.elm
- History.elm
- Main.elm
- Zipper.elm
tests/
```

Listing 3.1: Code structure

/index.html contains the HTML skeleton which imports the styles and the compiled JavaScript. In this file, we also define the JavaScript bindings which are used in Elm for the persistence layer.

/styles.css contains the application styles which overrides the Bootstrap design.

/src/Main.elm contains code which connects all the parts of the application.

/src/Editor.elm contains all the renderers, which render the HTML for the tool.

/src/Core/ contains the core of the applications such as the representation of the proof tree, data types we use in our tool, validators and matchers.

/src/Exporting/ contains the persistent layer of our application including JSON serializers, deserializers, and ports which define the communication with native JavaScript.

/src/History.elm contains code for the application history. It implements the functionality of Undo & Redo buttons.

/src/Zipper.elm contains a proof tree wrapper for moving around in the tree.

/tests/ contains the unit-tests for application logic, especially for matchers.

3.2 Formula parser

The tableau editor [15] already had a parser implemented in Elm. Therefore, making a new parser would be re-implementing the wheel. We have decided to use the one from the tableau editor.

The editor itself is currently being upgraded – see the bachelor thesis by Alexandra Nyitraiová [11]. Having the parser's code in two separate projects would make it harder to maintain. Therefore, we have decided to move it to a separate package, publish it on GitHub [9] and also upload it to Elm packages [8]. This allows us to reuse it seamlessly in other projects in the future.

In order to make the contribution to the package easier for future contributors, we have decided to set up a continuous integration server – Travis. Travis checks that the tests in the parser still pass thus the code is not broken. Travis also checks the code formatting, by running the automatic code formatter *elm-format*.

Elm packages are usually uploaded to `http://package.elm-lang.org`. To keep the uploaded packages quality high, Elm packages enforce the maintainers to write documentation for every exposed function in the module. For this reason, we had to document all the exposed functions.

3.3 Proof tree

We defined our proof tree as a recursive data type `Proof`. To match our proof representation in Section 2.2, we defined it as shown in Listing 3.2.

```
type Proof
    = FormulaNode Explanation FormulaStep (Maybe Proof)
    | CasesNode FormulaStep (Maybe Proof) FormulaStep (Maybe Proof)

type Explanation
    = Premise
    | Rule (Maybe Justification)
    | Goal (Maybe Proof)
    | Contradiction (Maybe Proof)
    | Generalization String (Maybe Proof)
```

Listing 3.2: Proof tree definition

`FormulaNode` represents a single node in the tree (step in the proof). It has an explanation, which explains why the step is correct, it has a `FormulaStep` which contains data about the step such as plain text and parsed formula, and it has a `Maybe Proof` which refers to the next step in the proof. The reason why the next step is optional is that leaf nodes do not have a successor.

Cases node represents a point in the tree where the proof forks. It has two `FormulaStep`s as the cases nodes has two successors, and it also has two `Maybe Proof` as those successors can have next steps.

Zipper. To travel in the tree we use a zipper. Our definition of zipper is shown in Listing 3.3. This definition allows us to fully reconstruct the tree from any particular node.

```
type alias Zipper =
    { proof : Proof, breadcrumbs : List Breadcrumb }

type Breadcrumb
    = GoDown Explanation FormulaStep
    | GoCase1 FormulaStep FormulaStep (Maybe Proof)
    | GoCase2 FormulaStep (Maybe Proof) FormulaStep
    | GoContradiction FormulaStep (Maybe Proof)
    | GoGoalProof FormulaStep (Maybe Proof)
    | GoAddUniversal FormulaStep String (Maybe Proof)
```

Listing 3.3: Zipper definition

Now, when we want to move down in the tree, we store the successor node in the `proof` and add the appropriate breadcrumb to `breadcrumbs`. Traveling up in the tree would mean to recover the parent node. Example functions for traveling down and up in the tree using our zipper is shown in Listing 3.4.

```
down zipper =
  case zipper.proof of
    FormulaNode exp data next ->
      Maybe.map (\nextProof ->
        { zipper
        | proof = nextProof
        , breadcrumbs = GoDown exp data :: zipper.breadcrumbs
        }) next
    _ -> Nothing

up zipper =
  case zipper.breadcrumbs of
    breadcrumb :: rest ->
      case breadcrumb of
         GoDown expl data ->
           Just
             { zipper
             | proof = FormulaNode expl data (Just zipper.proof)
             , breadcrumbs = rest }
         ...
    [] -> Nothing
```

Listing 3.4: Functions for traveling in zipper

3.4 Validation

Let's go through how we have implemented the validation designed in Section 2.3. Validation is defined in two files, `Core/Matchers.elm` and `Core/Validator.elm`.

Matchers. Matchers are simple functions which determine whether the formula can be derived by an inference rule from the previous formulas. We define three types of matchers, nullary, unary, and binary.

Nullary matchers. Nullary matchers are used for axioms and tautologies. Nullary matchers take only the formula which needs to be matched. For this reason, each matcher is executed only once for a particular node.

Unary matchers. Unary matchers implement inference rules with one premise. Unary matchers take the formula which needs to be matched and one formula from the branch. That means, we execute the same matcher n times, where n is the size of the branch.

Binary matchers. Most of the rules of inference have two premises, and are thus implemented as binary matchers. Binary matchers take the formula which needs to be matched and two different formulas from the branch. That means, we execute the same matcher n^2 times, where n is the size of the branch.

Validator. Each time when we change a step in our proof, either by changing the formula or by changing the type of the step we rerun all the matchers for all the steps which could be affected. The affected steps must be on the current branch.

3.5 History

We have two choices to implement state history (Section 2.4.3). We can either use an existing package or create our own. In languages with immutable data structures, creating the undo/redo functionality is trivial. We have chosen to implement our own because implementing it has taken only an hour, it is only 70 lines of code and is customizable to our needs. We need the following properties:

1. Save the current state to the history.
2. Replace the current state in the history – editing text in the formula should not make a new entry.
3. Go back to the previous state.
4. Go forward to the next state if we went back before.

Custom implementation. The structure shown in Listing 3.5 can fulfill all of our requirements. To save the current state, we move the old state to `prev` list, we drop the `next` list and save the state to `current`. To replace the current state, we replace it in `current`, this operation will thus not make a new entry in history. To go back, we move the `current` to the next list and move the last state from `prev` list to `current`. To go forward, we move `current` to `prev` list and move the first state in `next` list to `current`. Actions number one and number two are invoked on user actions which are modifying the proof tree. Actions number three and four when the user clicks to the undo/redo buttons.

```
type alias History m =
    { current : m
    , prev : List m
    , next : List m
    }
```

Listing 3.5: History structure

3.6 Persistence

As mentioned in Section 2.4.3 our tool without persistence would be useless. We decided to create a separate module called *Exporting*. This module contains three files: `Encode.elm`, `Decode.elm` and `Ports.elm`.

Saving the proof. Saving the proof requires encoding the application state. Elm has a core module for encoding the model to JSON. This module, however, works out of the box only with primitive data types. To be able to serialize custom types, we must write custom type serializers. These serializers define the format of the serialized JSON data. To save the data, we use a link which has set the `data` source and the HTML attribute `download`.

Importing the proof. Importing the proof requires decoding the uploaded JSON file to application state. In Elm, it is not possible to access the uploaded files directly. We will, therefore, read the data in JavaScript and send it to Elm. Ports can accomplish communication between Elm and JavaScript. We have two ports, `fileSelected` – to let the JavaScript know when the user clicks the button which shows a file dialog and `fileContentRead` – to send the selected file content back to Elm. At this point, we need to deserialize the data using our custom deserializers. If the deserialization succeeds we replace the application state with the uploaded serialized state.

3.7 User interface

In order to obtain a contemporary visual design of the user interface we have decided to use Twitter Bootstrap v4. Instead of writing custom HTML elements with classes and much boilerplate HTML (as it is usually done), we decided to use a module for this. The `elm-bootstrap` package aims to make it pleasant and reasonably type safe to use Bootstrap in Elm applications.

Bootstrap is very useful and saves us much work; however, it has some shortcomings. One of them is a bug which does not allow to have a validation message on input groups[1]. We had to fix this issue on our own and we also wanted to have a simple button for proof import. For these reasons, we had to write some custom CSS which are contained in the file `/styles.css`.

[1] See the issue on Github: `https://github.com/twbs/bootstrap/issues/23454`.

Chapter 4

Evaluation

This chapter describes the process of evaluation and summarizes how the students reacted to our proof assistant. It also shows the students' feedback to the assistant and describes the next steps we have taken to implement it.

4.1 Goals and processes

This section describes the goals of the evaluation, explains the process of the evaluation and shows the assignments given to the students.

Goal of evaluation. The goal of the evaluation was to obtain feedback on user interface and usability. We were particularly interested how the students will able to use the tool by themselves.

Process of evaluation. At the beginning of the evaluation, we gave out a simple assignment to the students. When they solved it, we gave them a new one. The difficulty of those problem assignments was increasing in time. The students were supposed to solve the problems on their own. We helped them only when they got stuck. At the end of the evaluation, we conducted an interview with the following five questions (originally by Milan Cifra [1]):

1. What would you use the tool for?
2. How did the tool help you?
3. Was the user interface intuitive?
4. Would you recommend the tool to someone?
5. What new functionality would you like to see?

Assignments and observations. We have given the students five assignments in total.

Assignment 1: $\{a \rightarrow b, b \rightarrow c, a\} \models c$

The assignment tested that the students can create a linear proof and change the explanation types of formulas. We observed that they did not know how to write the set of formulas to the proof assistant. When we told the students that they need to write one formula per line and they need to mark them as premise, they showed that c holds easily.

Assignment 2: $\{a \rightarrow b, a \rightarrow \neg b\} \models \neg a$

This assignment tested the ease of using proof by contradiction in the proof assistant. We observed that the students are not confident with proving by contradiction. When they were told to use proof by contradiction, they managed to solve the problem.

Assignment 3: $\{a \rightarrow \neg b\} \models (\neg a \quad \neg a) \quad \neg b$

The assignment tested proof by cases. Most of the students, however, proved it by contradiction. We asked them to prove it again by cases. A few were struggling because of a non-informative error message.

Assignment 4: $\{ \; xP(x), \; x(P(x) \rightarrow Q(x))\} \models \; xQ(x)$

The purpose of this assignment was to try to prove something in first-order logic. It was clear to the students that they can use substitution, but they did not know how to prove it. We told them to use the explanation type generalization. This hint was sufficient as they managed to create the proof quickly.

Assignment 5 (by Gordon S. Novak Jr. [5]):

1. All hounds howl at night.
2. Anyone who has any cats will not have any mice.
3. Light sleepers do not have anything which howls at night.
4. John has either a cat or a hound.
5. (Conclusion) If John is a light sleeper, then John does not have any mice.

The purpose of this assignment was to apply the skills learned in the previous assignments to a more complex problem. The students got the propositions already formalized as we did not want to test their skills in formalization. The formalized propositions are shown below:

1. $x(hound(x) \rightarrow howl(x))$
2. $x(\; m(has(x,m) \quad cat(m)) \rightarrow \neg \; m(has(x,m) \quad mice(m)))$
3. $x(light\_sleeper(x) \rightarrow \neg \; y(has(x,y) \quad howl(y)))$

4. $z(has(John, z) \quad (cat(z) \quad hound(z))$

5. $light\_sleeper(John) \rightarrow \neg \quad m(has(John, m) \quad mice(m)))$

Figure 4.1 shows a proof of the example proven in our proof assistant (margins and padding were modified so the screenshot can fit on one page).

4.2 Feedback from students

We tested the proof assistant with ten students of applied informatics with different knowledge of mathematics. We were testing the user experience of the assistant and collecting feedback on usability. After the testing, we asked the students five questions. The summarized answers are in Table 4.1.

Summary. To summarize the feedback from Table 4.1: The students liked the proof assistant and would also used it in the future, even for homeworks if it would be possible to submit them in that format. They were missing some features such as De Morgan rules in first-order logic, tooltips and they would like to have some additional information about the proof assistant.

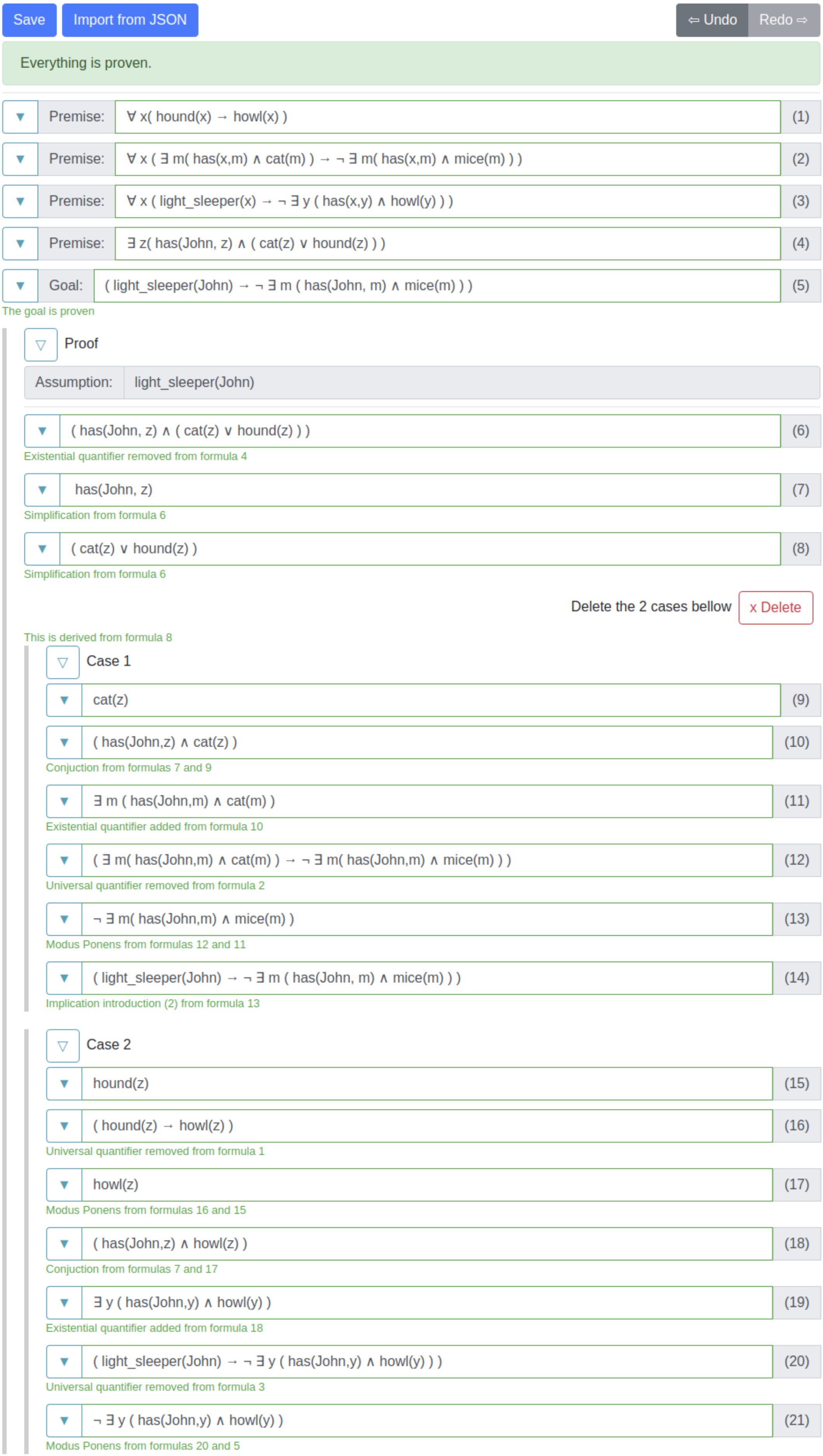
Additional feedback. We received much additional feedback as well. This feedback from the students is paraphrased and grouped in the following list:

Parsing the formulas:

- The tool is great because it also checks the syntax of the formulas.
- We did not know the format of formulas the tool can parse. An example or some hint under the proof assistant would be helpful.
- The parser's error messages are hard to read. It would be nicer to show what exactly is wrong instead of a four line long error message.

Proof tree modification:

- It is not possible to delete the whole proof. It would be helpful when we finish the exercise and move to another. Refreshing the window does the work; however, we lose the history of the application.
- At the beginning of the proof we usually write a few premises. It would be helpful if the default explanation type would be set to be a premise, until the explanation is changed for the first time. Then, it could be by default set to consequence.



Everything is proven.

Premise: ∀ x(hound(x) → howl(x)) (1)

Premise: ∀ x (∃ m(has(x,m) ∧ cat(m)) → ¬ ∃ m(has(x,m) ∧ mice(m))) (2)

Premise: ∀ x (light_sleeper(x) → ¬ ∃ y (has(x,y) ∧ howl(y))) (3)

Premise: ∃ z(has(John, z) ∧ (cat(z) ∨ hound(z))) (4)

Goal: (light_sleeper(John) → ¬ ∃ m (has(John, m) ∧ mice(m))) (5)

The goal is proven

Proof

Assumption: light_sleeper(John)

(has(John, z) ∧ (cat(z) ∨ hound(z))) (6)

Existential quantifier removed from formula 4

has(John, z) (7)

Simplification from formula 6

(cat(z) ∨ hound(z)) (8)

Simplification from formula 6

Delete the 2 cases bellow

This is derived from formula 8

Case 1

cat(z) (9)

(has(John,z) ∧ cat(z)) (10)

Conjuction from formulas 7 and 9

∃ m (has(John,m) ∧ cat(m)) (11)

Existential quantifier added from formula 10

(∃ m(has(John,m) ∧ cat(m)) → ¬ ∃ m(has(John,m) ∧ mice(m))) (12)

Universal quantifier removed from formula 2

¬ ∃ m(has(John,m) ∧ mice(m)) (13)

Modus Ponens from formulas 12 and 11

(light_sleeper(John) → ¬ ∃ m (has(John, m) ∧ mice(m))) (14)

Implication introduction (2) from formula 13

Case 2

hound(z) (15)

(hound(z) → howl(z)) (16)

Universal quantifier removed from formula 1

howl(z) (17)

Modus Ponens from formulas 16 and 15

(has(John,z) ∧ howl(z)) (18)

Conjuction from formulas 7 and 17

∃ y (has(John,y) ∧ howl(y)) (19)

Existential quantifier added from formula 18

(light_sleeper(John) → ¬ ∃ y (has(John,y) ∧ howl(y))) (20)

Universal quantifier removed from formula 3

¬ ∃ y (has(John,y) ∧ howl(y)) (21)

Modus Ponens from formulas 20 and 5

Figure 4.1: Screenshot of the proof assistant

What would you use the tool for?	Number of votes
To check the correctness of my proof.	1
To prove my homework assignment.	2
Both, to prove and to check the correctness.	6
I will not use it in the future.	1

How did the tool help you?	Number of votes
I checked the correctness of my proof.	4
I reminded myself of some rules of inference.	2
I saw other types of proof than tableaux.	3

Was the user interface intuitive?	Number of votes
Yes, but it needs some improvement.	6
Yes, but examples or a tutorial would be appreciated.	4

Would you recommend the tool to someone?	Number of votes
Yes, to check the correctness of my proof.	4
Yes, if we could submit the homework in it.	3
Yes, it has nice design and it is practical.	2

What new functionality would you like to see?	Number of votes
De Morgan laws in first order logic.	4
List of rules of inference known to the assistant.	3
Add a proof step before the first one.	2
Show the reason why was the branch closed.	3
Rename the generated new free variable	3
Tool-tips, so the assistant is easier to use.	2

Table 4.1: Summarized feedback from students

The feedback given by the assistant:

- The tool is great. We like that it shows from which rule of inference and from which steps does the consequence came from.
- The success message of the cases node is above both cases. It would be to more intuitive to have it under both cases nodes, even if it would mean the error message is repetitive.
- All the explanation types except for consequence, have the type, written in front of the formula. At first, it was not clear to us what type of explanation

does the formula have.

- The success messages for sub-proofs are misleading. They show that the goal was proven even if the steps we used to prove it are wrong.

Other feedback:

- When the browser is accidentally closed, the proof is lost. It could save the progress to local storage or show a popup asking whether we want to close it.
- Hiding the sub-proof is an excellent feature. It allows us to concentrate only on the part of the proof we are currently working on.

Conclusion. The feedback we received was informative with a lot of good suggestions.

We have decided to implement some of the functionality the students wanted to have. De Morgan laws in first-order logic and renaming the generated new free variable was implemented right after the testing. Other requests have either low priority or are very time-consuming to implement and therefore they are left as future work.

To summarize the testing, the testers considered our proof assistant helpful and it will probably be able to help the students to learn proper mathematics proofs; however, there is still space for improvement.

Conclusion

We have designed, implemented and tested a prototype of a proof assistant working with first-order logic (the proof assistant is accessible at `https://fmfi-uk-1-ain-412.github.io/proof-assistant/`), and described the process and our results in this thesis. The proof assistant is based on a hybrid formal proof system mixing Hilbert calculus and Sequent calculus. The proof assistant allows three types of proofs: direct proof, proof by cases, and proof by contradiction. The user creates a proof tree consisting of formalized first-order formulas, each with an explanation. Some explanations are automatically verified by the assistant using built-in inference rules and some require sub-proofs. The proven formulas can be later used in the proof.

The proof assistant has been implemented in the functional programming language Elm. Elm seems to be the right choice for this project. Thanks to its powerful and type-safe compiler we saved much valuable time. Since we expect third-party contributions to the project in the future, we wrote unit-tests for the critical parts of the application and published the code under MIT license.

We tested the proof assistant with ten students during evaluation. We received a lot of positive and informative feedback.

By building a working prototype of the proof assistant, we have achieved the goal of this thesis. The proof assistant is already helpful; however, there is still space for improvement. The primary improvement is to implement the first-order logic with equality. The secondary improvement is to make the rules of inference configurable. Currently, the editor has a fixed set of rules. If it would be configurable, the teacher could specify which rules can be used in a particular exercise. Another useful improvement would be telling the user, which branches were closed due to contradiction and which branches were proven to be correct. These features are however left for future work.

Appendix A: Source code

The source code of the proof assistant can be found on the attached CD and is also available online at `https://github.com/FMFI-UK-1-AIN-412/proof-assistant`.

The proof assistant itself can be found online at `https://fmfi-uk-1-ain-412.github.io/proof-assistant/`.

The source code on the attached CD has the following structure:

/bin/ contains the compiled code. To open the proof assistant open `index.html` in the browser.

/src/ contains the source code of the proof assistant, licence in `LICENSE` and also some information how to set up the development environment in `README.md`.

Bibliography

[1] Milan Cifra. *Prieskumník sémantiky logiky prvého rádu*. Univerzita Komenského v Bratislave, 2018. Submitted.

[2] Evan Czaplicki. An introduction to elm. [online; accessed May 16, 2018]. `https://legacy.gitbook.com/book/evancz/an-introduction-to-elm/details`.

[3] Herbert Enderton. *A mathematical introduction to logic*. Academic Press, San Diego, California, 2001.

[4] GitHub, Inc. Github pages. `https://pages.github.com/`.

[5] Gordon S. Novak Jr. Resolution example and exercises. [online; accessed May 16, 2018]. `https://www.cs.utexas.edu/users/novak/reso.html`.

[6] Kenneth Kunen, Jon Barwise, and Howard Jerome Keisler. *Handbook of mathematical logic*. North-Holland Publishing Co., 1983.

[7] Ján Kľuka. [personal communication, 2018].

[8] Ján Kľuka, Jozef Šiška, Alexandra Nyitraiová, and Zoltán Onódy. Package for parsing first-order logic formulas. [online; accessed May 20, 2018]. `http://package.elm-lang.org/packages/FMFI-UK-1-AIN-412/elm-formula`.

[9] Ján Kľuka, Jozef Šiška, Alexandra Nyitraiová, and Zoltán Onódy. Parser for first-order logic formulas. [online; accessed May 20, 2018]. `https://github.com/FMFI-UK-1-AIN-412/elm-formula`.

[10] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 2011.

[11] Alexandra Nyitraiová. *Educational tools for first-order logic*. Comenius University, Bratislava, 2018. Submitted.

[12] Raymond R. Smullyan. *First-Order Logic*. Springer, 1968.

[13] Stanford University. Fitch system. [online; accessed January 29, 2018]. `http://logic.stanford.edu/intrologic/glossary/fitch_system.html`.

[14] University of Cambridge, Technical University of Munich. Isar - intelligible semi-automated reasoning. [online; accessed January 29, 2018]. `https://isabelle.in.tum.de/Isar/`.

[15] Jozef Šiška. Propositional tableau editor. [online; accessed January 29, 2018]. `https://github.com/FMFI-UK-1-AIN-412/tableauEditor`.

[16] Vítězslav Švejdar. *Logika : neúplnost, složitost a nutnost*. Academia, Praha, 2002.