Praωf: Interactive Proof Assistant for Predicate Logic

MSc Thesis

Olga Petrovska
(843356)

Submitted to Swansea University in partial fulfilment
of the requirements for the Degree of Master of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

September 2016

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date .....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

Date .....

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date .....

Abstract

This thesis presents *Prawf* [prauv], an educational proof assistant for learning about natural deduction in predicate logic. It is an extended version of *Proof Editor*, a proof assistant for propositional logic, which is used as a part of the Logic in Computer Science course at Swansea University. We are extending its language with terms and adding natural deduction inference rules for quantified formulas. This requires implementation of substitution and alpha-equality check. The software is implemented in Haskell and uses LaTeX to present proofs in a user-friendly and aesthetic way. The improvements have been confirmed by a small-scale user study.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Dr Ulrich Berger, for his continuous support throughout the project, his valuable advice, patience and constant encouragement, which goes beyond this project.

Besides my supervisor, I would also like to thank my husband, Ferdinand Veselý, for love, belief in me, hours of explaining every little thing about logic and genuine excitement when using *Praωf*.

I thank Monika Seisenberger, Liam O'Reilly, HanYu Tang, Eoin Grua, Jean-Jose Razafindrakoto for participating in the survey and providing their valuable feedback.

Table of Contents

Chapter 1

Introduction

The idea of using computers to perform mathematical proofs in first-order logic is not new. It was in 1957 when Dag Prawitz, a first year student at the University of Stockholm, got inspired by the possibilities that technology could bring. This interest later made him one of the most important figures in the field of automated deduction [6].

There are various interactive theorem provers (further referred to as ITPs or proof assistants), and automatic theorem provers (ATPs) available today. The main purpose of both types of tools is getting the final result, i.e., a formal proof. In most of these applications the details about the way a proof has been derived are not always clearly presented. In a university environment, however, there is need for a tool that could help students understand every step of the process, i.e. an application that primarily focuses on communicating the process and not just deriving the final result.

There have been quite a few attempts to create such educational tools. Chapter 2 provides an overview of existing software focusing on the educational aspect and also shows how proofs are made using common tools such as *Coq* [32] and *Isabelle* [16].

In the given project we concentrate on proof assistants rather than ATPs. The first reason is that even though ATPs can work on complex proofs, ITPs are more expressive. Another reason is that proof assistants facilitate the learning process by letting the user develop a proof interactively in steps while ensuring the soundness of each step. It is important, however, that while students are interacting with the system, the input language used is as clear and straightforward as possible. The system should display proofs efficiently; all instructions, hints and error messages need to be clear and easily understandable. Already available applications do not always meet all these criteria.

An educational ITP, further referred to as *Proof Editor*, has previously been developed by Ulrich Berger in order to teach how natural deduction inference rules can be applied within the scope of propositional logic. This tool is being successfully used in labs for the *Logic for Computer Science* course (further referred as "Logic course") at Swansea University. However, since it only covers propositional logic, there has been a need to extend its functionalities. It is noteworthy that similar tools that cover predicate logic have been developed before, for example Roy Dyckhoff's et al. *MacLogic* [12] is currently used as a learning tool within the *Reasoning* module at the University of St Andrew [23]. Yet such tools have certain drawbacks, which will be discussed in detail in Chapter 2.

The aims of this project were the following:

- to extend the current *Proof Editor* to support natural deduction in first-order *predicate logic* by adding new functionalities, further definitions of formulas, proofs, rules, etc.,
- to improve readability of the source code by providing relevant comments within the code, breaking down large functions into smaller sub-functions, etc.,
- to improve the software documentation by making it more detailed and providing additional information about functionalities related to predicate logic.

The rest of the thesis is organised as follows. The second chapter gives an overview of modern interactive proof systems, particularly focusing on the recording and the communicating stages of proofs. It also provides more detailed information about *Proof Editor*, the source code of which serves as a base for developing a new extended ITP *Prawf*. In the third chapter software requirements for the new prover are listed. This chapter also contains information regarding project planning and execution, software architecture, specifics of implementing support for predicate logic, testing and user study results. Additionally, it also includes the *Prawf* user manual. The fourth chapter gives an overview of the areas that can be improved in the future. The final chapter concludes this thesis by summarising the major achievements of the project.

Chapter 2

Background and Related Work

2.1 Formal Proof and Styles of Proof Calculi

In [14] the author distinguishes two notions of a proof. The first one is a proof as a social construct that is used to convince an expert that a theorem is true by utilising natural language and/or symbols and figures. The second notion is that of a *formal proof*, which is "a string of symbols which satisfy some precisely stated set of rules and which prove a theorem, which itself must also be expressed as a string of symbols". For the purposes of this project a proof is viewed as a formal proof.

From that perspective we may consider the following styles of proof calculi [26, 5, 28]:

- *Hilbert style*, a deductive system with a larger set of axiom schemas and a limited number of inference rules, for example it may be just the *Modus ponens* rule: $P \rightarrow Q,\ P \vdash Q$.
- *Gentzen style* which was developed as an alternative to the Hilbert-style deduction system. Gentzen's approach relies on a larger number of inference rules and less on axioms. Within Gentzen-style deduction system one can distinguish *Calculus of Natural Deduction (ND)* and *Sequent Calculus (SC)*.

ND works with syntactic objects called *sequents*. A sequent has the form $\Gamma \vdash Q$, where Γ is a set of *assumptions (antecendents)*, e.g. $P_1, ...P_k$, that logically imply the *consequent (succedent)* Q. In case of natural deduction rules of inference operate on the consequent side only. For example, the ND conjunction rules are:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge^{+} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge^{-}\mathrm{l} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge^{-}\mathrm{r}$$

SC was invented by Gentzen after he had realised that he was not able prove normalisation theorem for a classical system of ND. In SC rules of inference operate on both sides, assumptions and consequents. For example, the first of the conjunction rules, $\wedge^+$, in SC applied to each side has the following forms:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge^+ \text{applied to conclusion} \qquad \frac{\Gamma,\ A,\ B \vdash C}{\Gamma,\ A \wedge B \vdash C}\ \wedge^+ \text{applied to assumption}$$

Overall, Gentzen-style approach is more practical than Hilbert-style systems. For example, logical expressions can be stripped off universal and existential quantifiers and manipulated upon by simpler rules of propositional calculus. Once a desired form of such expressions is achieved, the quantifiers can be added back again. In practice, it means that proving process is much simpler and proof recording is less tedious.

Sequent calculus is good for general theoretical analysis, while Natural Deduction is preferred in practical theorem proving. It perfectly aligns with educational purposes being "close to the natural way of human reasoning, and thus easy to learn" and its relation to functional programming makes it "particularly well suited for program synthesis from proofs" [4].

Having opted for Gentzen-style and ND in particular for the purposes of the given project, we view a proof as a tree that is constructed using the rules of natural deduction. There rules are applied to logical statements, formulas.

In propositional logic a formula is a proposition, a statement that carries some truth value. *Propositional formulas* can be either *basic* or *composite*. "Every atomic proposition is a propositional formula" [4]. Apart from that, $\top$ and $\bot$ are also basic propositional formulas. Composite formulas contain some kind of a logical connective (e.g., $\neg$, $\wedge$, $\vee$ or $\rightarrow$). Thus, "- If A is a propositional formula, then $\neg A$ is a propositional formula ... - If A and B are propositional formulas, then $(A \wedge B)$, $(A \vee B)$, and $(A \rightarrow B)$ are propositional formulas" [4].

In the context of propositional logic the rules of natural deduction are shown in Table 2.1.

Predicate formulas are more complex. For example, an atomic predicate formula consists of a *predicate* and *term(s)*, e.g. $P(x)$. Thus, it not only carries some truth value but also asserts something about its constituent term. Composite predicate formulas can be formed using connectives, as in propositional logic, and also by adding universal and existential quantifiers ($\forall$, $\exists$). Predicate formulas and specific rules of natural deduction for predicate logic will be discussed in more detail in Chapter 3.

Assumption rule	$\dfrac{}{\Gamma, A \vdash A}\ \text{use}$	
	Introduction rules	Elimination rules
$\wedge$	$\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge^{+}$	$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge^{-}_{\text{l}} \qquad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge^{-}_{\text{r}}$
$\rightarrow$	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\ \rightarrow^{+}$	$\dfrac{\Gamma \vdash A \rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \rightarrow^{-}$
$\vee$	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee^{+}_{\text{l}} \qquad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee^{+}_{\text{r}}$	$\dfrac{\Gamma \vdash A \vee B \qquad \Gamma \vdash A \rightarrow C \qquad \Gamma \vdash B \rightarrow C}{\Gamma \vdash C}\ \vee^{-}$
$\bot$		$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \text{efq} \qquad \dfrac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\ \text{raa}$

Table 2.1: Natural Deduction Rules in Propositional Logic [4]

2.2 Modern Interactive Proof Systems

2.2.1 Roles and Stages of Proofs

There exist various systems for interactive theorem proving. However, not all of them are suitable for educational environment. There is a multitude of reasons for that. Some tools are working with higher levels of proofs, where basic rules of natural deduction are normally hidden. Others may have interfaces that create "tunnel vision" by displaying only a current goal and not the whole proof. Some tools may use syntax that is not very clear and straightforward and communicate proofs using notation that may be unfamiliar to the user.

Before looking at specific tools, it is important to specify what roles a proof has and what stages of a proof are. In his paper on proof assistants [14], Herman Geuvers defines 2 roles that a proof has, namely:

- to *convince* that the statement is correct, which we interpret as a final result that an ITP provides, and

- to *explain* why it is so, which is a sequence of steps that describe how the result has been achieved.

There are also 3 stages of a proof: a *finding* stage, a *recording* stage and a *presenting/communicating* stage. Table 2.2 below reflects Guevers' overview of how "proof assistants support the roles and stages of proofs" at the current stage of development [14].

Proofs		Proof Assistants
Roles	Check (convince)	++
	Explain	– –
Stages	Finding	–
	Recording	+
	Communicating	–

Table 2.2: Roles and Stages of Proofs by Geuvers

From a pedagogical point of view, the most important aspects of an ITP are the ability to use it for *explaining* a certain proof result and to *record* and *communicate* it efficiently. The finding stage of a proof depends on the possibilities that an ITP provides in terms of recording and communicating, i.e., any constraints in language or application of rules. Described below are two main approaches to constructing proofs. In practice, a combination of both is often used.

- *Forward reasoning*, where proofs are built by applying inference rules to available premises to until the goal (conclusion) is reached;

- *Backward reasoning*, a goal-directed reasoning, where inference rules are applied backwards starting from the goal.

Among the most popular tools used are *Coq* [32], *Isabelle/HOL* [16] and *Mizar* [22]. These tools are very powerful and have various functionalities, which enable users to construct complex proofs. It should be noted that these ITPs are not primarily educational tools. Nevertheless, they are very important in terms of giving a broader perspective on what functionalities a good proof assistant should have.

The following sections look at how recording and communication of proofs works in some of these ITPs.

2.2.2 Recording Proofs

Below is a formalised proof of $\neg(P \wedge \neg Q) \rightarrow (P \rightarrow Q)$ written in *Coq*. Here a low-level natural deduction proof style is used, a style that is close to the way how students are taught. Usually proofs like this one do not need to be written out in detail. The steps in the proof are applications of *tactics* that roughly correspond to inference rules of natural deduction. However, this correspondence might not be immediately obvious to a beginner due to the naming of tactics (e.g., `split` being used as conjunction introduction in this example). This is usually due to *Coq* tactic covering more than just a particular natural deduction step and the language not being intuitive for someone who is looking at the proof only from the natural deduction inference perspective.

```
Variables P Q : Prop.
Theorem conjimp: ¬ (P ∧ ¬ Q) → (P → Q).
Proof.
  intro npq.
  intro p.
  apply NNPP.
  intro nq.
  apply npq.
  split.
  assumption.
  assumption.
Qed.
```

For comparison, here is how this proof can be formalised in *Isabelle/HOL*:

```
theorem conjimp: ¬(P ∧ ¬Q) ⟶ (P ⟶ Q)
apply(unfold not-def)
apply(rule impI)
apply(rule impI)
apply(rule notnotD, unfold not-def)
apply(rule impI)
apply(erule impE)
apply(rule conjI)
apply assumption
apply assumption
apply assumption
done
```

In this case the connection between the tactics and the natural deduction rules is more straightforward. For example, "**apply**(*rule impI*)" is an application of implication introduc-

tion. However, the drawback of using such formalisation in *Isabelle/HOL* is that it is not visible what is physically happening with the actual formula.

To solve this issue one may opt for *Isar*, a structured proof language for *Isabelle*, developed by M. Wenzel, because "... none of the major semi-automated reasoning systems support an adequate primary notion of proof that is amenable to human understanding." [33].

The development of *Isar* was inspired by the *Mizar* language because of the benefits of its declarative style, which is easier to understand. An *Isar* version of the proof is presented below:

theorem *conjimp*: $\neg(P \wedge \neg Q) \longrightarrow (P \longrightarrow Q)$
proof (*rule impI*, *rule impI*)
  assume $\neg(P \wedge \neg Q)$ **and** P
  have $\neg\neg Q$ **unfolding** *not-def*
  proof (*rule impI*)
    assume $Q \longrightarrow$ *False*
    with ‹P› **have** $P \wedge (Q \longrightarrow$ *False*$)$
      by (*rule conjI*)
    with ‹$\neg(P \wedge \neg Q)$› **show** *False* **unfolding** *not-def*
      by (*rule impE*)
  qed
  thus Q
    by (*rule notnotD*)
qed

Although the style of *Isar* is more expressive and easier to grasp, the main drawback is that it is using a combination of forward and backward reasoning, which may be confusing for students. Backward reasoning is more commonly used when teaching the inference rules. Although at the start of the proof implication introduction rules are applied backwards, once assumptions are available we switch to the forward reasoning, shifting the focus from the formula that we want to prove to assumptions.

To conclude, an educational ITP calls for as simple input syntax as possible. Input languages that the mentioned tools are using can make beginners struggle even if advanced users may find them straightforward.

2.2.3 Communicating Proofs

An input language is only covering the recording stage of a proof. Communicating the structure of the proof in an simple way brings a benefit to the overall learning process and at the same time makes it easier for a marker to evaluate the result.

Prawitz acknowledges the importance of Jaśkowski and Gentzen's contributions in terms of inventing their versions of natural deduction systems [28]. Based on these systems, apart from simpler *Suppes'* and *Lemmon's approaches* [18] to recording proofs, there exist two main trends in representing proofs, namely as *Gentzen-style trees* of sequents like $\Gamma \vdash A$ or by means of various sequential notations deriving from Jaśkowski's style of expressing natural deduction. Among the most common sequential ones is *Fitch notation* [10].

Below is an example of how a proof of $\neg(P \wedge \neg Q) \rightarrow (P \rightarrow Q)$ can be communicated using Fitch notation. It roughly corresponds the formalisation in *Isar* that has been shown in the previous section. This example is taken from [31]:

$\neg(P \wedge \neg Q)$	(1) premiss
P	(2) supposition
$\neg Q$	(3) supposition
$(P \wedge \neg Q)$	(4) from 2, 3 by $\wedge$I
$\bot$	(5) from 1, 4 by Abs
$\neg\neg Q$	(6) from 3 to 5 by RAA
Q	(7) from 6, by DN
$(P \rightarrow Q)$	(8) from 2 to 7 by CP

Finally, Figure 2.1 shows how a proof of the same formula can be represented as a Gentzen-style tree using *PANDA* [25], an educational proof assistant for teaching natural deduction. In this representation only consequents, the right sides of sequents on which the rules are applied are displayed. Assumptions are shown on a separate area of the main application window. Numbers in the proof tree are references to specific assumption formulas.

The main difference is that in the case of Fitch-style representation the forward proof approach is used. The tree, on the other hand, represents a more natural backward reasoning approach. As noted by Gasquet et al. [13], until recently Fitch notation and Suppes' approach — from which Lemmon's approach derives — had prevailed, as they are easier to typeset. However, recently Gentzen's approach is becoming more common.

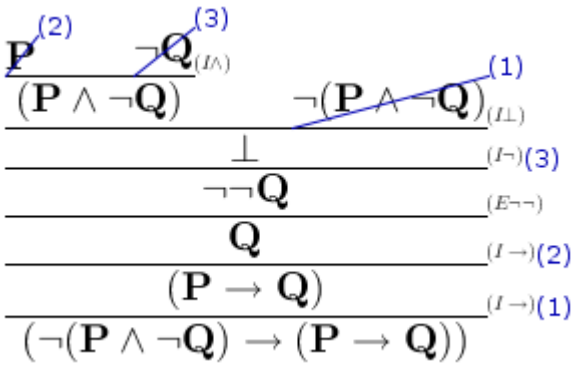

Figure 2.1: Example of a Proof Tree created in *PANDA*

2.2.4 Specific Educational Proof Assistants

In [9] the Committee on Logic Education of the Association of Symbolic Logic provides a list of logic software with a focus on the educational aspect. Although no longer actively maintained, this list is a valuable resource to understand what efforts have been undertaken in the area of proof assistants, in particular those supporting natural deduction for predicate logic.

The general overview of some selected educational tools is given in Table 2.3. The review of available logic software has shown that the platform on which it runs can make a big difference. Some tools such as *Bertie3* by Austen Clark [7] and *Program to learn Natural Deduction in Gentzen-Kleene's style (DN)* (Patrice Bailhache) [1] have been developed to run on older versions of Windows only and are not supported by newer versions. Other tools such as *Hyperproof* (Jon Barwise, John Etchemendy) [2] and *MacLogic* (Roy Dyckhoff *et al.*) [20] have been developed to run on Mac OS, which limits the number of potential users. Both of these tools seem to have been abandoned by their original creators. However, the latter has been "revived" by Branden Fitelson. He has developed emulators to run *MacLogic* on newer versions of Mac OS as well as Windows [11]. This has increased accessibility of the tool.

Due to the nature of single-platform-oriented tools, they are not very suitable for educational purposes. Even though students can access them in their university computer labs, they may not be able to do so at home if they use a different OS.

For that reason, quite a few tools have been developed on a "neutral" platform, i.e., web. This approach brings a big advantage because it increases accessibility of the tool. Students do not need to install software on their computers and this is especially relevant if the size of a class is large and it is physically impossible for the lecturer to help everyone with issues during the

Tool	Platform	Proof Display Style	Details
Bertie 3	DOS, Windows (older versions)	Unclear from the documentation[1]	It supports backward and forward reasoning; provides immediate feedback on correctness of proofs.
DN	DOS, Windows (older versions)	Schematic trees that represent steps of a proof	*DN* is a sequent-style natural deduction tool for predicate logic. The developer also created a sister tool called *FN* for learning how to calculate normal forms.
Hyper-proof	Mac (older versions)	Diagrams	It focuses of graphical representation of proofs as diagrams to make the learning process more visual and engaging [3].
MacLogic	Mac (older versions), emulator for newer Mac and Windows	System L (Lemmon)	In *MacLogic* "proofs may be constructed in the top-down mode (Construct mode), using Gentzen's sequent calculus LJ; these are then translated mechanically, as described by Prawitz, to natural deduction proofs, laid out in the linear style of Lemmon's text. (Proofs may also be entered line-by-line in this style)" [20].
Logic Daemon	web (Perl)	System L (Lemmon)	It supports natural deduction for predicate logic and is linked with various useful logic tools, e.g. Logic Primer, Quizmaster, Countermodel checker, Well-formed formula checker and Equivalency checker.
Pandora	web (Java)	Fitch	It supports backward and forward rules, provides examples of proofs in propositional and predicate logic and offers an extensive help manual.
ProofWeb	web (Javascript / OCaml)	Gentzen, Fitch	It is based on *Coq* but is much closer related to the natural deduction rules of predicate logic. It has more intuitive syntax and proof tactics than *Coq*.
PANDA	web (Java)	Gentzen	It supports backward and forward rules, provides examples of proofs in propositional and predicate logic, enables working on various proofs simultaneously, splitting of proof trees, etc.

Table 2.3: Overview of Educational ITPs for Teaching Natural Deduction Rules for Predicate Logic

installation.

Among the existing web tools are *Logic Daemon* [19], which uses Lemmon's approach to presenting natural deduction [18]. This tool is relatively straightforward but the way of writing down proofs is cumbersome and inefficient. Probably the most notable web-based tools are *Pandora* (**P**roof **A**ssistant for **N**atural **D**eduction using **O**rganised **R**ectangular **A**reas) [24], which uses Fitch notation, and *ProofWeb* [29], which supports both Gentzen and Fitch styles of representing natural deduction.

Having tried out various tools, *PANDA* (**P**roof **A**ssistant for **N**atural **D**eduction for **A**ll) seems to be the most user friendly and accessible. The developers of this software have taken into account their personal experience teaching logic to undergraduate students and created a user-friendly Java-based tool with multiple functionalities. These include backward and forward reasoning approaches for constructing proof trees, ability to split trees and work on sub-proofs separately, automatic filtering for applicable rules, formula input assistant, voice reader and other.

Nevertheless, from our viewpoint *PANDA* has certain drawbacks. From the usability perspective formula input assistant is a good idea but the syntax that the software uses is rather cumbersome with superfluous parentheses. Warning messages are displayed in French language only and often are longer than the size of the assistant window. Depending on what formula you work with, some rule buttons become too large and it is not possible to see the rule name without moving the scroll bar. Filtering of rules is a useful functionality but it would be helpful to have an option to switch it off. As students are learning, it is beneficial for them to learn distinguishing which rules are applicable and which are not without getting hints from the system.

Another issue is that some applicable rules are not displayed. For example, a user may want to apply backward reasoning approach to prove $\forall x P(x) \to P(c)$ and construct a proof tree similar to the one shown below.

$$\dfrac{\dfrac{u : \forall x P(x)}{P(\mathrm{c})}\,\forall^{-}}{\forall x P(x) \to P(\mathrm{c})}\,{\to}^{+}\ u : \forall x P(x)$$

In *PANDA* they will not be able to apply $\forall$-Elimination rule backwards to $P(c)$ as it is not displayed in the list of applicable rules once the user selects $P(c)$.

[1] Information regarding *Bertie 3* is based on software documentation as it is not possible to run this software on the latest versions of Windows.

Overall, *PANDA* is a helpful learning tool. Its graphical interface enables users to select rules and apply them in both ways, manipulate with proof trees breaking them down into parts. These functionalities are giving *PANDA* an advantage. Yet, at the same time, the multitude of the ways in which a user can interact with the system is disadvantageous as it may distract them from the actual learning goals.

This overview of the existing software enables us to define the most important factors in developing an educational proof assistant.

- Choice of a *development platform* (Windows, Linux, Mac, mobile, web) dictates how accessible the tool will be.
- *Efficient user interface* should be intuitive and free from distractions.
- *Simplicity of input* and clear way of *communicating proofs* that is aligned with the module materials.
- *Maintainability* and *extendibility* of software. Ability to access source code and adapt it to the needs of the course is highly desirable.

2.3 Berger's *Proof Editor* for Propositional Logic

Not being able to find a proof assistant that would fulfil all the teaching goals of the Logic course, Ulrich Berger developed *Proof Editor*, his own educational ITP that is currently being used as a part of the module.

The main advantage of this software is that it can be maintained internally and easily customised to fully match the specifics of the module.

Proof Editor is simple and elegant both from the point of view of recording and communicating proofs.

As mentioned before, the pedagogical aspect of an ITP calls for simplicity of the input language used. Otherwise, the language may distract the students from a proof itself. Berger's *Proof Editor* uses a concise and straightforward input language. Formulas are constructed using any capital letter except F. $\bot$ can be represented by any of the following strings:

```
"bot", "\bot", "Bot", "F", "_|_"
```

Composite formulas are created using connectives like $\neg$ (input options: `not, \neg, -`), $\wedge$ (input options: `&, and, \land`), $\vee$ (input options: `or, |, \lor`) or $\rightarrow$ (input options: `->, \to`).

Proof commands used by *Proof Editor* directly correspond to the rules of natural deduction as taught in the module (see Table 2.4).

Proof commands (in backwards reasoning mode)		
`use u`	use assumption `u`	
`andi`	And introduction	$\wedge^+$
`andel`	And elimination left	$\wedge_l^-$
`ander`	And elimination right	$\wedge_r^-$
`impi u`	Implication introduction	$\rightarrow^+ u$
`impe`	Implication elimination	$\rightarrow^-$
`impe A`	Implication elimination with premise	$\rightarrow^-$
`oril`	Or introduction left	$\vee_l^+$
`orir`	Or introduction right	$\vee_r^+$
`ore`	Or elimination	$\vee^-$
`efq`	ex-falso-quodlibet	*efq*
`raa`	reductio-ad-absurdum	*raa*
Control commands		
`undo`	undo a proof step	
`quit`	leave the prover	
`new`	start a new proof (without saving your current proof)	
`submit` *i*	submit current proof as solution to question *i*	
`delete` *i*	delete question *i*	
`?`	more explanations on the commands above	

Table 2.4: Proof and control commands of the *Proof Editor* [5]

Below is a log taken from the *Proof Editor*. These are the steps to prove the same formula that we have previously proven in *Coq* and *Isabelle*, $\neg(P \wedge \neg Q) \rightarrow (P \rightarrow Q)$.

```
Enter goal formula X > not (P and not Q) -> (P -> Q)
Enter command> impi u1
Enter command> impi u2
Enter command> raa
Enter command> impi u3
Enter command> impe (P and not Q)
Enter command> use u1
Enter command> andi
```

```
Enter command> use u2
Enter command> use u3
Proof complete.
```

The style is similar to the one used in *Isabelle/HOL* earlier. The use of the *Proof Editor*, however, has the following advantages:

- at each of the input steps students can see the proof visualised in LaTeXas a Gentzen-style tree. A complete proof tree for $\neg(P \wedge \neg Q) \rightarrow (P \rightarrow Q)$ generated in *Proof Editor* in shown in Figure 2.2),

- unlike in case with more advanced ITPs, e.g., *Coq*, *Isabelle* and *HOL*, which support higher order logics, *Proof Editor* focuses on non-automated lower level operations. This and the overall simplicity of the tool means that students are not distracted or confused by any extra functionalities.

- unlike some of the educational ITPs, e.g. *PANDA*, it uses simple text input, which obliges students to write out formulas and commands and therefore reinforces the learning process.

$$\dfrac{\dfrac{u1 : (P \wedge (Q \rightarrow \bot)) \rightarrow \bot \qquad \dfrac{u2 : P \qquad u3 : Q \rightarrow \bot}{P \wedge (Q \rightarrow \bot)} \wedge^{+}}{\dfrac{\bot}{(Q \rightarrow \bot) \rightarrow \bot} \rightarrow^{+} u3 : Q \rightarrow \bot} \rightarrow^{-}}{\dfrac{\dfrac{Q}{P \rightarrow Q} \rightarrow^{+} u2 : P}{((P \wedge (Q \rightarrow \bot)) \rightarrow \bot) \rightarrow (P \rightarrow Q)} \rightarrow^{+} u1 : (P \wedge (Q \rightarrow \bot)) \rightarrow \bot} \text{raa}$$

Figure 2.2: Example of a Proof Tree generated by *Proof Editor*

It is noteworthy that *Proof Editor* does not use $\neg$ when communicating proofs. Instead it uses implication of falsity. Thus, even if a user inputs `\neg Q` it will be shown as $Q \rightarrow \bot$ and not $\neg Q$. This simplifies application of the rules of inference and reminds students that $\neg Q$ is implication in its essence. By using the implication introduction rule they are able to get a new assumption Q. When $\neg Q$ is used all of this may not be very obvious as in case of *PANDA* proof visualisation (see Figure 2.1 on page 10).

From a technical perspective, the *Proof Editor* uses Sam Buss' bussproofs.sty package and draws proof trees in LaTeX[17].

The interrelation between input and output in *Proof Editor* is best described by looking at how a proof tree is visualised after each step that a user applies. The example below shows the step-wise derivation of the proof for $\neg(P \wedge \neg Q) \rightarrow (P \rightarrow Q)$.

```
Enter goal formula X > not (P and not Q) -> (P -> Q)
```

$?\mathbf{0} : ((P \wedge (Q \rightarrow \bot)) \rightarrow \bot) \rightarrow (P \rightarrow Q)$

Current goal: $?\mathbf{0} : ((P \wedge (Q \rightarrow \bot)) \rightarrow \bot) \rightarrow (P \rightarrow Q)$

```
Enter command> impi u1
```

$$\frac{?\mathbf{1} : P \rightarrow Q}{((P \wedge (Q \rightarrow \bot)) \rightarrow \bot) \rightarrow (P \rightarrow Q)} \rightarrow^{+} u1 : (P \wedge (Q \rightarrow \bot)) \rightarrow \bot$$

Current goal: $?\mathbf{1} : P \rightarrow Q$

Assumptions available: $u1 : \quad (P \wedge (Q \rightarrow \bot)) \rightarrow \bot$

```
Enter command> impi u2
```

$$\frac{\dfrac{?\mathbf{2} : Q}{P \rightarrow Q} \rightarrow^{+} u2 : P}{((P \wedge (Q \rightarrow \bot)) \rightarrow \bot) \rightarrow (P \rightarrow Q)} \rightarrow^{+} u1 : (P \wedge (Q \rightarrow \bot)) \rightarrow \bot$$

Current goal: $?\mathbf{2} : Q$

Assumptions available: $u2 : \quad P$
$u1 : \quad (P \wedge (Q \rightarrow \bot)) \rightarrow \bot$

```
Enter command> raa
```

$$\frac{\dfrac{\dfrac{?\mathbf{3} : (Q \rightarrow \bot) \rightarrow \bot}{Q}\,\text{raa}}{P \rightarrow Q} \rightarrow^{+} u2 : P}{((P \wedge (Q \rightarrow \bot)) \rightarrow \bot) \rightarrow (P \rightarrow Q)} \rightarrow^{+} u1 : (P \wedge (Q \rightarrow \bot)) \rightarrow \bot$$

Current goal: $?\mathbf{3} : (Q \rightarrow \bot) \rightarrow \bot$

Assumptions available: $u2 : \quad P$
$u1 : \quad (P \wedge (Q \rightarrow \bot)) \rightarrow \bot$

```
Enter command> impi u3
```

$$\dfrac{\dfrac{\dfrac{?\mathbf{4}:\bot}{(Q\to\bot)\to\bot}\to^{+}u3:Q\to\bot}{Q}\,\text{raa}}{\dfrac{}{P\to Q}\to^{+}u2:P}\to^{+}u1:(P\wedge(Q\to\bot))\to\bot}{((P\wedge(Q\to\bot))\to\bot)\to(P\to Q)}$$

Current goal: $?\mathbf{4}:\bot$

Assumptions available:

$u3:\ Q\to\bot$

$u2:\ P$

$u1:\ (P\wedge(Q\to\bot))\to\bot$

```
Enter command> impe (P and not Q)
```

$$\dfrac{?\mathbf{5}:(P\wedge(Q\to\bot))\to\bot\qquad ?6:P\wedge(Q\to\bot)}{\dfrac{\dfrac{\dfrac{\bot}{(Q\to\bot)\to\bot}\to^{+}u3:Q\to\bot}{Q}\,\text{raa}}{\dfrac{}{P\to Q}\to^{+}u2:P}\to^{+}u1:(P\wedge(Q\to\bot))\to\bot}{((P\wedge(Q\to\bot))\to\bot)\to(P\to Q)}}\to^{-}$$

Current goal: $?\mathbf{5}:(P\wedge(Q\to\bot))\to\bot$

Assumptions available:

$u3:\ Q\to\bot$

$u2:\ P$

$u1:\ (P\wedge(Q\to\bot))\to\bot$

```
Enter command> use u1
```

$$\dfrac{u1:(P\wedge(Q\to\bot))\to\bot\qquad ?\mathbf{6}:P\wedge(Q\to\bot)}{\dfrac{\dfrac{\dfrac{\bot}{(Q\to\bot)\to\bot}\to^{+}u3:Q\to\bot}{Q}\,\text{raa}}{\dfrac{}{P\to Q}\to^{+}u2:P}\to^{+}u1:(P\wedge(Q\to\bot))\to\bot}{((P\wedge(Q\to\bot))\to\bot)\to(P\to Q)}}\to^{-}$$

Current goal: $?\mathbf{6}:P\wedge(Q\to\bot)$

Assumptions available:

$u3:\ Q\to\bot$

$u2:\ P$

$u1:\ (P\wedge(Q\to\bot))\to\bot$

```
Enter command> andi
```

$$\dfrac{\dfrac{u1 : (P \wedge (Q \to \bot)) \to \bot \qquad \dfrac{?\mathbf{7} : P \qquad ?8 : Q \to \bot}{P \wedge (Q \to \bot)} \wedge^{+}}{\dfrac{\dfrac{\bot}{(Q \to \bot) \to \bot} \to^{+} u3 : Q \to \bot}{\dfrac{Q}{P \to Q} \to^{+} u2 : P}\ \text{raa}} \to^{-}}{((P \wedge (Q \to \bot)) \to \bot) \to (P \to Q)} \to^{+} u1 : (P \wedge (Q \to \bot)) \to \bot$$

Current goal: $?\mathbf{7} : P$

Assumptions available:
$u3 : \quad Q \to \bot$
$u2 : \quad P$
$u1 : \quad (P \wedge (Q \to \bot)) \to \bot$

```
Enter command> use u2
```

$$\dfrac{\dfrac{u1 : (P \wedge (Q \to \bot)) \to \bot \qquad \dfrac{u2 : P \qquad ?\mathbf{8} : Q \to \bot}{P \wedge (Q \to \bot)} \wedge^{+}}{\dfrac{\dfrac{\bot}{(Q \to \bot) \to \bot} \to^{+} u3 : Q \to \bot}{\dfrac{Q}{P \to Q} \to^{+} u2 : P}\ \text{raa}} \to^{-}}{((P \wedge (Q \to \bot)) \to \bot) \to (P \to Q)} \to^{+} u1 : (P \wedge (Q \to \bot)) \to \bot$$

Current goal: $?\mathbf{8} : Q \to \bot$

Assumptions available:
$u3 : \quad Q \to \bot$
$u2 : \quad P$
$u1 : \quad (P \wedge (Q \to \bot)) \to \bot$

```
Enter command> use u3
```

$$\dfrac{\dfrac{u1 : (P \wedge (Q \to \bot)) \to \bot \qquad \dfrac{u2 : P \qquad u3 : Q \to \bot}{P \wedge (Q \to \bot)} \wedge^{+}}{\dfrac{\dfrac{\bot}{(Q \to \bot) \to \bot} \to^{+} u3 : Q \to \bot}{\dfrac{Q}{P \to Q} \to^{+} u2 : P}\ \text{raa}} \to^{-}}{((P \wedge (Q \to \bot)) \to \bot) \to (P \to Q)} \to^{+} u1 : (P \wedge (Q \to \bot)) \to \bot$$

Proof complete.

Overall, Berger's *Proof Editor* is a simple yet very useful interactive theorem prover as it addresses the learning needs of students and matches with the materials that they are given throughout the course. It is noteworthy that *Proof Editor* is written in *Haskell* [15], which provides a natural and concise way of expressing manipulations on formulas in propositional logic. *Haskell* code is not only easy to maintain and extend but it can also be used for educational purposes to give students deeper understanding of inference in propositional logic.

Chapter 3

Interactive Proof Assistant for Predicate Logic

3.1 Motivation and Software Requirements

The main motivation behind using a proof assistant as a part of the module is to give students hand-on experience of what is taught at the course. Although there are various ITPs available — including specifically educational tools — there has been no tool that would fulfil all the requirements of the *Logic in Computer Science* module as taught at Swansea University.

Firstly, definitions of rules and the way of communicating proofs should exactly match what is taught during the course. For example, in the booklet with exercises complied for the *Mathematical logics* module at the University of Trento [21] existential elimination rule has a different form (Trento Version) from that used in the Logic course at Swansea University (Swansea Version) [4], as shown in Table 3.1.

Trento Version	Swansea Version
$\dfrac{\Gamma \vdash \exists x A(x) \qquad \Gamma, A(x) \vdash B}{\Gamma \vdash B}\ \exists^{-}$	$\dfrac{\Gamma \vdash \exists x A(x) \qquad \Gamma \vdash \forall x (A(x) \to B)}{\Gamma \vdash B}\ \exists^{-}$

Table 3.1: Ways of expressing $\exists$-Elimination rule

These definitions are simply different interpretation of the same rule and are both correct. It is important that a chosen definition is used consistently in course materials and in the ITP

because proof strategy will depend on this definition. Otherwise, a student may get confused.

Secondly, syntax should be clear and intuitive; proofs should be represented in a simple way that is familiar to users, in this case Gentzen-style trees. With regard to syntax, parentheses are normally used to avoid ambiguity when parsing formulas. Namely, parentheses are required to distinguish between $A \vee (B \wedge C)$ and $(A \vee B) \wedge C$. However, when there are some optional parentheses — as in case of $((A(x)) \wedge (B(x)))$ — users should be able to omit them. In case a user prefers to add all of them, the program should be able to strip off the unnecessary ones and display the formula in its simplest form, i.e. $A(x) \wedge B(x)$.

Thirdly, the ITP should not contain any distracting elements or have limitations as, for example, previously mentioned missing $\forall$-Elimination rule button in *PANDA*. While modern programming languages offer various ways of creating interesting and elaborate user interfaces, we opt for a more basic interface as the main focus should be on the actual proof process.

Lastly, the tool should be easily maintained and adapted. One of the major problems of using externally developed tools is that often-times there may be no access to the source code and if any bugs are spotted it may take some time before it is fixed. Developing the tool internally means that it can be easily modified and customised should the need for it arise. Another benefit of developing it locally is that it is possible to chose a preferred platform. *Haskell* is the chosen platform for this project. It is compatible with Windows and Linux OS, which are two most common operating systems at Swansea University. *Haskell Platform* is also available for Mac OS X, so the tool can be potentially extended to support this OS in the future.

To sum up, the main system requirements for the proof assistant for predicate logic are:

- clear input syntax;
- use of simplified Gentzen-style trees to show proofs[1];
- clear user interface without distractions or limitations;
- correct output at all times (there should be no errors in proof calculations as this would contradict the educational purpose of the tool);
- easy maintainability and customisability.

Our new proof assistant for predicate logic, further referred to as *Prawf* (from Welsh *prawf* [prau̯v] for "proof, test"), meets all the above requirements.

[1] Assumption side of sequents should not be displayed to avoid unnecessarily large proof trees.

3.2 Project Planning and Execution

Haskell was chosen as the implementation language for this project for a number of reasons. *Firstly*, the original *Proof Editor* is written in this language and it was easier to adapt it rather than rewrite it using a different language. *Secondly*, syntax of *Haskell* is clear and concise. Thus, the tool is easy to maintain and modify in case it needs to be extended further. *Thirdly*, although the primary goal of this new ITP it to teach the rules of natural deduction, the source code can serve as a learning tool for deeper understanding of the mentioned rules as well as the way how predicate formulas are constructed.

The main goal was to create a robust easily portable software with low software requirements. *Praωf* can be used on any Windows or Linux machine that has *Haskell* compiler and can compile LaTeX files. It can potentially be extended to run on Mac OS, however, this was not a priority as this OS is less commonly used for teaching at the Department.

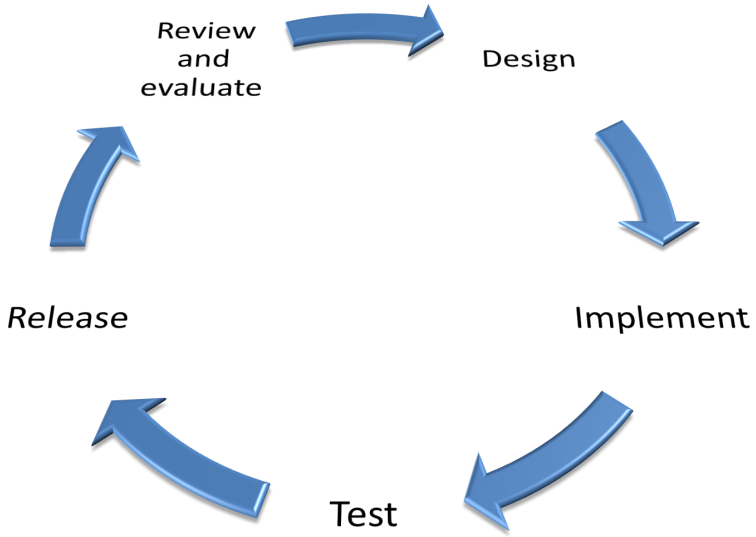

Figure 3.1: Development Cycle

The choice of programming language and the fact that *Prawf* is built upon *Proof Editor's* foundation influenced its structure and, consequently, the development approach on the whole. *Prawf* is modular in its design. Due to this kind of software architecture, an iterative development method with 3-week cycles was chosen. It gave an opportunity to perform constant testing of smaller components and apply necessary fixes where required. It also offered flexibility when defining the list of functionalities to include in the prover.

As Figure 3.1 shows each of the existing modules was first reviewed and evaluated, then additions and modifications to it were designed. This was followed by implementation and local testing on the module level. "Release" stands for a release of a complete working module.

Normally development was performed on one module at a time. However, as Figure 3.3 on page 25 shows, there are many dependencies between the modules and occasionally there was a need to go back to a "finished" module and add more updates.

The Gantt chart in Figure 3.2 describes the project schedule as initially planned. There were two milestones in the process, namely completion of the first prototype and completed implementation of the natural deduction rules within the scope of first-order logic.

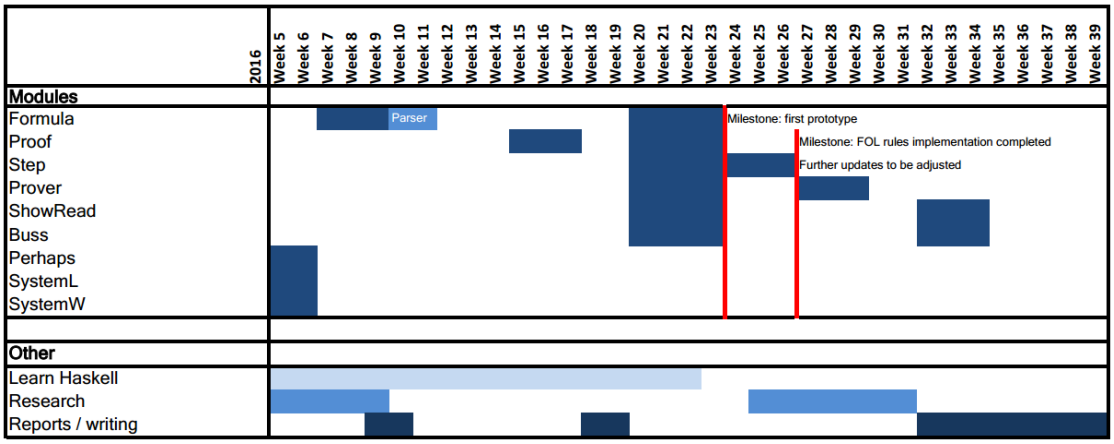

Figure 3.2: Project Gantt Chart

When undertaking the project two main possible risks were defined:

- loss of data
- inability to implement changes within the deadlines

With regard to the first risk, a Bitbucket repository was used to keep track of the latest updates to the code.

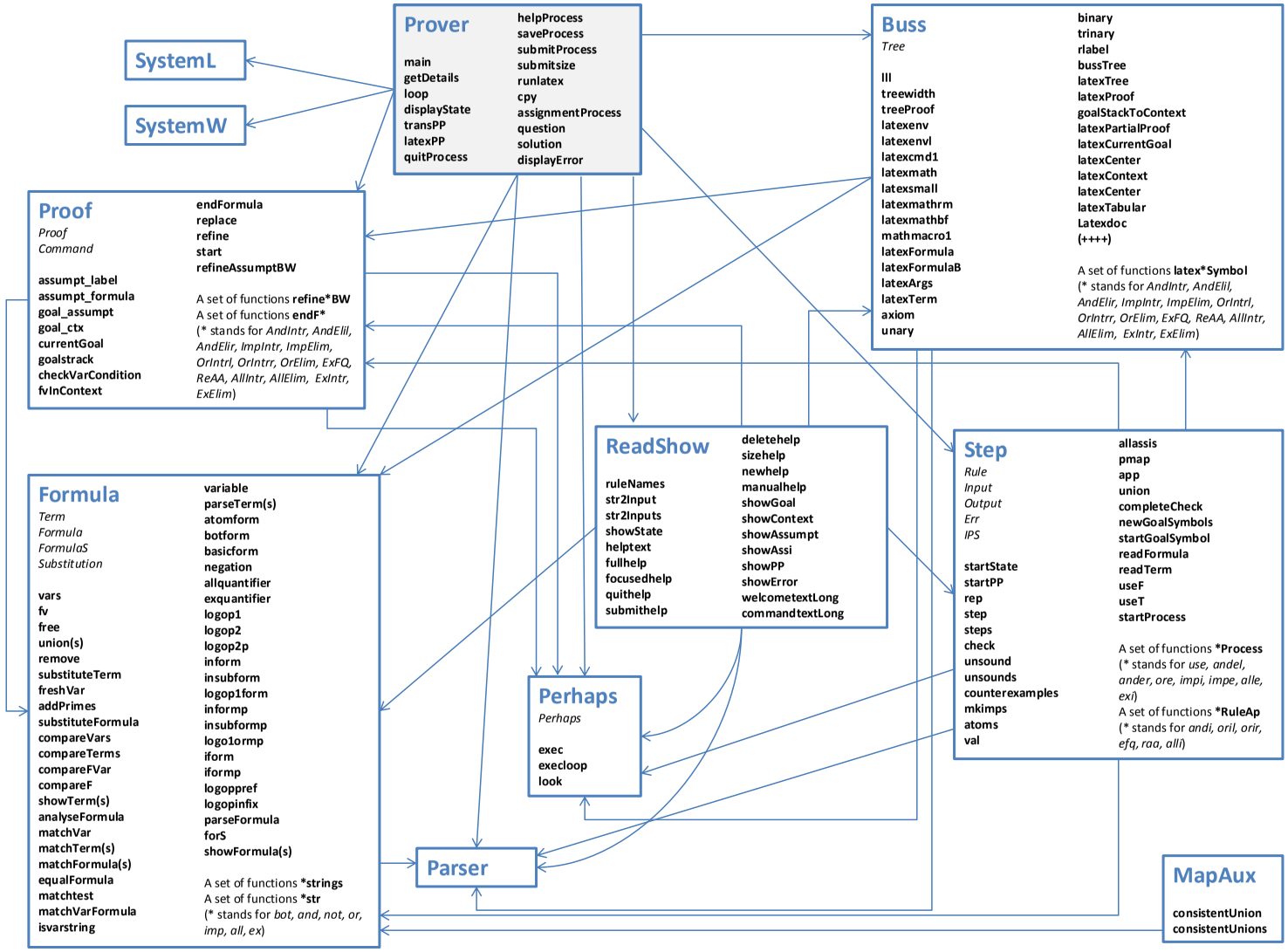

Figure 3.3: Modules Dependency Graph

The second risk was associated with the uncertainty as to the actual scope of the project. It was hard to estimate possible hidden complications that could arise from the fact that predicate logic is more complex than propositional logic. Apart from that, the module that manages I/O was one of the last ones to be updated. Thus, it was not possible to predict potential I/O issues until the later development stages.

The final software product has been released in week 34 as planned without encountering any significant issues. A working version of *Praωf* was available earlier but some additional changes have been implemented to improve usability of the tool, namely automatic term recognition, equivalence checks, variable/term substitution, and other.

3.3 Software Architecture

Software architecture of *Praωf* in based on the structure of *Proof Editor*. The tool consists of 11 modules, 10 of which correspond to the modules of *Proof Editor*. `MapAux`, a new additional module in *Praωf*, extends the existing Map library. Short description of each module is given in Table 3.2 on page 27.

A session starts by calling the `main` function of the `Prover` module. A loop within this function triggers calls to functions of other modules. Figure 3.3 on page 25 shows dependencies of the modules and gives an overview of data types and functions defined in these modules.

`Prover`, being the main module, manages user interaction with the system. However, `ReadShow` and `Buss` are the modules that visually represent this interaction. These modules receive calls from `Prover` and display appropriate information to the user. The `ReadShow` module writes out instructions/hints and displays error messages in GHCi, i.e. guides users in the process of proof construction. `Buss` module "draws" corresponding proof trees in LaTeX, visualising the results of the user's actions.

The actual proof process is done through interaction of three primary modules: `Formula`, `Proof` and `Step`. This interaction is also triggered by `Prover` making calls to functions in `Proof` and `Step`. The latter also interacts with `Proof` directly. `Proof` needs to work with the structure of formula, thus it utilises functions defined in `Formula`.

The remaining modules provide support functions to the primary modules. For example, `Formula` contains formula parsing functions that make calls to `Parser`, a library of monadic parser combinators. Functions defined within `MapAux` are used by `Formula` when alpha-equality checks are performed.

Module	Description
`Prover`	Main application module; controls the user's interaction with the system allowing the user to apply the rules of natural deduction in the backwards fashion
`Formula`	Defines data types for *Formula* and *Term*, performs operations with terms and variables in a formula and provides parsing for formulas using the parser combinators in the Parser module
`Proof`	Defines data type of *Proof*, refines and replaces proofs and generates an end formula based on the context and a proof, performs appropriate correctness checks
`MapAux`	Enhances existing Map library with utility functions
`Step`	Defines data types for *Rule* and *State* of a proof; executes user commands, etc.
`Parser`	A library of monadic parser combinators
`Buss`	Draws proof trees in LaTeX using the bussproofs.sty package
`Perhaps`	Defines *Perhaps* data type for communicating a result of a computation or an error message in case of failure
`ReadShow`	Interprets user input strings (parsing and "pretty-printing" for user interaction) and displays error messages and hints
`SystemL`	Operating system interface implementation for Linux
`SystemW`	Operating system interface implementation for Windows

Table 3.2: *Prawf* Modules Overview

3.4 Formulas: Old vs New Syntax

This and the following sections describe the specifics of extending definitions of propositional logic to predicate logic within the scope of an ITP development.

Prawf utilises existing definitions of propositional formulas used in *Proof Editor*. Syntax of predicate logic is, however, more complex. Therefore, there was a need to introduce a new data type, *Term*, and re-define the structure of an *atomic formula*. Atomic formula is a *basic formula* with no deeper structure. It does not contain any sub-formulas or logical connectives like $\neg, \wedge, \vee$ or $\rightarrow$.

Previously the syntax of an atomic formula was defined as `Atom String`. This is because *atomic propositions* were represented by simple strings such as P, Q, R.

In predicate logic there is a concept of a *predicate*, a property of an object. In order to express that an object, for example x, has property P we write $P(x)$. Such formula is an example of an

atomic formula in predicate logic. Here P stands for a predicate instead of a proposition, and x for an object. This object does not necessarily have to be a variable. Our current implementation also allows the use constants and functions as objects, referred to as *Terms* in the code.

```
data Term = Var String | Const String | Fun String [Term]
```

A property P may refer to more than one object. For example, consider the following statements:

- *Helen is a Welsh girl*
- *Maria is a Ukrainian girl*
- *Helen is Maria's colleague*
- *Maria is Helen's colleague*
- *Helen and Maria met in Swansea*

Let W represent a property of being Welsh, U represent a property of being Ukrainian and C mean that objects are colleagues. These statements can be expressed as $W(Helen)$, $U(Maria)$ and $C(Helen,\ Maria)$. Note that in the last case the property F refers to both objects, so we have a proposition with a list of two terms.

The last statement can be used as an example of a predicate with three interrelated terms: *M(Helen, Maria, Swansea)*, where M stands for "met in". Therefore, the new definition of an atomic formula is `Atom String [Term]`.

In order to build up *composite formulas* atomic sub-formulas are used. Thus, the definitions of conjunction, disjunction and implication remain unchanged.

```
| And Formula Formula
| Or Formula Formula
| Imp Formula Formula
```

For expressing negation all of these inputs are valid: `not`, `\neg` and `-`. In a proof tree $\neg P$ is always displayed as an implication: $P \to \bot$. There is a benefit in it from the educational perspective. Implication helps students to understand the essence of negation better and apply natural deduction rules more intuitively. In $P \to \bot$ there is already a "hint" that one may apply the implication introduction rule and get a new assumption P, while is case of $\neg P$ it is not that explicit.

Newly added *quantified formulas* are built upon the existing definitions of formulas. *Quantifiers* are not added as a separate data type as it would add additional complexity to the code and there is no benefit in having them as a separate concept. Universal and existential formulas are constructed in a similar way as other formulas, that is a prefix to define the type of formula and additional parameters, a string for the quantified variable and a formula.

```
| All String Formula
| Ex String Formula
```

For example, following the above definition, the quantified formula $\forall x\, P(x)$ is internally represented in *Praωf* as `All "x" (Atom "P" [Var "x"])`. The user's input used to construct this formula is simpler and is closer to the syntax that is used in the Logic course. The most common input would be `all x P(x)` but one could also use `All`, `For all`, `for all` or `\forall` to denote $\forall$.

With introduction of quantified formulas an issue of distinguishing free versus bound variables arose. A couple of new functions have been added to work with free variables, e.g., the function `fv` below returns a list of free variables in a formula. The function `free` checks if a variable is free or bound in the given formula.

Within `fv` a call to function `union` is made. In `union` elements of two lists are combined. The function `unions` is a generalisation of `union` to lists of lists. The function `remove` removes variable x from the list of variables.

```
fv :: Formula -> [String]
fv f = case f of
   Atom w ts -> unions (map vars ts)
   And f1 f2 -> fv f1 `union` fv f2
   Or f1 f2  -> fv f1 `union` fv f2
   Imp f1 f2 -> fv f1 `union` fv f2
   All x g   -> remove x (fv g)
   Ex x g    -> remove x (fv g)

free :: String -> Formula -> Bool
free x f = x `elem` (fv f)
```

Additionally, some more new functions have been added to handle substitution and perform alpha-equality checks on formulas. These will be discussed in more detail in the following section.

3.5 Rules: New Definitions and Hidden Pitfalls

As the aim of this project has been to extend the functionalities of the existing proof assistant to support natural deduction rules in predicate logic, the `Rule` data type defined in the module `Step` has been updated to include four new rules: ∀-Introduction, ∀-Elimination, ∃-Introduction and ∃-Elimination.

	Introduction rules	Elimination rules
∀	$\dfrac{\Gamma \vdash A(x)}{\Gamma \vdash \forall x A(x)}\ \forall^{+}$ (*)	$\dfrac{\Gamma \vdash \forall x A(x)}{\Gamma \vdash A(t)}\ \forall^{-}$
∃	$\dfrac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x A(x)}\ \exists^{+}$	$\dfrac{\Gamma \vdash \exists x A(x) \qquad \Gamma \vdash \forall x\,(A(x) \rightarrow B)}{\Gamma \vdash B}\ \exists^{-}$ (**)

(*) ∀-Introduction rule is subject to the condition that x should not occur free in Γ.

(**) ∃-Elimination rule is subject to the condition that x should not occur free in B.

Table 3.3: The Rules of Natural Deduction for Quantifiers (Sequent Notation) [4]

All rules are applied through a call to the function `step`. This function takes an input — either a rule or a text string — and a current proof state, and returns a new state.

```
step :: Input -> State -> State
```

Some rules such as conjunction introduction, right and left disjunction introduction, *efq (ex falso sequitur quodlibet)* and *raa (reductio ad absurdum)* can be applied straight away and do not require any additional input from the user. In this case, as is shown in the below snippet of code, a corresponding `*RuleAp` function is called and the rule is applied directly.

```
step inp state@(out,ips,pp,n) =
     check state'
```

```
 where
   state' =
     case inp of
       ...
       RuleI rule  ->
         ...
            case rule of
             {
              AssuR  -> (OkO,UseS,pp,n);
              AndiR  -> andiRuleAp state;
              AndelR -> (OkO,AndelS,pp,n);
              AnderR -> (OkO,AnderS,pp,n);
              OrilR  -> orilRuleAp state;
              OrirR  -> orirRuleAp state;
              OreR   -> (OkO,OreS,pp,n);
              ImpiR  -> (OkO,ImpiS,pp,n);
              ImpeR  -> (OkO,ImpeS,pp,n);
              EfqR   -> efqRuleAp state;
              RaaR   -> raaRuleAp state;
              AlliR  -> alliRuleAp state;
              AlleR  -> (OkO,AlleS,pp,n);
              ExiR   -> (OkO,ExiS,pp,n);
              ExeR   -> (OkO,ExeS,pp,n)
             }
```

Of all the newly added rules only ∀-Introduction requires no additional input. The function `alliRuleAp` takes the current state, calls the `refine` function on the partial proof and returns a new proof state.

```
alliRuleAp :: State -> State
alliRuleAp (out,ips,pp,n) =
  let { [gl] = newGoalSymbols n 1}
  in case refine pp (AllIntrBW gl) of
   {
    Success pp' -> (OkO,OkS,pp',n+1);
```

```
 Failure _   -> (Error0 RuleNotApplicableE,ips,pp,n)
}
```

∀-Introduction is subject to the condition that x should not occur free in any assumption valid at the point when the rule is applied. Therefore during the refinement, the software runs an internal check to make sure that this condition is met. In case it is not met, the bound variable in the formula to which the rule is applied is alpha-renamed. For example, if we apply the implication introduction rule to $B(x) \to \forall x\, A(x)$, x will occur free in the assumption $B(x)$. When the ∀-Introduction is then applied to $\forall x\, A(x)$, the proof assistant automatically substitutes the bound variable x by x', resulting in $\forall x'\, A(x')$.

Before the rule is applied	After the rule is applied
$\dfrac{?\mathbf{1} : \forall x A(x)}{B(x) \to \forall x A(x)} \to^{+} u : B(x)$	$\dfrac{\dfrac{?\mathbf{2} : A(x')}{\forall x' A(x')}\,\forall^{+}}{B(x) \to \forall x' A(x')} \to^{+} u : B(x)$

Table 3.4: Variable Substitution in ∀-Introduction

This helps users to distinguish free and bound variables and — from the teaching perspective — it is also a good way of demonstrating *alpha-equality*.

The notion of alpha-equality is especially important in case of ∀-Elimination, ∃-Introduction as applying these rules requires substitution.

Substitution is used to replace all occurrences of a free variable in a formula for a given term, for example $A(x)[x := t] = A(t)$.

The `substituteFormula` function is defined as shown below. It takes a string (the free variable name), a term (the term that will be used to substitute the variable), a formula that contains the free variable and then constructs a new formula in which the variable is substitued by the term.

```
substituteFormula :: String -> Term -> Formula -> Formula
```

When in the process of constructing a proof tree substitution has taken place, comparison of equality of formulas may be required. The alpha-equality check covers such cases.

As previously mentioned, some rules of natural deduction can be applied directly; the remaining ones require the user to provide some additional input. For example, conjunction elimination, disjunction elimination and implication elimination expect a formula as an input, while in case of implication introduction an assumption variable is expected. The software is designed in such a way that users can either type in the name of the rule and a required formula or a variable in one line, for example `impi u` (for implication introduction with assumption variable u), or they can type in the rule command and then will be asked to provide the missing information.

```
Enter goal formula X > A(x) -> B(x)
Enter command> impi
Enter assumption variable> u
```

This functionality is achieved by introduction of intermediate states which correspond to specific rules of natural deduction. Adding states for the rules that are applied directly is unnecessary.

```
data IPS = OkS | StartS | CompleteS
         | UseS | ImpiS | ImpeS | AndelS | AnderS | OreS
         | AlleS | ExiS | ExeS
  deriving (Show, Read)
```

When the user wants to apply one of such rules, for example disjunction elimination, the state changes to `OreS` for $\vee$-Elimination state. This way a corresponding process, `oreProcess`, is triggered.

The states referring to the rules of propositional calculus did not require altering. The three remaining rules of predicate calculus were built on the same principle. However, this original naïve implementation has proven to have certain drawbacks. For example, in case of $\forall$-Elimination.

During the initial implementation, two new states, `alleS` and `alleS1`, were introduced. When the `alleProcess` function was triggered, the state was changed from `alleS` to `alleS1`. Thus, in order to get the premise $\forall x A(x)$ in $\dfrac{?\mathbf{1} : \forall x A(x)}{A(\mathrm{t})}\,\forall^{-}$ the user had to go through the following steps:

```
Enter goal formula X > A(t)
Enter command> alle
```

```
Enter the quantified formula> All x A(x)
Enter the term that should be generalised> t
```

This solution was cumbersome and not user-friendly. Later an alternative solution has been found. The term is now automatically calculated within the `alleProcess` function. This makes the last input step redundant, improves usability and eliminates user frustration from the need to take what intuitively seems to be an unnecessary action.

Rules that are applied to existential formulas also require additional input, however, the processes triggered by the `ExiS` and `ExeS` states are less complex compared to the `alleProcess`. In case of ∃-Introduction, apart from the rule name `exi`, a user provides a term as additional input. The below snippet of code shows that during refinement an existential formula to which the ∃-Introduction rule is applied is de-constructed and the bound variable `v` is substituted by this term `t`, generating the new level formula `f1`. A new goal `g1` is created by updating formula in assumption. This goal together with the new proof `p1` is then used to create a partial proof.

```
refineExIntrBW :: PlProof -> GlSymbol -> Term -> Perhaps PlProof
refineExIntrBW ((a0,c0):gs,ctx,p) gl t =
  case (assumpt_formula a0) of
   {
    Ex v f ->
      let {
           f1 = substituteFormula v t f;
           g1 = ((gl,f1), c0);
           p1 = replace p (assumpt_label a0) (ExIntr (AnProof gl) v f t)
          }
      in return (g1:gs,ctx,p1);
    _ -> fail ("ExIntrBW, command not applicable to: "
               ++ showFormula (assumpt_formula a0))
   }
```

3.6 *Praωf* User Manual

This section contains a user manual for *Praωf*, which is vastly based on the user manual for *Proof Editor* [5]. This manual is also available on the project website [27].

Software Requirements *Prawf* has been developed in Haskell and runs within GHCi, the interactive environment of GHC (Glasgow Haskell Compiler). In Windows it is possible to run the software using WinGHCi or run GHCi in shell mode in Emacs. All prover files need to be in the directory `prover`. Additionally, LaTeX is required to display proofs.

Getting started To run the tool in shell mode, follow the below steps:

Move into the directory prover: `cd prover`
Open the Emacs editor: `emacs&` (or `xemacs&`)
Open a shell in Emacs: `M-x shell` (M is a Meta-key or the ESC-key)
Start interactive Haskell: `ghci`
Load the file Prover.hs: `:l Prover.hs`

When using WinGHCi, in order to load the prover click *File* $\rightarrow$ *Load* and go to the directory where the file Prover.hs is saved and load it.

Run the function main: `main`

In Linux this opens a DVI file displaying the current state of your proof and giving information about the current goal and the available commands.

In Windows you may need to open this file manually. The file name is `pproof.dvi` and it is normally located in the same directory as the prover. All instructions and hints are given at the command prompt and not in the DVI file.

Once a proof is completed the proof tree is written in the file pproof.tex, a LaTeX document.

Syntax of formulas The usual bracketing rules apply when typing in formulas. For example, `A -> (B -> C)` can be written `A -> B -> C`. *Prawf* strips unnecessary parentheses. For example, the input `((A) and (B)) -> ((B) and (A))` will be displayed as $(A \wedge B) \rightarrow (B \wedge A)$.

Atomic formula can be any letter except `F`. It also may contain terms. The input format can be `A` or `A(x)`, where `x` is a term or a list of terms. Terms can be either constants, variables or

functions (`f(x)`). Terms are separated by commas in the list: `A(x, f(x))`. There should be no space between the predicate and terms.

Composite formulas are built using logical connectives and can be written in various ways as shown in the table below. All input options are case sensitive. Negation can also be written as

Connectives & Quantifiers	Possible Input Options			Examples	
$\wedge$	`and`	`&`	`\land`	$A \wedge B$	`A and B`
$\vee$	`or`	`|`	`\lor`	$A \vee B$	`A or B`
$\bot$	`bot` or `Bot` or `F`	`_|_`	`\bot`	$\bot$	`bot`
$\neg$	`not`	`-`	`\neg`	$\neg A$	`not A`
$\forall$	`all` or `All`	`For all`	`\forall`	$\forall x\, A(x)$	`all x A(x)`
$\exists$	`ex` or `Ex`	`Exists`	`\exists`	$\exists x\, A(x)$	`ex x A(x)`
$\to$		`->`	`\to`	$A \to B$	`A -> B`

implication: `not A -> B` is the same as `(not A) -> B` and the same as `(A -> bot) -> B`.
Binding priorities (from strong to weak): `not, and, or, ->`.
Implication, conjunction and disjunction associate to the right.

Example:
`not A -> not B -> A or B and C -> bot`
is the same as
`(not A) -> ((not B) -> ((A or (B and C)) -> bot))`
and also the same as
`(A -> bot) -> (B -> bot) -> not (A or B and C)`

Supported Commands There are two types of commands in *Praωf*: general control commands and specific commands to apply the natural deduction rules.

Control commands	
`undo`	undo a proof step
`quit`	leave the prover
`new`	start a new proof (without saving your current proof)
`submit` i	submit curent proof as solution to question i
`delete` i	delete question i
`?`	more explanations on the commands above

Proof commands		
`use u`	Use an available assumption with label u	
`use`	Same as above. Since the label is missing, you are prompted to enter it.	
`andi`	And introduction rule applied backwards	$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge^{+}$
`andel B`	And elimination left backwards. If the goal was A, the new goal will be $A \wedge B$.	$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge^{-}_{\mathrm{l}}$
`andel`	As above, but since the formula B is missing you are prompted to enter it.	
`ander A`	And elimination left backwards. If the goal was B, the new goal will be $A \wedge B$.	$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge^{-}_{\mathrm{r}}$
`ander`	As above, but since the formula A is missing you are prompted to enter it.	
`impi u`	Implication introduction backwards. The current goal must be of the form $A \to B$. The new goal is B and A is added as an assumption with a label u.	$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to^{+}$
`impi`	As above, but since the assumption label is missing you are prompted to enter it.	
`impe A`	Implication elimination backwards. If the goal was B, there will be two new goals: $A \to B$ and A.	$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \to^{-}$
`impe`	As above, but since the formula A is missing you are prompted to enter it.	
`oril`	Or introduction left backwards. The current goal must be of the form $A \vee B$. The new goal is A.	$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee^{+}_{\mathrm{l}}$

`orir`	Or introduction right backwards. The current goal must be of the form $A \vee B$. The new goal is B.	$\dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee_{\mathrm{r}}^{+}$
`ore A or B`	Or elimination backwards. If the goal was C, there will be three new goals: $A \vee B$, $A \to C$, and $B \to C$.	$\dfrac{A \vee B \qquad A \to C \qquad B \to C}{C} \vee^{-}$
`ore`	As above, but since the formula $A \vee B$ is missing you are prompted to enter it.	
`efq`	Ex-falso-quodlibet backwards. The goal can be any formula. The new goal will be $\bot$.	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}$ efq
`raa`	Reductio-ad-absurdum backwards. The goal can be any formula A. The new goal will be the double negation of A.	$\dfrac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}$ raa
`alli`	All introduction rule backwards. The current goal must be of the form $\forall x\, A(x)$. The new goal will be $A(x)$. NOTE: x must not occur free in any assumption valid at the point.	$\dfrac{\Gamma \vdash A(x)}{\Gamma \vdash \forall x A(x)} \forall^{+}$
`alle all x A(x)`	All elimination rule backwards. The current goal must be a predicate formula of the form $A(t)$[2], where t is a term.	$\dfrac{\Gamma \vdash \forall x A(x)}{\Gamma \vdash A(t)} \forall^{-}$
`alle`	As above, but since the quantified formula is missing you are prompted to enter it.	
`exi t`	Exists introduction rule backwards. The current goal must be of the form $\exists x\, A(x)$.[2] You need to add a term t which will subsitute the variable x.	$\dfrac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x A(x)} \exists^{+}$

`exi`	As above, but since the term is missing you are prompted to enter it.	
`exe ex x A(x)`	Exists elimination rule backwards. The current goal can be any formula. NOTE: x must not be free in B.	$\dfrac{\exists x A(x) \qquad \forall x\,(A(x) \to B)}{B}\ \exists^-$
`exe`	As above, but since the quantified formula is missing you are prompted to enter it.	

Example Session in GHCi

```
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Prelude> :cd C:\Users\Lenovo\Desktop\Haskell\Prover_files\prover
Prelude> :load "Prover.hs"
[ 1 of 10] Compiling MapAux           ( MapAux.hs, interpreted )
[ 2 of 10] Compiling SystemW          ( SystemW.hs, interpreted )
[ 3 of 10] Compiling Perhaps          ( Perhaps.hs, interpreted )
[ 4 of 10] Compiling Parser           ( Parser.hs, interpreted )
[ 5 of 10] Compiling Formula          ( Formula.hs, interpreted )
[ 6 of 10] Compiling Proof            ( Proof.hs, interpreted )
[ 7 of 10] Compiling Buss             ( Buss.hs, interpreted )
[ 8 of 10] Compiling Step             ( Step.hs, interpreted )
[ 9 of 10] Compiling ReadShow         ( ReadShow.hs, interpreted )
[10 of 10] Compiling Prover           ( Prover.hs, interpreted )
Ok, modules loaded: Parser, Prover, Perhaps, Formula, Proof, Buss, Step,
ReadShow, SystemW, MapAux.
*Prover> main
Enter goal formula X > ex x (A(x)) -> all x (A(x) -> B(x)) -> ex x B(x)
Enter command> impi u1
Enter command> impi u2
Enter command> exe ex x A(x)
Enter command> use u1
```

[2] By $A(t)$ we mean any formula containing the term t, not just the application of the predicate A to the term t.

```
Enter command> alli
Enter command> impi u3
Enter command> exi x
Enter command> impe A(x)
Enter command> alle all x (A(x) -> B(x))
Enter command> use u2
Enter command> use u3
Proof complete.
Enter quit, submit <i>, delete <i>, new, or ?> quit
*Prover>
```

This session generates the following proof tree:

$$\dfrac{u1 : \exists x A(x) \qquad \dfrac{\dfrac{\dfrac{u2 : \forall x(A(x) \to B(x))}{A(x) \to B(x)}\forall^- \qquad u3 : A(x)}{B(x)}\to^-}{\exists x B(x)}\exists^+}{\dfrac{\dfrac{A(x) \to \exists x B(x)}{\forall x(A(x) \to \exists x B(x))}\to^+ u3 : A(x)}{\forall x(A(x) \to B(x)) \to \exists x B(x)}\forall^+}\exists^-$$
$$\dfrac{\forall x(A(x) \to B(x)) \to \exists x B(x)}{\exists x A(x) \to (\forall x(A(x) \to B(x)) \to \exists x B(x))}\to^+ u2 : \forall x(A(x) \to B(x))$$
$$\to^+ u1 : \exists x A(x)$$

3.7 Testing and Sample Proofs

When developing *Praωf* it was important to make sure that the software is fault-free in a way that all proof calculations are correct at all times. Moreover, any invalid input needed to be caught by the system. This was achieved through testing the output of particular functions on the level of each module during each iteration. At this testing stage, deliberately faulty input was used to make sure that the tool responds properly and catches errors and exceptions.

Once all modules were completed, overall testing was performed. This was done by proving validity of various predicate formulas, mostly taken from [21] but also variations of these formulas with wrong input. For example, one of the exercises included in [21] was to prove validity of $\forall x(A \to B) \to (A \to \forall x B)$ assuming that x does not occur free in A. For testing purposes this for-

mula was slightly modified: $\forall x(A \to B(x)) \to (A \to \forall x B(x))$. Figure 3.4 shows a corresponding proof tree generated in *Praωf*.

$$\dfrac{\dfrac{\dfrac{\dfrac{u1 : \forall x(A \to B(x))}{A \to B(x)}\forall^- \qquad u2 : A}{B(x)}\to^-}{\forall x B(x)}\forall^+}{A \to \forall x B(x)}\to^+ u2 : A \qquad \Big/ \forall x(A \to B(x)) \to (A \to \forall x B(x))\to^+ u1 : \forall x(A \to B(x))$$

Figure 3.4: Proof Tree for $\forall x(A \to B(x)) \to (A \to \forall x B(x))$

If the mentioned condition is ignored and x occurs free in A it is not possible to complete the proof. An attempt to prove such formula shown in Figure 3.5 also contains a test for the $\forall$-Introduction rule condition. This condition states that x should not occur free in any free assumption at the point when the rule is applied. When the initial formula is added it has the form: $\forall x(A \to B(x)) \to (A(x) \to \forall x B(x))$. When the $\forall$-Introduction rule is applied to $\forall x\ B(x)$ the condition is checked and since x occurs free in the second assumption ($u2 : A(x)$) the bound variable x in $\forall x\ B(x)$ is changed to x'. This confirms that the condition check is working as expected.

$$\dfrac{\dfrac{\dfrac{?\mathbf{4} : A \to B(x') \qquad ?5 : A}{B(x')}\to^-}{\forall x' B(x')}\forall^+}{A(x) \to \forall x' B(x')}\to^+ u2 : A(x) \qquad \Big/ \forall x(A \to B(x)) \to (A(x) \to \forall x' B(x'))\to^+ u1 : \forall x(A \to B(x))$$

Figure 3.5: Proof Attempt of $\forall x(A \to B(x)) \to (A(x) \to \forall x B(x))$

Another test example is a check of the $\exists$-Elimination rule condition (see Table 3.3 on page 30). This test can be run when proving $\forall x A(x) \to \neg\exists x(\neg A(x))$ or, if converted to *Praωf's* way of displaying it, $\forall x A(x) \to (\exists x(A(x) \to \bot) \to \bot)$. Figure 3.6 shows a proof tree for this formula generated by the prover. When applying the $\exists$-Elimination rule to $A(y) \to \bot$ users are asked to input the existential formula which will become the next level left branch of the tree. Technically, this is the only input that is required. The right branch is formed automatically through deconstruction of the provided existential formula. The components obtained in this way are then used in construction of a new universal formula, which would start the right branch.

$$\dfrac{u2 : \exists x(A(x) \to \bot) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{u3 : A(x') \to \bot \qquad \dfrac{u1 : \forall x A(x)}{A(x')}\,\forall^-}{\bot}\,\to^-}{A(x) \to \bot}\,\to^+ u4 : A(x)}{(A(x') \to \bot) \to (A(x) \to \bot)}\,\to^+ u3 : A(x') \to \bot}{\forall x'((A(x') \to \bot) \to (A(x) \to \bot))}\,\forall^+}{A(x) \to \bot}\,\exists^- \qquad \dfrac{u1 : \forall x A(x)}{A(x)}\,\forall^-}{\dfrac{\dfrac{\bot}{\exists x(A(x) \to \bot) \to \bot}\,\to^+ u2 : \exists x(A(x) \to \bot)}{\forall x A(x) \to (\exists x(A(x) \to \bot) \to \bot)}\,\to^+ u1 : \forall x A(x)}\,\to^-$$

Figure 3.6: Proof Tree for $\forall x A(x) \to \neg\exists x(\neg A(x))$

For testing purposes an incorrect input was provided deliberately. Since x occurs free in $A(x) \to \bot$, the existential formula added was $\exists x(A(x) \to \bot)$, i.e., the bound variable used in it was x, which is an invalid input. This resulted in an error message, which confirmed that the condition check is executed correctly:

```
Enter missing formula of the form Ex x A(x)> ex x (A(x) -> F)
Failed to refine: ExElimBW, provided variable: x must not be free
in the current goal.>
```

Although it is not possible to add an existential formula in this case as $\exists x(A(x) \to \bot)$, the proof tree still shows it. This is due successful outcome when testing correctness of the alpha-equality checker as well as of variable substitution. Formula $\exists x(A(x) \to \bot)$ matches the assumption $u2$. However, since x is free in $(A(x) \to \bot)$, it is necessary that a bound variable x is distinguished from the free variable x in the universal formula created when the rule is applied. Thus, if the input is $\exists x'(A(x') \to \bot)$, the partial proof has a form as shown in Figure 3.7. The alpha-equality checker is tested by applying the use rule to the current goal. Since $\exists x'(A(x') \to \bot)$ is alpha-equivalent to $\exists x(A(x) \to \bot)$, the variable names are changed to match those in the assumption. Thus, this test has also passed.

$$\dfrac{?\mathbf{5} : \exists x'(A(x') \to \bot) \qquad ?6 : \forall x'((A(x') \to \bot) \to (A(x) \to \bot))}{A(x) \to \bot}\,\exists^-$$

Figure 3.7: Application of $\exists$-Elimination Rule when Constructing the Tree in Figure 3.6.

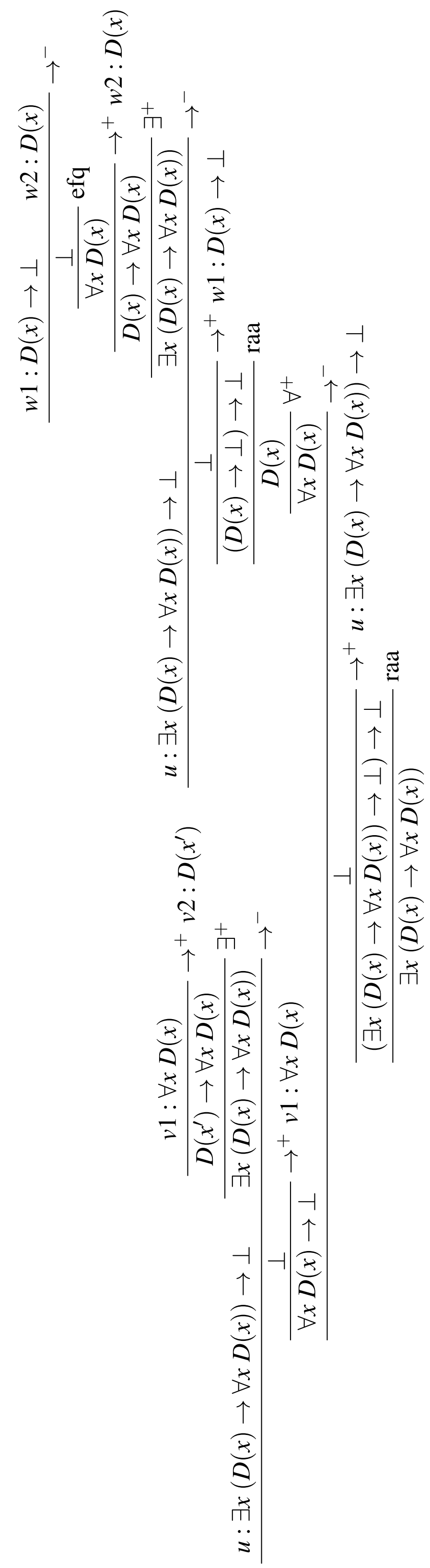

Figure 3.8: An Example of the Drinker Paradox Solution

Although *Praωf* is mainly aimed at proving simpler formulas, it has also been tested for a slightly larger proof of the *Drinker principle*, which states that there exists a person that when they are drinking then everyone else in the pub are drinking. This principle can be formulated as follows: $\exists x(D(x) \rightarrow \forall x D(x))$. Figure 3.8 on page 43 shows a formal proof of this principle created in *Praωf*.

Overall, testing of the software was successful and useful for detecting minor bugs. All of these bugs are now fixed.

3.8 User Study

A small user study involving six participants was conducted to evaluate user experience when using *Praωf*. Due to the nature of this software all participants were required to have at least basic knowledge of natural deduction. Three participants were advanced users with experience of working with more advanced proof assistants and theorem provers. The other three participants had no or little experience with such tools.

Each participant was given a form (see Appendix A), which contained brief information about the prover, a number of formulas to prove, a proof example, a list of applicable rules, a list of proof and control commands, and a short questionnaire.

A little proof session demonstration was given to each participant before they tried using the software themselves. The purpose of such demonstration was to give them a general idea about how the prover works and to show them where to find useful references. This is an approach that is currently used in lab sessions of the Logic course when *Proof Editor* is introduced to students.

Participants were asked to use *Praωf* and prove a couple of formulas — either from those provided in the handout or any other formulas of their choice — using *Praωf*. While constructing proofs, they could ask questions in case anything needed clarification.

According to the data collected through the questionnaire and discussions in person, only one participant found input syntax unclear. One reason for confusion was unclarity on whether the tool is case sensitive or not. Another reason was too broad choice of how different connectives can be input. To avoid this issue when presenting *Praωf* to students, only a limited number of options should be given. This does not mean reducing the number of options that the tool supports in general. More experienced users may appreciate the availability of options, which are similar to those they use when working in other proof assistants.

All participants agreed that the proof tree and available assumptions are displayed efficiently. However, one participant suggested that since some proofs may have a long list of assumptions

it might useful to have "Assumptions available" text displayed on the level of the last added assumption, i.e. aligned with the top of the assumptions list and not centred as below.

$$\begin{array}{rrl} & u5: & E \\ & u4: & D \\ \text{Assumptions available:} & u3: & C \\ & u2: & B \\ & u1: & A \end{array}$$

Another participant suggested that terms should be highlighted where substitution of quantified variables by terms takes place.

When asked whether any functionalities are missing, two experienced participants suggested that they would like to be able to select a goal to work on from a list of pending goals. Being generally satisfied with the available functionalities, one of these participants was, nevertheless, also interested in whether forward reasoning can be incorporated into the prover.

Another experienced participant suggested that there should be a more efficient way for adding assumptions before starting a proof. This idea and a related idea of creating an option for a bulk implication introduction rule are described further in the suggested future improvements section.

One of the participants encountered a difficulty understanding the format in which a missing formula needs to be added when applying the ∀-Elimination rule. When the rule was applied, *Praωf* instructed the user: `Enter the quantified formula>`. The expected input was a formula of the form `all x A(x)`, however the user typed in only `A(x)`. To avoid similar situations, unclear instructions like this one have been re-worded and now include the format of expected formulas.

The participants were also asked to add any additional suggestions that they had. One of them commented that the user interface was too serious. They suggested that the interface could be made a bit more graphical as not all users like to work in a command-line environment.

Figure 3.9 on page 46 shows participants' responses to each of the questions in the questionnaire.

Overall, based on discussions with all participants, all of them found the tool useful for educational purposes. Their feedback was used to discover potential areas of improvement. As the user study was conducted very close to the end of the project, only minor corrections were made in *Praωf* based on the feedback received.

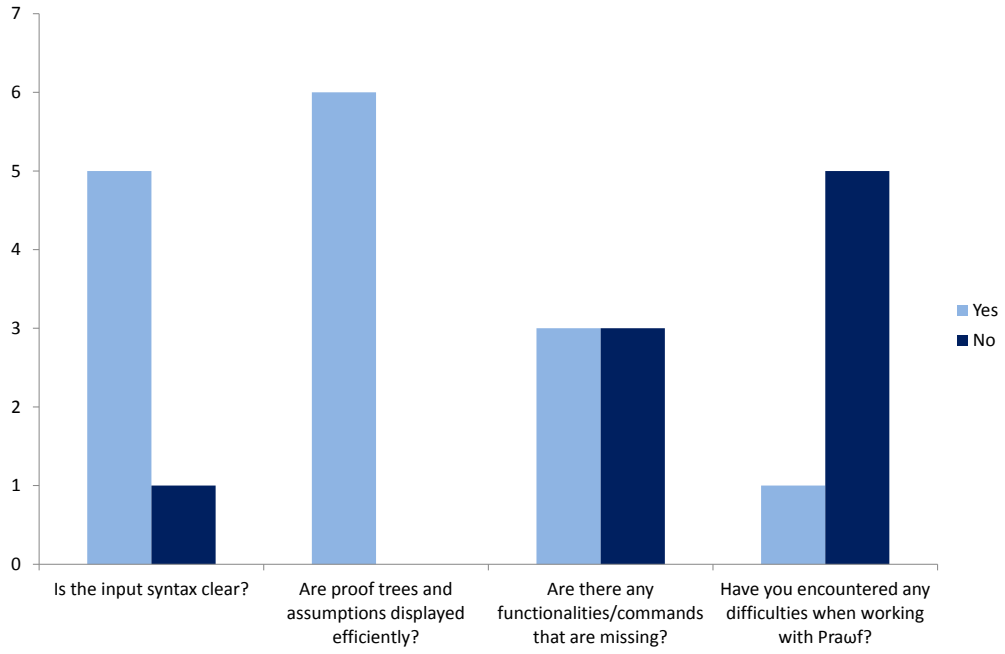

Figure 3.9: Participants' Responses to the Questionnaire

Chapter 4

Future Improvements

Praωf fulfils all the requirements set at the start of the project. Nevertheless, there are some areas where its functionalities can be extended in the future. These ideas for future improvements are based on experience during the testing process as well as on feedback from the user study. Each area is described below.

Richer Term Language Currently the only terms that *Praωf* supports are constants, variables and function application. In the future the term language could be extended further to support λ-calculus.

Proof Tree Insertion Future versions of *Praωf* may also be updated to support proof tree insertion. Users would work on smaller proofs and would save them as theorems with corresponding proof trees. These proof trees could then be inserted into other proof trees to save time. The system, however, needs to ensure that in case of such insertions the reasoning stays sound.

Shrinking/Expanding of Tree Branches This functionality would enable working with bigger proof trees that do not fit in the window. It may require reconsidering the technology behind proof display and potentially moving away from LaTeX to some other more efficient display option.

Provability Checks In *Proof Editor* provability of a goal formula is checked at each step. Therefore, if a user applies a rule that makes it impossible to prove the formula they will be

notified that the goal is unprovable. It is possible to perform such checks in *Proof Editor* propositional logic. However, it is not fully possible to do it in *Praωf* due to undecidability of predicate logic. Currently *Praωf* runs provability checks only on propositional formulas. However, it would be beneficial to add limited unprovability checks for predicate logic. These would help users to avoid spending time on unprovable goals.

Assumptions Pre-loading and Bulk Implication Introduction At the moment it is not possible to pre-load assumptions. In order to get the all assumptions recognised by the system as assumptions, it is necessary to input them as a sequence of implications. Thus, in order to prove B under assumptions A, B and C ($A, B, C \vdash B$), users need to input `A -> B -> C -> B` into the prover. Then the implication introduction rule needs to be applied three times to get A, B and C recognised as assumptions, as shown in the proof tree below.

The input would be as follows.

```
Enter goal formula X > A -> B -> C -> B
Enter command> impi u1
Enter command> impi u2
Enter command> impi u3
```

This input would produce a proof tree as below:

$$\dfrac{\dfrac{\dfrac{?\mathbf{3} : B}{C \to B} \to^{+} u3 : C}{B \to (C \to B)} \to^{+} u2 : B}{A \to (B \to (C \to B))} \to^{+} u1 : A$$

This is helpful from the educational perspective for novice users to reinforce their learning. However, it may save time to have an option to pre-load existing assumptions. This would also reduce the size of proof trees.

Another possibility is to allow bulk implication introduction. Thus, instead of 3 steps, it would be possible to take just one: `impi u1 u2 u3`. And the corresponding proof tree could have the following shape:

$$\dfrac{?\mathbf{1} : B}{A \to (B \to (C \to B))} \to^{+} u1 : A;\ \mathrm{u2} : \mathrm{B};\ u3 : C$$

This gives benefits similar to those of assumptions pre-loading.

Switching Between Goals In *Praωf* a proof tree is constructed by working on a *current goal*, which is always the leftmost unproven leaf of the proof tree. However, as suggested by some participants of the users study, it can be useful if users could decide on which of the pending goals they would like to work. For example, in the below proof tree a user might want to work on the goal number 5 first. Since all goals are labelled, it should not be complicated to select a goal that a user prefers to work on.

$$\dfrac{\dfrac{\dfrac{?\mathbf{4} : A \to B(x) \qquad ?5 : A}{B(x)} \to^{-}}{\forall x B(x)} \forall^{+}}{\dfrac{A \to \forall x B(x)}{\forall x(A \to B(x)) \to (A \to \forall x B(x))} \to^{+} u2 : A} \to^{+} u : \forall x(A \to B(x))$$

Being able to select a goal gives users more flexibility on how they want to approach the proof. In some cases they may want to get a simpler goal out of the way first. In other cases they may want to see if the later goal is provable before spending effort on simpler goals, which they expect to be provable.

Forward Reasoning Approach Currently *Praωf* only allows applying rules in a backward reasoning mode. Adding support for forward reasoning could be beneficial as it will give students a deeper understanding of natural deduction. This may be a difficult task to achieve but could make an interesting project topic in the future.

User Interface In order to create a more engaging and intuitive user interface, a dedicated user experience study in a larger group of students may be needed. One of the possible options would be making a web-based version of *Praωf* using a script on the client side and *Haskell* on the server side (similar to *ProofWeb*).

Support of Mac OS As previously mentioned, *Praωf* can be extended to support Mac OS. Since software requirements are low (interactive *Haskell* environment and ability to open LaTeX files), this should be a trivial process. It would require creating a special module for Mac environment and adding minor changes to the Prover module.

Program Extraction *Praωf* was developed as a learning tool for the Logic course. However, in the future it may be used for broader purposes. Potential program extraction functionality

could make it useful for research purposes, namely those related to the ongoing *CORCON (Correctness by Construction)* project [8].

Chapter 5

Conclusion

Although there exist many useful interactive theorem provers for predicate logic, none of them sufficiently matches all the aspects of the Logic course as taught at Swansea University. Therefore, the main aim of this project was to develop a proof assistant for predicate logic on the basis of the existing *Proof Editor* for propositional logic, which is currently used as a learning tool in the course. This aim has been achieved successfully and four additional rules, $\forall$-Introduction, $\forall$-Elimination, $\exists$-Introduction, $\exists$-Elimination, have been added to the proof assistant. This required adding an infrastructure to support application of these rules, which included introduction of substitution and alpha-equality checks. *Praωf's* input language supports full general first-order terms.

The main focus of this new proof assistant is educational. Hence, the most important feature of such a tool is clarity of input and output. An educational tool should be easy to use and the information that it produces should be easily understandable. The results of the user study have shown that most of the participants agree that input syntax and output of proofs in Gentzen-style trees fulfil those requirements. Representing proofs as trees using the rules, which follow the exact style used in the course notes, facilitates and reinforces understanding of the course material.

Another aim of the project was to improve readability of the source code. Although not explicitly mentioned in the previous chapters, this task has also been completed through adding comments and splitting of larger functions into sets of smaller sub-functions.

Documentation of the tool has been improved and extended. The *Praωf* user manual is available on a web page specifically created for the project [27]. It contains information regarding software requirements, starting instructions, input syntax, control and proof commands; it

also gives an example of a session during which all four newly added rules are applied.

Possible further improvements include a richer term language, program extraction functionality, and various additional features such as proof tree insertion, provability checks, or assumptions pre-loading.

Bibliography

[1] Patrice Bailhache. *Program to learn Natural Deduction in Gentzen-Kleene's style*. Accessed 25/08/2016. URL: http://patrice.bailhache.free.fr/dnfn/deductioneng.html.

[2] Jon Barwise and John Etchemendy. *Hyperproof*. Accessed on 25/08/2016. 1994. URL: https://web.stanford.edu/group/cslipublications/cslipublications/site/1881526119.shtml.

[3] Jon Barwise and John Etchemendy. "Hyperproof: logical reasoning with diagrams". In: *Stanford University*. 1992, pp. 80–84.

[4] Ulrich Berger. "CSC375/CSCM75 Logic for Computer Science". Lecture notes. 2015.

[5] Ulrich Berger. *Proof editor source files and documentation*. Accessed on 22/02/2016. 2015. URL: http://www-compsci.swan.ac.uk/~csulrich/ftp/logic/prover/.

[6] Wolfgang Bibel. "Early History and Perspectives of Automated Deduction". In: *KI 2007: Advances in Artificial Intelligence: 30th Annual German Conference on AI, KI 2007, Osnabrück, Germany, September 10-13, 2007. Proceedings*. Ed. by Joachim Hertzberg, Michael Beetz, and Roman Englert. Berlin, Heidelberg: Springer, 2007, pp. 2–18. ISBN: 978-3-540-74565-5. DOI: 10.1007/978-3-540-74565-5_2.

[7] Austen Clark. *The Bertie/Twootie Home Page*. Accessed on 22/08/2016. URL: http://selfpace.uconn.edu/BertieTwootie/software.htm.

[8] *Correctness by Construction CORCON*. Accessed on 30/07/2016. URL: https://corcon.net/about/.

[9] *Educational Logic Software*. Accessed on 22/08/2016. URL: http://www.ucalgary.ca/aslcle/logic-courseware.

[10] Frederic Brenton Fitch. *Symbolic logic: an introduction*. Ronald Press Co., 1952.

[11] Branden Fitelson. *Running MacLogic Under Emulation*. Accessed on 22/08/2016. URL: http://fitelson.org/maclogic.htm.

[12] Graeme Forbes. *A Brief Guide to MacLogic*. Accessed on 23/08/2016. URL: http://spot.colorado.edu/~forbesg/pdf_files/MacLogic_Guide.pdf.

[13] Olivier Gasquet, François Schwarzentruber, and Martin Strecker. "Panda: A Proof Assistant in Natural Deduction for All. A Gentzen Style Proof Assistant for Undergraduate Students". In: *Tools for Teaching Logic: Third International Congress, TICTTL 2011, Salamanca, Spain, June 1-4, 2011. Proceedings*. Ed. by Patrick Blackburn et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 85–92. ISBN: 978-3-642-21350-2. DOI: 10.1007/978-3-642-21350-2_11. URL: http://dx.doi.org/10.1007/978-3-642-21350-2_11.

[14] Herman Geuvers. "Proof assistants: History, ideas and future". In: *Sadhana* 34.1 (2009), pp. 3–25. ISSN: 0973-7677. DOI: 10.1007/s12046-009-0001-5.

[15] *Haskell*. Accessed on 05/03/2016. URL: https://www.haskell.org/.

[16] *Isabelle*. Accessed on 01/03/2016. URL: https://isabelle.in.tum.de/.

[17] *LaTeX - A document preparation system*. Accessed on 29/02/2016. URL: https://www.latex-project.org/.

[18] Edward John Lemmon. *Beginning Logic*. London: Nelson, 1965.

[19] *Logic Daemon*. Accessed on 17/08/2016. URL: http://logic.tamu.edu/daemon.html.

[20] *MacLogic*. Accessed on 22/08/2016. URL: https://rd.host.cs.st-andrews.ac.uk/logic/mac/.

[21] Annapaola Marconi, Luciano Serafini, and Chiara Ghidini. *Mathematical Logic Excercises (Academic Year 2012-2013)*. Accessed on 17/07/2016. URL: http://disi.unitn.it/~ldkr/ml2013/MLexercises.pdf.

[22] *Mizar Home Page*. Accessed on 01/03/2016. URL: http://mizar.uwb.edu.pl/.

[23] *Module Syllabus, PY1012: Reasoning*. Accessed on 27/08/2016. University of St Andrew. 2016. URL: http://www.st-andrews.ac.uk/~eg35/1012syllabus.pdf.

[24] *New Pandora*. Acessed on 15/08/2016. URL: https://www.doc.ic.ac.uk/pandora/newpandora/.

[25] *Panda : Proof Assistant for Natural Deduction for All*. Accessed on 25/08/2016. URL: https://www.irit.fr/panda/.

[26] Jan von Plato. *The Development of Proof Theory, Stanford Encyclopedia of Philosophy*. Accessed on 09/09/2016. Center for the Study of Language and Information (CSLI), Stanford University. 2014. URL: http://plato.stanford.edu/entries/proof-theory-development/.

[27] *Praωf: Proof Assistant for Predicate Logic*. Accessed on 26/09/2016. Swansea University. 2016. URL: https://prawftree.wordpress.com/.

[28] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Dover Publications, 1965.

[29] *ProofWeb*. Accessed on 17/08/2016. URL: http://proofweb.cs.ru.nl/login.php.

[30] Peter Smith. *bussproofs.sty: A User Guide*. Accessed on 03/03/2016. Mar. 2012. URL: http://www.math.ucsd.edu/~sbuss/ResearchWeb/bussproofs/BussGuide2_Smith2012.pdf.

[31] Peter Smith. *Types of proof system*. Accessed on 05/03/2016. Oct. 2010. URL: http://www.logicmatters.net/resources/pdfs/ProofSystems.pdf.

[32] *The Coq Proof Assistant*. Accessed 01/03/2016. URL: https://coq.inria.fr/.

[33] Markus Wenzel. "Isar — A Generic Interpretative Approach to Readable Formal Proof Documents". In: *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs' 99 Nice, France, September 14–17, 1999 Proceedings*. Ed. by Yves Bertot et al. Berlin, Heidelberg: Springer, 1999, pp. 167–183. ISBN: 978-3-540-48256-7. DOI: 10.1007/3-540-48256-3_12.

Appendix A

User Study Handout

Praωf is a simple proof assistant for learning natural deduction. It supports natural deduction rules of propositional logic and predicate logic (see page 59) applied in backwards reasoning mode. Proofs are displayed as Gentzen-style proof trees.

Prove the following formulas using the rules of natural deduction:

- $(P \wedge Q) \to (Q \wedge P)$
- $\forall x(A \to B(x)) \to (A \to \forall x B(x))$
- $\forall x A(x) \to \neg\exists x(\neg A(x))$

To enter a chosen formula in *Praωf* (GHCi), refer to the table below. Then press *Enter* and proceed with the proof using the proof and control commands provided on page 60. In order to see the proof tree updating, move the cursor over the *Yap* window.

Connectives & Quantifiers	Possible Input Options			Examples	
$\wedge$	`and`	`&`	`\land`	$A \wedge B$	`A and B`
$\vee$	`or`	`|`	`\lor`	$A \vee B$	`A or B`
$\bot$	`bot` or `Bot` or `F`	`_|_`	`\bot`	$\bot$	`bot`
$\neg$	`not`	`-`	`\neg`	$\neg A$	`not A`
$\forall$	`all` or `All`	`For all`	`\forall`	$\forall x\, A(x)$	`all x A(x)`
$\exists$	`ex` or `Ex`	`Exists`	`\exists`	$\exists x\, A(x)$	`ex x A(x)`
$\to$		`->`	`\to`	$A \to B$	`A -> B`

* Negation can also be written as implication: `not A -> B` is the same as `(not A) -> B` and the same as `(A -> bot) -> A`. Please note that *Praωf* automatically converts $\neg A$ to $A \to \bot$.

Once completed, please answer the questions on page 61.

Example
To prove $\forall x A(x) \to A(x)$, the steps are the following:

```
Enter goal formula X > ex x (A(x)) -> all x (A(x) -> B(x)) -> ex x B(x)
Enter command> impi u1
Enter command> impi u2
Enter command> exe ex x A(x)
Enter command> use u1
Enter command> alli
Enter command> impi u3
Enter command> exi x
Enter command> impe A(x)
Enter command> alle all x (A(x) -> B(x))
Enter command> use u2
Enter command> use u3
Proof complete.
```

It will generate the following tree:

$$\dfrac{u1 : \exists x A(x) \qquad \dfrac{\dfrac{\dfrac{u2 : \forall x(A(x) \to B(x))}{A(x) \to B(x)}\forall^- \qquad u3 : A(x)}{B(x)}\to^-}{\exists x B(x)}\exists^+}{\dfrac{\dfrac{A(x) \to \exists x B(x)}{\forall x(A(x) \to \exists x B(x))}\to^+ u3 : A(x)}{\forall x(A(x) \to B(x)) \to \exists x B(x)}\forall^+}\exists^- \qquad u2 : \forall x(A(x) \to B(x))$$
$$\dfrac{}{\exists x A(x) \to (\forall x(A(x) \to B(x)) \to \exists x B(x))}\to^+ u1 : \exists x A(x)$$

Assumption rule	$\dfrac{}{\Gamma, A \vdash A}\ \text{use}$	
	Introduction rules	Elimination rules
$\wedge$	$\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge^{+}$	$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge_{\mathrm{l}}^{-} \qquad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge_{\mathrm{r}}^{-}$
$\rightarrow$	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\ \rightarrow^{+}$	$\dfrac{\Gamma \vdash A \rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \rightarrow^{-}$
$\vee$	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee_{\mathrm{l}}^{+} \qquad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee_{\mathrm{r}}^{+}$	$\dfrac{\Gamma \vdash A \vee B \qquad \Gamma \vdash A \rightarrow C \qquad \Gamma \vdash B \rightarrow C}{\Gamma \vdash C}\ \vee^{-}$
$\bot$		$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \text{efq} \qquad \dfrac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\ \text{raa}$
$\forall$	$\dfrac{\Gamma \vdash A(x)}{\Gamma \vdash \forall x A(x)}\ \forall^{+}$ (*)	$\dfrac{\Gamma \vdash \forall x A(x)}{\Gamma \vdash A(t)}\ \forall^{-}$
$\exists$	$\dfrac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x A(x)}\ \exists^{+}$	$\dfrac{\Gamma \vdash \exists x A(x) \qquad \Gamma \vdash \forall x\,(A(x) \rightarrow B)}{\Gamma \vdash B}\ \exists^{-}$ (**)

(*) $\forall$-Introduction rule is subject to the condition that x should not occur free in Γ.

(**) $\exists$-Elimination rule is subject to the condition that x should not occur free in B.

Proof commands in backwards reasoning mode.

`use u`	use assumption `u`	
`andi`	And introduction	$\wedge^+$
`andel`	And elimination left	$\wedge_l^-$
`ander`	And elimination right	$\wedge_r^-$
`impi u`	Implication introduction	$\rightarrow^+ u$
`impe`	Implication elimination	$\rightarrow^-$
`impe A`	Implication elimination with premise	$\rightarrow^-$
`oril`	Or introduction left	$\vee_l^+$
`orir`	Or introduction right	$\vee_r^+$
`ore`	Or elimination	$\vee^-$
`efq`	ex-falso-quodlibet	efq
`raa`	reductio-ad-absurdum	raa
`alli`	All introduction	$\forall^+$
`alle all x A(x)`	All elimination with a quantified formula	$\forall^-$
`alle`	All elimination	$\forall^-$
`exi`	Ex introduction	$\exists^+$
`exi t`	Ex introduction with a term to substitute the bound variable	$\exists^+$
`exe ex x A(x)`	Ex elimination with an existential formula	$\exists^-$
`exe`	Ex elimination	$\exists^-$

Control commands:

`undo`	undo a proof step
`quit`	leave the prover
`new`	start a new proof (without saving your current proof)
`?`	more explanations on the commands above

1. Is the input syntax clear? Yes ☐ No ☐ If not, what could be improved?

2. Are the proof tree and available assumptions displayed efficiently? Yes ☐ No ☐ If not, what could be improved?

3. Are there any functionalities/commands that are missing? Yes ☐ No ☐ If yes, what would you find helpful?

4. Have you encountered any difficulties when working with *Prawf*? Yes ☐ No ☐ If yes, what are those?

5. Please add any other comments or suggestions that you might have below: