University of Sheffield

Implementing A Natural Deduction Proof Assistant In Haskell

Abdur-Rahman Muhsin

Supervisor: Professor Georg Struth

This report is submitted in partial fulfilment of the requirement for the degree of BSc in Computer Science.

in the

Department of Computer Science

July 8, 2024

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Abdur-Rahman Muhsin

Signature: Abdur-Rahman Muhsin

Date: 08/05/2024

Abstract

Natural deduction is a method of solving proofs that closely resembles how humans naturally reason about the world. It is widely used in applications such as formal verification and automated reasoning. This paper aims to introduce a new framework and algorithm for computing natural deduction using the Haskell functional programming language.

Acknowledgements

I would like to express great thanks to my mother for her incredible support and invaluable advice throughout my dissertation. The dissertation would surely not be possible without you. Thank you for always being there to listen to my problems and guiding me through them. I would also like to thank my supervisor, Professor Struth, for all his expert guidance throughout the project and even outside of it.

Contents

Chapter 1

Introduction

1.1 Motivation

For centuries, mathematicians and logicians have strived to develop robust and clear reasoning and proof construction methods. While axiomatic systems have provided a robust foundation, their abstract nature can sometimes feel distant from the intuitive steps of human reasoning. Natural deduction systems bridge this gap by offering a framework that mimics the natural flow of logical argumentation.

The origins of natural deduction lie in the 1930s with the independent works of Gerhard Gentzen [5] and Stanisław Jaśkowski [8]. Dissatisfied with the complexity and perceived artificiality of axiomatic proofs, they sought to capture the essence of how we reason in everyday discourse. Their formal systems, built on intuitive inference rules, mirrored the natural steps we take when constructing arguments.

Natural deduction has flourished since its inception. Today, it serves a multitude of purposes in formal verification and automated reasoning. It is also widely used as an educational tool for teaching logic due to its intuitive structure. By following the clear inference rules, students can readily grasp the process of constructing valid arguments and identifying fallacies.

1.2 Project aims

This project aims to formulate a framework for natural deduction in the Haskell programming language. Its primary purpose is to serve as an interactive proof assistant for second-year students studying the *Logic in Computer Science* (LICS) module, allowing them to practice and experiment with the natural deduction process and better understand it. By providing helpful guidance for syntax errors and errors made during the proof process, this system will become a valuable resource to students. To ensure the system is relevant and accessible to students, it will model the syntax, rules and proof process as closely as possible to those shown in the course.

This project will:

- Introduce a new way of viewing seqents and inference rules, repurposing them as computational constructs.
- Introduce a framework for computing natural deduction.
- Design an algorithm for the natural deduction proof process.
- Design a powerful and versatile parser for parsing user input.

1.3 Report structure

- **Chapter 2: Background** - Provides a logic and natural deduction primer.
- **Chapter 3: Design** - Translates concepts from natural deduction into computational concepts.
- **Chapter 4: Implementation** - A detailed walk-through of the implementation of the natural deduction framework.
- **Chapter 5: Evaluation** - Discusses the project's results, identified limitations, and possible extension points.
- **Chapter 6: Conclusion** - A summary of the entire project.

1.4 Relation to the degree programme

Many concepts taught throughout the degree are incorporated within this project. Some of these, such as the functional programming concepts seen through the Haskell implementation and the logic concepts at the heart of the project, are plain to see. The project also includes many concepts from the theory of automata and formal languages, such as context-free grammar, parsers, states, and transition functions, whose relation to the project is not immediately apparent.

This project is so closely tied to the degree program as it is directed towards students taking a module in it. This is a great help, as many resources, such as course notes and practice sheets from the course, were used to design specifications for the project and test its implementation. It also means that many students can use and test the system and provide feedback to improve it, though no students have currently used it.

Chapter 2

Background

This chapter briefly reviews propositional and first-order logic and natural deduction taught to second-year computer science students studying the (LICS) module. It summarises information found in chapters of the lecture notes for the module [13]. Gerhard Gentzen's paper on natural deduction [5] is also referenced throughout the chapter.

2.1 Propositional logic

Propositional logic revolves around propositions, which are true or false statements. They are represented in propositional formulae using variables: $p, q, r, ...$ from a countably infinite set P. Schematic variables: $\varphi, \psi, \chi, ...$ can be used in place of propositional variables and are used when discussing a formula's syntax (structure) to divert attention away from its semantics (meaning). Simple (atomic) propositions can be combined using logical connectives: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ to express more complex statements, and the BNF (Backus-Naur Form) grammar to do so is recursively defined: for all $p \in P$,

$$\Phi ::= \top \mid \bot \mid p \mid (\neg\Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi) \mid (\Phi \rightarrow \Phi) \tag{2.1}$$

2.2 First-order logic

First-order (predicate) logic extends propositional logic by introducing the concept of *entities*, *quantifiers*, and *relations*. This allows for a more expressive and nuanced representation of logical statements and makes it possible to reason about the entities within statements and the relationships between them.

Entities found in predicate formulae can be represented using *variables* from the countably infinite set $\mathcal{V}$ of logical variables or as *constants*. Variables act as placeholders for entities, and constants represent the entities themselves. Collectively, constants and variables are known as atomic terms. They can be applied to functions from a set $\mathcal{F}$ of function symbols, which output terms. An example of a function might be *add*, which adds together two integers,

and whose the domain of discourse (the set of entities a predicate formula is concerned with) is the set of integers. The BNF grammar to form terms is recursively defined: let Σ be a signature, then for all $x \in \mathcal{V}$ and $f \in \mathcal{F}$,

$$\mathcal{T}_\Sigma ::= x \mid f(\mathcal{T}_\Sigma, ..., \mathcal{T}_\Sigma) \tag{2.2}$$

In the definition above, there is no sign of constants, as they are represented as functions with an arity of zero.

Relations over terms are expressed in the form of *predicates* from a set $\mathcal{P}$ of predicate symbols. Though it may be confusing, the terms *relation* and *predicate* are used interchangeably even when they mean different things. A relation refers to the relationship between two or more entities, e.g. *Love(x, y)*, which translates to "x loves y." On the other hand, predicates refer to an attribute of an entity, e.g. *Red(x)* or *IsEven(x)*. As predicates are a special type of relation, which are only over a single term, this report will refer to both predicates and relations simply as relations.

The signature Σ specifies all the function and relation symbols that can be used in a Σ-formula. Function and relation symbols in a signature must have a fixed arity. Predicate formulae are recursively defined: let Σ be a signature, then for all $t_1, ..., t_n \in \mathcal{T}_\Sigma$ and $P \in \mathcal{P}$,

$$\begin{aligned}\Phi_\Sigma ::= \top \mid \bot \mid t_1 = t_2 \mid P(t_1, ..., t_n) \mid \\ (\neg\Phi_\Sigma) \mid (\Phi_\Sigma \wedge \Phi_\Sigma) \mid (\Phi_\Sigma \vee \Phi_\Sigma) \mid (\Phi_\Sigma \to \Phi_\Sigma) \mid \forall x.(\Phi_\Sigma) \mid \exists x.(\Phi_\Sigma)\end{aligned} \tag{2.3}$$

The operators of predicate logic follow an *order of precedence*. This order gives operators with higher precedence the right to be applied before those with lower precedence. The order of precedence is: (φ), $[\neg, \forall, \exists]$, $[\wedge, \vee]$, $\to$ where bracketed formulae are of the highest precedence. Operators grouped into square brackets are in the same precedence level, which can lead to situations where there are multiple different ways of interpreting a formula. These formulae are said to be *ambiguous*. As ambiguous formulae are not well-formed, a strategy is needed to ensure they can be consistently interpreted in the same way.

A quantified formula, one which is enclosed in the brackets of either a *universal* ($\forall$) or an *existential* ($\exists$) quantifier, is said to be *bound* by it. Specifically, the quantifier $Qx.(\varphi)$ binds all occurrences of the variable x within φ. All variables within the bound formula fall into the *scope* of the binding quantifier. For example, the variable y in the formula $\exists y.(P(x, y))$ is in the scope of the existential quantifier but not bound by it; however, the variable x is. Whether a variable is bound or free (not bound) is significant when substituting for a term. This is because only terms free for that variable in a formula (i.e. ones that do not contain variables that are bound in the formula) can be substituted into it. The substitution function below shows how terms are substituted into formulae and the conditional that prevents terms that contain bound variables from substituting.

$$
\begin{aligned}
\bot[t/x] &= \bot, \\
\top[t/x] &= \top, \\
P(t_1,\ldots,t_n)[t/x] &= P(t_1[t/x],\ldots,t_n[t/x]), && \text{if } \varphi = P(t_1,\ldots,t_n), \\
(s_1 = s_2)[t/x] &= (s_1[t/x] = s_2[t/x]), && \text{if } \varphi = (s_1 = s_2), \\
(\neg\psi)[t/x] &= \neg(\psi[t/x]), && \text{if } \varphi = \neg\psi, \\
(\psi_1 \diamond \psi_2)[t/x] &= \psi_1[t/x] \diamond \psi_2[t/x], && \text{if } \varphi = \psi_1 \diamond \psi_2 \text{ for } \diamond \in \{\wedge, \vee, \rightarrow\}, \\
(Qy.\ \psi)[t/x] &= \begin{cases} Qy.\ (\psi[t/x]) & \text{if } x \neq y, \\ Qy.\ \psi & \text{if } x = y, \end{cases} && \text{if } \varphi = Qy.\ \psi \text{ for } Q \in \{\forall, \exists\}.
\end{aligned}
$$

Figure 2.1: Substitution function. Source: [13]

Figure 2.1 above shows a recursive function defined over predicate formulae. Many others like it can be defined with many different uses, e.g. the function BV, which computes the set of bound variables in a formula.

Multiple consecutive quantifiers may bind a formula, known as mixed quantification. The order in which they appear is significant to the interpretation of the formula. Consider the formula $\forall x \exists y. Likes(x, y)$. It can be read: "Everyone likes someone." However, when the order of the quantifiers is swapped, it reads: "Someone likes everyone," which has a completely different meaning. When mixed quantifiers bind the same variable, the formula they enclose is only bound to the scope of the innermost quantifier by convention. Therefore, the formula $\forall x \exists x P(x)$ is bound by the inner existential quantifier. There are also instances where quantifiers are not mixed, but a quantifier inside the scope of another binds the same variable. In these cases, the convention of applying the quantifier closest to the formula still applies. For example, in the formula $\forall x(\exists x(P(x)) \wedge Q(x))$, the x within P is quantified by the inner existential quantifier, and the outer universal quantifier quantifies the x within Q. Formulae involving multiple quantifiers binding the same variable in a scope are difficult to read and understand and can be *renamed* to ensure this is not the case. The process of renaming variables is given in [13].

2.3 Natural deduction

"I intended, first of all, to set up a formal system which came as close as possible to actual reasoning. The result was a 'calculus of natural deduction'." - Gerhard Gentzen [5].

2.3.1 Judgements

A judgement is an assertion made in the *metalanguage* about a particular statement. The metalanguage is a language used to describe another; in this case, the language being described would be either propositional or predicate logic. Two important judgements are

made on formulae in natural deduction: φ *is a proposition* (φ_{prop}) and φ *is true* (φ_{true}).

2.3.2 Sequents

Often, judgements of truth cannot be made unconditionally and depend on the truth of others. Such judgements are known as *hypothetical judgements*. Hypothetical judgements can be written as sequents: $J_1, ..., J_n \vdash J$ where $J_1, ..., J_n$ are the *antecedents* (also called the *hypotheses*, *premises* or *assumptions*), and J is the *consequent* (or *conclusion*). The hypotheses of a derivation are the known truths, while the consequent is the judgement *derived* from them. The process of proving the truth of a consequent from the hypotheses is known as a *derivation* or a *proof*.

The hypotheses are a finite set [4], and Greek uppercase letters Γ, Δ, Θ are often used in place of them as a shorthand: $\Gamma \vdash \varphi_{true}$. Where particular hypotheses from the set are to be highlighted, they can be written separately to the set: $\Gamma, \varphi_{true}, \psi_{true}, ... \vdash \chi_{true}$. The Γ, in this case, represents the rest of the hypotheses. While the assertion of truth over formulae is made explicit here, it can be assumed, and the subscript can be dropped. Some sequents do not depend on the truths of prior assumptions and are known as *unconditional assertions* or *axioms*: $\vdash \varphi$. They are the original set of assumptions, and their set of hypotheses is the empty set. The system of natural deduction described here is the system $\mathcal{NJ}$, identified by Gentzen in [**?**]. This system restricts the conclusion to only a single formula. Gentzen also identified the system $\mathcal{NK}$, which forgoes this restriction [FERSCO], but this system is not used in the LICS module and, thus, will not be a concern of the proof assistant.

2.3.3 Inference rules

Inference rules govern the process of deriving new logic statements from existing ones. They specify how to make valid deductions or inferences from assumptions. Gentzen portrayed inference rules using *inference figures*. The inference figure for the conjunction introduction rule is shown below:

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi} \wedge\text{I} \tag{2.4}$$

The formulae at the top of the inference figure (upper formulae) are the assumptions, and the formula at the bottom (lower formula) is the conclusion. The name of the inference rule is on the right side of the figure. Proof figures can also be written as sequents, where the figure above would translate to the sequent: $\varphi, \psi \vdash \varphi \wedge \psi$. The conjunction introduction rule says, "If φ and ψ are true (i.e. proven hypotheses), then $\varphi \wedge \psi$ is true."

If an upper formula were part of a proof's original set of hypotheses (an assumption), it would not need to be derived. It must be derived using another inference figure if it is not an assumption. This creates a *tree* of inference figures, where the *leaves* are the assumptions

of the proof and the *root* is its conclusion. This style of displaying natural deduction proofs was used by Gentzen and is called *tree notation*.

$$\dfrac{\neg p \vee q \qquad \dfrac{\dfrac{\dfrac{[p]_b \quad [\neg p]_a}{\bot}\,\neg E}{q}\,\bot E}{p \to q}\,\to I_b \qquad \dfrac{[q]_a}{p \to q}\,\to I}{p \to q}\,\vee E_a$$

Figure 2.2: Tree notation of $\neg p \vee q \vdash p \to q$

Some inference figures, such as implication introduction, prove their consequent indirectly through what is known as a subproof. Subproofs are not separate proofs but are an inner scope within a proof. This means that they can use any assumptions known to the conclusion of the outer scope, but any derivations made inside of them cannot be used in outer scopes. The notion of scopes is important as inference rules involving subroofs often add local assumptions into the set of hypotheses, which can be used to derive their conclusions but are discharged as soon as they are complete. Local assumptions can be seen in the proof tree above, which are the assumptions surrounded by square brackets. The implication introduction rule is shown below:

$$\dfrac{\begin{array}{c}[\varphi]\\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\,\to \mathrm{I} \tag{2.5}$$

The vertical ellipses signify that there must be a derivation from the local assumption φ to the subproof's *goal* (the consequent) ψ. A subproof's goal can be different from that of the outer proof.

In predicate logic, inference rules may also introduce *new free variables* or *fresh parameters* into the scope of a subproof. These variables are new in that they would not have been seen before in the proof up to the opening of the new subproof and, thus, cannot possibly be bound in any formulae. As with local assumptions, they become forgotten when the subproof is exited. The existential elimination rule shown below is an inference rule that opens up an inner scope with a new free variable.

$$\dfrac{\exists x.\varphi \qquad \begin{array}{c}[\varphi[a/x]]\\ \vdots \\ \psi\end{array}}{\psi}\,\exists \mathrm{E} \tag{2.6}$$

One final observation on inference rules is that they all have a specific direction in which they are read or applied. The conjunction introduction rule is applied top-down (from the upper formulae to the lower one). Top-down rules derive a formula from assumptions.

However, bottom-up rules, like implication introduction, divert a proof into a subproof, and the derivation of their consequent depends on whether or not the subproof can be proven. The outer proof is completed as soon as the scope returns from the subproof.

2.4 Notations

There are many styles of natural deduction, and the proof tree is one of them. While they help show how each formula within a proof was derived, proof trees can become unwieldy as proofs become larger and more complex. This is especially so when formulae with lengthy derivations are used in different parts of a proof, as the notation has no concept of state, so they have to be derived again from scratch whenever they are to be used. This clutters and visually obscures the notation.

An alternative is the *Fitch notation* developed by Fitch and adapted from the nested boxes that Jaśkowski invented. It fixed many of the issues present in Gentzen's tree notation. It places each formula as a *line* in the proof, which is annotated with information on which other lines, if any, it was derived from. These lines make it possible to refer back to previously derived formulae without re-deriving them like in tree notations. The boxes around subproofs also made it very clear to see the different scopes in a proof and the local assumptions and variables available to use.

1.	$\neg p \lor q$	hyp
2.	$\neg p$	hyp
3.	p	hyp
4.	$\bot$	$\neg E$, 2,3
5.	q	$\bot E$, 4
6.	$p \to q$	$\to I$, 3-5
7.	q	hyp
8.	p	hyp
9.	q	copy, 7
10.	$p \to q$	$\to I$, 8,9
11.	$p \to q$	$\lor$ E, 1, 2-6, 7-10

Figure 2.3: Fitch notation of $\neg p \lor q \vdash p \to q$

Chapter 3

Design

This chapter begins by describing how concepts from natural deduction can be translated into computational concepts. An algorithm for computing natural deduction is then defined, followed by the introduction of a new notation based on this algorithm.

3.1 Sequents

Sequents are data structures storing the set of hypotheses within the scope of a proof, along with the conclusion that the proof aims to reach. As they encapsulate all the essential information about a proof's current state, sequents can be used to reflect the state of a proof. In predicate logic, the sequent data structure can be extended to also encompass the set of free variables within a proof's scope, giving it the form: $(V, \Gamma) \vdash c$. The words sequent and state will now be used synonymously.

3.2 Inference rules

If sequents are perceived as states in a proof, then inference rules can be understood as the transition functions that describe how a proof should advance from a given state. These sequent transition functions take in a state as input and output the the next proof state (or states). However, to use the inference rules of natural deduction as state transition functions, they must be translated from functions over formulae to functions over sequents. This translation is quite an intuitive one and some examples are explained below. The inference rule is given on the left and its corresponding sequent transition rule is given on the right.

The example below shows the most common case: where formulae in the set of hypotheses are matched with those at the top of the inference rule, and a single formula is added to the set as a result of applying the rule. These rules are read top-down and do not involve the proof changing scope.

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi}\ \wedge\mathrm{I} \tag{3.1}$$

$$\frac{\Gamma,\ \varphi,\ \psi \vdash \chi}{\Gamma,\ \varphi,\ \psi,\ \varphi \wedge \psi \vdash \chi}\ \wedge\mathrm{I} \tag{3.2}$$

The following example shows a case where an inference rule causes the proof to change its scope. Upon entering the new scope, a different goal may be pursued and new local assumptions and free variables may be introduced. These rules are read bottom-up, with both the top and bottom half referring to different scopes within the proof.

$$\frac{\begin{array}{c}[\varphi]\\ \vdots\\ \psi\end{array}}{\varphi \to \psi}\ \to\mathrm{I} \tag{3.3}$$

$$\frac{\Gamma,\ \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ \to\mathrm{I} \tag{3.4}$$

The example below is similar to the last but involves the proof branching into two distinct sub-proofs, thus it has two outputs.

$$\frac{\varphi \vee \psi \qquad \begin{array}{c}[\varphi]\\ \vdots\\ \chi\end{array} \qquad \begin{array}{c}[\psi]\\ \vdots\\ \chi\end{array}}{\chi}\ \vee\mathrm{E} \tag{3.5}$$

$$\frac{\Gamma,\ \varphi \vee \psi,\ \varphi \vdash \chi \quad \Gamma,\ \varphi \vee \psi,\ \psi \vdash \chi}{\Gamma,\ \varphi \vee \psi \vdash \chi}\ \vee\mathrm{E} \tag{3.6}$$

This final example illustrates a case where an inference rule results in a new free variable being brought into the new scope.

$$\frac{\exists x.\varphi \qquad \begin{array}{c}[\varphi[a/x]]\\ \vdots\\ \psi\end{array}}{\psi}\ \exists\mathrm{E} \tag{3.7}$$

$$\frac{a,\ \Gamma,\ \exists x.\varphi,\ \varphi[a/x] \vdash \psi}{\Gamma,\ \exists x.\varphi \vdash \psi}\ \vee\mathrm{E} \tag{3.8}$$

Inference rules for natural deduction are typically split into introduction and elimination rules. However, this split does not highlight the differences in the computation they require. We can better categorise inference rules based on whether they involve the computation branching (**branching** inference rules) or proceed linearly without such branching (**linear** inference rules). State transition rules are read in the same direction as their corresponding inference rules. If a rule is top-down, then the input state required to use the function can be found at the top of the transition function, and its output is found at the bottom. For top-down rules, the reading is switched, the input is at the bottom, and the output is at the top. We can identify whether a rule is linear or branching by examining how many states it outputs. For example, the disjunction elimination function has two output states at its top, so it is a branching rule. However, the implication introduction rule only has a single output state, so it is linear. It does not matter that $\to I$ and $\vee E$ are both bottom-up rules involving subproofs; the distinction between linear and branching rules is only in their output.

3.3 Natural deduction process

The process of proving a sequent through consecutive applications of inference rules is an inherently recursive one. When a sub-proof has been proven, the proof backtracks out of it, discharging any assumptions it used to do so. The proof may backtrack to a branch that it is yet to prove, or to its original state if there are no such branches. Once the proof returns to its original state, we can say that it has been completed. Up until now, we have not defined any means to complete a proof, or in the context of recursion, a base case. We can use the identity rule from sequent calculus for this "*which completes a proof when an antecedent matches a succedent*"[11]:

$$\frac{}{\Gamma, \varphi \vdash \varphi}\ \text{id} \tag{3.9}$$

Note that we can also conclude a proof if $\bot \in \Gamma$, as this would mean the proof has been reduced to absurdity, thus the *"thesis must be accepted because its rejection would be untenable.[12]"* Although this may be the case, we choose not to accept this as a concluding case as the proper way to end an absurd proof would be to use the $\bot E$ rule to derive the conclusion.

By using the identity inference rule as a predicate function, we can define an algorithm to better understand how this recursive proof process might be computed:

Algorithm 1 Build proof tree

```
function buildProofTree(sequent)
    rule ← getRule()
    if rule is linear then
        s ← applyRule(rule, sequent)
        if id(s) then
            return True
        else
            buildProofTree(sequent)
        end if
    else if rule is branching then
        (s1, s2) ← applyRule(rule, sequent)
        if id(s1) and id(s2) then
            return True
        else
            buildProofTree(sequent)
        end if
    end if
end function
```

Here, the *applyRule* function applies the given rule to the given sequent and produces either a single sequent if the rule is of a linear type, or a pair of sequents if it is a branching rule.

The function *buildProofTree* has a relationship of mutual recursion with the function *id*, which is the entry point into the recursion. *id* ends a proof if the proof state is complete, and calls *buildProofTree* to progress the proof if not.

Algorithm 2 Identity

```
function id(Γ ⊢ c)
    if c ∈ Γ then
        return True
    else
        buildProofTree(Γ ⊢ c)
    end if
end function
```

3.4 Proof trees

There are many different ways of portraying natural deduction proofs, and recursion trees constructed through the recursive proof process are yet another one of these. These recursion trees show every state that a proof goes through and the paths that the computation takes, which may make them useful in finding errors in proofs and even make it possible to automate the error detection process. The recursion tree for the proof $(\neg p \vee q \vdash p \to q)$ is shown below. Look to Figures 2.3 and 2.2 for the same proof in Fitch and tree notations.

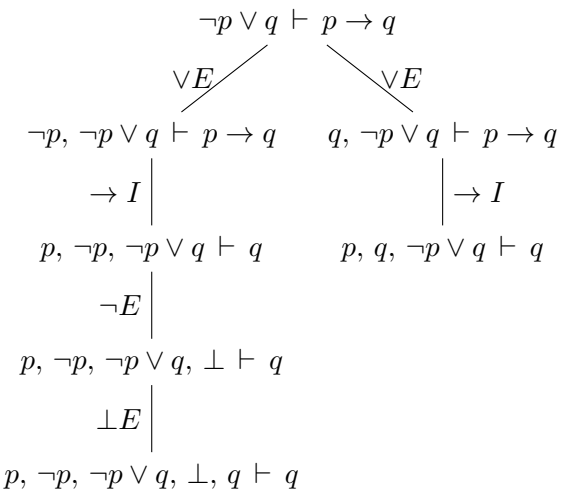


Figure 3.1: Recursive proof tree notation of $\neg p \vee q \vdash p \to q$

Chapter 4

Implementation

This chapter presents a comprehensive guide to implementing a natural deduction proof assistant, building upon the concepts introduced in the preceding chapter. While implementations for both propositional and predicate logic systems have been written, our discussion will primarily center on predicate logic. This decision stems from the fact that predicate logic extends propositional logic, and our system has been developed accordingly.

4.1 Data type definitions

In this implementation, all of the main concepts of the system (formulae, terms, sequents, and rules) are represented using data types, which may also have functions and instances of certain type classes defined over them. This section will discuss the implementations behind these data types in depth.

Haskell data types define data constructors separated by the '|' symbol [3]. These constructors establish a name for the value being constructed and specify zero or more other types that will be used to build it. We can create recursive data types by defining constructors that use the data type itself in their construction. A striking resemblance between Haskell data types and BNF syntax definitions can now be seen, making it possible to define data types almost exactly as they appear in their BNF definitions.

Type classes can be thought of in a similar way to interfaces in OOP (object-oriented programming). They require types to implement certain functions, allowing them to take on new properties and expand their functionality. For example, by defining the == operator, the *Eq* type class allows values of a data type to be compared.

4.1.1 Terms

The *Term* data type defines data constructors for variables and functions as they appear in their BNF syntax definition (2.3). In addition to these constructors, a constructor is also

defined for constants. In the BNF definition, constants are defined implicitly as functions with an arity of zero. However, the decision to define them explicitly was made to better the user experience by increasing the ease of use. An example can be seen when a user enters a constant and can enter a simple lowercase string rather than needing to follow it with a pair of empty brackets.

The lecture notes [13] define the set of logical variables for the language of predicate logic $\mathcal{V}$ as countably infinite. If characters were used to represent variables, we would fall short of this definition as the set of characters available is finite. Thus, we use strings to represent variables, and for the sake of consistency, this is also why the names of constants, functions, and relations are also strings.

```
data Term = Var    String
          | ConstT String
          | Func   String [Term]
```

The *Term* data type also defines instances for the *Show* and *Eq* type classes.

There are many points in the program where the use of a variable is specifically called for rather than a term. Thus, a type synonym ($type\ Variable = Term$) is defined to signify this.

A couple of functions are then defined over this data type for extracting the set of variables found in a term ($varsT$) and for substituting a term into a specific variable in another term ($subT$). These functions correspond to their definitions given in [lics] and are even written in much the same way. They have been suffixed with a 'T' to avoid having to import them as qualified when used in conjunction with their counterparts over formulae.

```
varsT :: Term -> [Term]
varsT (Var v)      = [Var v]
varsT (ConstT _)   = []
varsT (Func _ ts) = nub $ concatMap varsT ts
```

Sets are implemented as lists and are *nubbed* to ensure they never contain duplicate elements.

```
subT :: Term -> Variable -> Term -> Term
subT t (Var x) (Var y) = if x == y then t else Var y
subT _ _ (ConstT c)    = ConstT c
subT t x (Func f ts)   = Func f (map (subT t x) ts)
subT _ _ t             = t
```

The data constructor *Var* is also used to represent free variables. To distinguish them from ordinary variables, strings of digits represent free variables, while strings of lowercase

characters represent ordinary variables. Two additional functions are defined to check if a given variable is free (*isFree*) and for generating a new free variable from a list of variables (*newFreeVar*).

```
isFree :: Variable -> Bool
isFree (Var x) = all isDigit x
isFree _ = False
```

New free variables are generated in increasing order from zero upwards. The list of variables input is all the currently *recognised* variables in the current scope of a proof, which is why the free variables need to be filtered out. What exactly recognised variables are will be discussed in a later section.

```
newFreeVar :: [Variable] -> Variable
newFreeVar vs = Var $ show var
  where var = length (filter isFree vs)
```

4.1.2 Predicate formulae

Like with terms, the data type for predicate logic formulae closely resembles its BNF syntax definition. One difference is that quantifiers are constructed from strings instead of the variable type synonym. This choice was made so we can be sure that quantifiers will never quantify over constants or functions, which is a risk that arises from using the *Term* data type to represent these variables. Computations using this string variable are also made easier as we can match this string with another variable's name string without worrying about types not matching up.

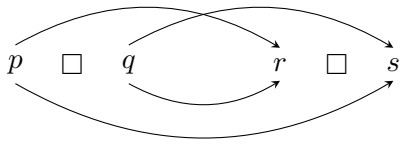
```
data Pred = Const  Bool
          | Eql    Term    Term
          | Rel    String  [Term]
          | Not    Pred
          | And    Pred    Pred
          | Or     Pred    Pred
          | Imp    Pred    Pred
          | Bicon  Pred    Pred
          | All    String  Pred
          | Exi    String  Pred
```

The type also defines a constructor for biconditionals ($\leftrightarrow$) instead of representing them implicitly through the conjunction of two opposing implications ($\varphi \rightarrow \psi \wedge \psi \rightarrow \varphi$) like in the BNF definition. Out of all the operators, $\leftrightarrow$ has the lowest level of precedence. This decision ensured that expectations were met when parsing formulae from user input. If a user

types out a formula containing biconditionals, they would expect to have it parsed into the exact formula they entered rather than one where the biconditionals have been transformed. These biconditionals can then be transformed into implications at the user's leisure with a couple of new inference rules:

$$\frac{\Gamma, \varphi \to \psi, \psi \to \varphi \vdash \chi}{\Gamma, \varphi \to \psi, \psi \to \varphi, \varphi \leftrightarrow \psi \vdash \chi} \leftrightarrow \mathrm{I} \quad (4.1) \qquad \frac{\Gamma, \varphi \leftrightarrow \psi \vdash \chi}{\Gamma, \varphi \leftrightarrow \psi, \varphi \to \psi \wedge \psi \to \varphi \vdash \chi} \leftrightarrow \mathrm{E} \quad (4.2)$$

Instances of *Show* and *Eq* are also defined for Predicate formulae. Some of the operators used in predicate formulae are commutative, and the *Eq* instance needs to accommodate this. Properly defining operators' commutative properties helps avoid many potential issues regarding the matching of formulae. It allows us to treat two formulae that we expect to be equivalent as such, even if they are written slightly differently. The diagram below aims to visualise how two formulae using the same binary operator have their operands compared for equivalence. Some of the binary operators (those with two operands) in the above definition are commutative, meaning that they express the same meaning regardless of how their operands are ordered.




```
instance Eq Pred where
  (==) :: Pred -> Pred -> Bool
  (Const x)      == (Const y)      =  x == y
  (u `Eql` v)    == (x `Eql` y)    = (u == x && v == y) || (u == y && v == x)
  (Rel x xs)     == (Rel y ys)     =  x == y && xs == ys
  (Not p)        == (Not q)        =  p == q
  (p `And` q)    == (r `And` s)    = (p == r && q == s) || (p == s && q == r)
  (p `Or` q)     == (r `Or` s)     = (p == r && q == s) || (p == s && q == r)
  (p `Imp` q)    == (r `Imp` s)    =  p == r && q == s
  (p `Bicon` q)  == (r `Bicon` s)  = (p == r && q == s) || (p == s && q == r)
  (All v p)      == (All u q)      =  v == u && p == q
  (Exi v p)      == (Exi u q)      =  v == u && p == q
  _              == _              =  False
```

Finally, a few recursive functions are defined over the *Pred* data type. Many of these extract different sets of terms from a formula: *terms*, *vars*, *freeVars*, and *boundVars*. A function *sub*

to substitute a term into a given variable in a formula is also defined. As all of these functions are implemented similarly and correspond to their definitions in [13], only the *sub* function has been given below to give an idea of how they are implemented.

```
sub :: Term -> Variable -> Pred -> Pred
sub _ _ (Const x)      = Const x
sub t v (x `Eql` y)    = subT t v x `Eql` subT t v x
sub t v (Rel x xs)     = Rel x (map (subT t v) xs)
sub t v (Not p)        = Not (sub t v p)
sub t v (p `And` q)    = sub t v p `And` sub t v q
sub t v (p `Or` q)     = sub t v p `Or` sub t v q
sub t v (p `Imp` q)    = sub t v p `Imp` sub t v q
sub t v (p `Bicon` q) = sub t v p `Bicon` sub t v q
sub t v (All x p)      = if v == Var x then All x p else All x (sub t v p)
sub t v (Exi x p)      = if v == Var x then Exi x p else Exi x (sub t v p)
```

4.1.3 Sequents

Sequents can be either a single-directional entailment ($\vdash$) or bi-directional ($\dashv\vdash$). Bi-directional entailment (called *equivalent* here) corresponds to the metalanguage sentence $\varphi \vdash \psi \ \ \& \ \ \psi \vdash \varphi$. However, it is easier to keep a type to parse equivalent sequents into and deal with the details of how they are computed later on. This also avoids having to explicitly introduce the notion of metalanguage to the user when it is better kept as a concern of the system. How exactly equivalent sequents are dealt with will be discussed later.

Equivalent sequents cannot have a set of hypotheses as this would make their backward entailment a single formula entailing a set of formulae. Sequents in the $\mathcal{NJ}$ natural deduction system that this system is concerned with can only entail a single formula; thus, equivalent sequents can only have a single hypothesis.

The list of variables a sequent holds are all the *recognised* variables in the current scope of a proof. Recognised variables are not only the free variables in the current scope but also all other variables that appear in formulae in the sequent and are available to use. Why sequents hold a set of recognised variables rather than exclusively holding free variables, as described in the design chapter, will become apparent when discussing rule applications.

```
data Sequent = ([Variable], [Pred]) `Entails` Pred
             | ([Variable], Pred) `Equivalent` Pred
```

The following function accepts a newly input sequent from a user and extracts all its variables into the set of recognised variables. As the proof progresses, variables are added and removed from the set as they go in and out of scope.

```
addVarsToSeq :: Sequent -> Sequent
addVarsToSeq ((_, as) `Entails` c) =
  (nub $ concatMap vars as ++ vars c, as) `Entails` c
addVarsToSeq ((_, a) `Equivalent` c) =
  (nub $ vars a ++ vars c, a) `Equivalent` c
```

As introduced in the design chapter, *identity* is a predicate function used to check whether or not a proof should be concluded.

```
identity :: Sequent -> Bool
identity ((_, as) `Entails` c) = c `elem` as
```

4.1.4 Rules

The rule data type is the "instruction set" of the system. It holds all the instructions available to execute during the main running of the proof and the operands (inputs) needed to execute them. Most of the instructions in the data type directly correspond to inference rules. However, it is is possible for there to be instructions, like *undo*, where this is not the case. The data stored for a given rule directly corresponds to the inputs required for the function that implements it. An entire listing of all of the rules available is provided below.

```
data Rule = Undo
          | AndIntro    Pred Pred
          | AndElimL    Pred                | AndElimR    Pred
          | ImpIntro                        | ImpElim     Pred Pred
          | OrIntroL    Pred                | OrIntroR    Pred
          | OrELim      Pred
          | NotIntro                        | NotELim     Pred Pred
          | BiconIntro  Pred Pred           | BiconElim   Pred
          | TopIntro                        | BottomElim  Pred
          | AllIntro                        | AllElim     Pred Term
          | ExiIntro    Pred Term Variable  | ExiElim     Pred
          | EqualIntro  Term                | EqualElim   Pred Pred
          | LemmaIntro  Pred                | Pbc
          deriving (Show)
```

4.2 Parsing input

4.2.1 Parser data type

A *parser* is a program that accepts a string as input and transforms it into an alternative representation that a computer program can understand. The data types introduced in the previous section are what user input will be parsed into. The parser introduced in this

section is an adaptation of the monadic parser first introduced by Graham Hutton [6]. Hutton designed this parser as a function that accepts a string as its input and attempts to consume part of the input, transforming it into a value of the parameterised type *a*. The value parsed is returned along with the rest of the string to be consumed by another parser.

The parser provided below wraps the output of the parser originally proposed by Hutton in an *Either String* [7]. This makes it possible to throw error messages from failed parses rather than having to indicate a failed parse through an empty list. Hutton defines his parser with a list as its output to keep it open for the possibility of multiple outputs resulting from a single application. A parser will only lead to multiple possible outputs if its grammar can be interpreted in multiple different ways, i.e. the grammar is ambiguous. Later in this section, a method for reducing grammar ambiguity will be discussed along with a parsing strategy to ensure that ambiguous grammars can be parsed in a definite way.

```
newtype Parser a = P (String -> Either String (a, String))

parse :: Parser a -> String -> Either String (a, String)
parse (P p) = p
```

The *item* parser is a primitive parser, outputting the first character of an input string and failing on empty inputs. It provides a way of consuming the input string that will be used by all other parsers that consume input.

```
item :: Parser Char
item = P $ \case
    []     -> Left "Empty input"
    (x:xs) -> Right (x, xs)
```

Monadic parsers make it possible to combine parsers in sequence to construct more complex ones. The first step in turning the parser type into a monadic parser is to instantiate an instance of Functor. A function fmap provides a way to construct a new parser by applying a function to the result of an existing one. Functors make it possible for functions to be applied to parsers. If the parser provided fails, the error is propagated.

```
instance Functor Parser where
  fmap :: (a -> b) -> Parser a -> Parser b
  fmap g p = P $ \input ->
    case parse p input of
      Left msg          -> Left msg
      Right (val, out) -> Right (g val, out)
```

Instantiating an instance of *Applicative* is the next step. Applicative requires the definition of two functions: *pure* and <*>. The *pure* function wraps a value in a parser that always succeeds, with the value as its output. On the other hand, the <*> function makes it possible to chain parsers together where the first parser in the chain is a function. The parsers that follow this function parser serve as its inputs and are each applied to it in turn. Any failure in the chain of applications is propagated through to the end.

```
instance Applicative Parser where
  pure :: a -> Parser a
  pure v = P $ \input -> Right (v, input)

  (<*>) :: Parser (a -> b) -> Parser a -> Parser b
  pg <*> px = P $ \input ->
    case parse pg input of
      Left msg       -> Left msg
      Right (g, out) -> parse (fmap g px) out
```

Finally, the *Parser* type can become monadic by defining the *bind* (>>=) operator. The bind operator allows parsers to be sequenced in a pipeline-like manner, where the output of one parser is input into the next one in the sequence. As with the <*> operator, a failure within any parser in the sequence will prevent any parsers down the line from being applied and the error to propagate to the end.

As the Parser type is now monadic, the *do* notation can sequence parsers and process their outputs in a manner that resembles an imperative programming style [10]. This style is often more intuitive than using a sequence of bind operations and will be used throughout the program where appropriate.

```
instance Monad Parser where
  (>>=) :: Parser a -> (a -> Parser b) -> Parser b
  p >>= f = P $ \input ->
    case parse p input of
      Left msg       -> Left msg
      Right (val, out) -> parse (f val) out
```

Alternatives provide another way of combining parsers. The *choice* operator (<|>) applies a parser to the input string and applies the alternative parser instead if it fails. These alternatives can be chained such that each parser is tried in turn, and the first one to succeed is applied to the input. The choice operator allows parsers to be written similarly to their CFG (context-free grammar) definitions. The second method defined is *empty*, the parser that always fails, regardless of input.

```
instance Alternative Parser where
  empty :: Parser a
  empty = P $ \_ -> Left "Failed parse"

  (<|>) :: Parser a -> Parser a -> Parser a
  p <|> q = P $ \input ->
    case parse p input of
      Left _           -> parse q input
      Right (val, out) -> Right (val, out)
```

Where more descriptive error messages are required, *fail* from the *MonadFail* typeclass can be used instead of *empty* to report errors during the parsing process.

```
instance MonadFail Parser where
  fail :: String -> Parser a
  fail msg = P $ \_ -> Left msg
```

Three primitive parsers have now been defined: *item* consumes a single character from a non-empty input string, *pure/return* is the parser that always succeeds, and *empty/fail* is the parser that always fails. The *return* function is implicitly defined in the *Monad* typeclass and is synonymous with pure. These parsers are the most basic building blocks and can be combined to build more complex ones. The functions that build new parsers by combining existing ones are called parser combinators. The first parser that these primitives are used to build is *sat*, the parser that consumes a single character only if it satisfies a given predicate.

```
sat :: (Char -> Bool) -> Parser Char
sat predicate = do
  x <- item
  if predicate x then
    return x
  else
    fail $ "Unexpected character: " ++ [x]
```

Using predicates from the *Data.Char* library, *sat* can be used to define parsers for different types of characters: *lower*, *upper*, *letter*, and *digit*. Consuming a specific, given character is achieved using the *char* parser, and by traversing it over an entire string, the *string* parser can consume an entire given string.

```
lower :: Parser Char
lower = sat isLower
```

```
char :: Char -> Parser Char
char x = sat (== x)

string :: String -> Parser String
string = traverse char
```

When parsing user input strings, it must be possible to ignore the white space that may surround the subject of a parser. The *space* parser consumes a single, continuous block of spaces in a string and uses *void* from the *Functor* type class to discard them. Here, we see *many* for the first time, a function provided by the *Alternative* type class. It makes it possible to consecutively apply a parser to an input zero or more times, as many times as possible. *Alternative* also provides *some* which applies a parser as many times as possible, like with *many*, but must be applied at least once. Tokens are units of an input and the subjects of parsers. Like words in a sentence, they may be surrounded by spaces but do not always have to be: e.g., $\forall x\ (P(x))$ where every character is a token, but there is only a single space.

```
space :: Parser ()
space = void $ many (sat isSpace)

token :: Parser a -> Parser a
token p = do space
             v <- p
             space
             return v
```

Tokens for lowercase and capitalised strings have parsers defined for them and will be used for parsing in the names of functions, variables, and relations. Numbers and counting numbers also have defined parsers. The number parser shown here uses the *bind* operator to sequence operations rather than add syntactic sugar by using the *do* notation.

```
lowerStr :: Parser String
lowerStr = token $ some lower

capitalisedStr :: Parser String
capitalisedStr = token $ do
                   x <- upper
                   xs <- many lower
                   return (x:xs)

number :: Parser Int
number = token $ some digit >>= \num -> return $ read num
```

The *symbol* parser parses any given string as a token, and an example of its use can be seen in the oft-used *comma* parser.

```
symbol :: String -> Parser String
symbol = token . string

comma :: Parser String
comma = symbol ","
```

Finally, a pair of parsers are defined to parse comma-separated lists. These lists do not necessarily have to consist of tokens, as the comma tokens consume all white space within the list, giving the effect of parsing a list of tokens. The *list* parser applies the given parser once and then keeps parsing values preceded by a comma as many times as possible. We see the >> operator from the *Monad* typeclass used here, which sequences parsers like *bind* but discards the result of the first application. This operator is an excellent fit for such a situation, and its use helps convey the inner workings of the parser better. The *listN* parser recursively parses a list up to a given length.

```
list :: Parser a -> Parser [a]
list p = do x <- p
            xs <- many (comma >> p)
            return (x:xs)

listN :: Int -> Parser a -> Parser [a]
listN n p = do x <- p
               do comma
                  xs <- listN (n-1) p
                  return (x:xs)
                <|> if n == 1 then return [x] else empty
```

The *eval* function evaluates the result of an application of a parser on an input string. Failures can result from either a parser not being able to fully consume the input string, or a failure occurring within the parsing process. Suitable error messages are then relayed to the caller. On success, the value resulting from the parse is output.

```
eval :: Parser a -> String -> Either String a
eval p xs =
  case parse p xs of
    Right (v, []) -> Right v
    Right (_, out) -> Left $ "Syntax error at" ++
      show (length xs - length out)
    Left msg -> Left msg
```

4.2.2 Symbols

Before delving into the specific implementations of parsers, we must first observe a couple of issues arising from parsing formulae without any knowledge of the symbols used within them.

Without any prior knowledge, a parser cannot differentiate between a constant and a variable when presented with a lowercase string token. The token *cat* could just as easily be a variable as a constant, and we have no idea which the user intended it to be. As explained in the background chapter, predicate formulae have a signature that lists all the function, constant, and relation symbols that can appear within them. If a parser has access to this signature, it can recognise the different symbol tokens it sees and differentiate between them. Therefore, if it sees a lowercase string token that it does not recognise as a constant, it must be a variable.

Another much more pressing issue comes from parsing without knowing the arity of functions and relations. Without this knowledge, it would be possible to parse functions and relations using different numbers of terms at each occurrence, which is an obvious error. This problem can again be solved by passing a signature into the parser, but this time including arities for functions and relations. If we know that a function f should have an arity n, then we know that any list of terms following it with a length different from n is an error. This assumes that know all the symbols that will appear within formulae in a proof and their arities where applicable.

Symbols are represented by the following data type, and a set of them can be used to represent the signature of formulae:

```
data Symbol = Constant String
            | Function String Int
            | Relation String Int
            deriving (Eq)
```

This data type can be parsed into using the parser shown below. This parser is listed as a CFG grammar instead of using its Haskell implementation as its implementation is very similar to it.

```
S -> lowerStr | lowerStr(countingNumber) | capitalisedStr(countingNumber)
```

A signature (represented as a list of symbols) is parsed using a recursive parser combinator. It consumes a single symbol according to the above symbol parser and keeps consuming commas followed by symbols until the end of the input is reached, at which point the list of symbols is output. For formulae that do not use symbols, there is an alternative parser that consumes a "-" token and outputs an empty list. We cannot use an empty input for this as the empty input always results in a failed parse. The parser for signatures has the CFG: $T \rightarrow S(,\ S)^* \mid -$, and is written as follows:

```
symbolsP :: [Symbol] -> Parser [Symbol]
symbolsP sig = do sym <- symbolP sig
                  do comma
                     otherSyms <- symbolsP (sym : sig)
                     return (sym : otherSyms)
                   <|> return [sym]
                <|> do symbol "-"
                       return []
```

4.2.3 Grammars and parser combinators

Predicate formulae

The language of a parser is its grammar. If we took the BNF definition for predicate logic formulae (2.3) as the grammar for their parser, we would find that it completely lacks the concept of an order of precedence. The formula $p \wedge q \rightarrow r$ (where p, q, and r are formulae) could be parsed as either $(p \wedge q) \rightarrow r$ or $p \wedge (q \rightarrow r)$. To address this issue, Hutton advises splitting the grammar so that there is a single production rule for each level of precedence [Hut]. There are five levels of precedence among the operators of predicate logic: $\{\leftrightarrow\}$, $\{\rightarrow\}$, $\{\wedge, \vee\}$, $\{\neg, \forall, \exists\}$, $\{(p)\}$ where the highest level is a bracketed formula. Rather than listing the terminals in their own production rule, they have been integrated into the final parser level. The production rules are below (all symbols except L1P-L5P are terminals):

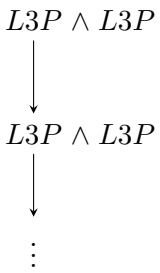
$$
\begin{aligned}
&L1P \rightarrow L1P \leftrightarrow L1P \mid L2P \\
&L2P \rightarrow L2P \rightarrow L2P \mid L3P \\
&L3P \rightarrow L3P \wedge L3P \mid L3P \vee L3P \mid L4P \\
&L4P \rightarrow \neg L4P \mid \forall Var\, L4P \mid \exists Var\, L4P \mid L5P \\
&L5P \rightarrow T \mid F \mid P(Terms) \mid (L1P) \mid Term = Term
\end{aligned}
$$

The production rules are ordered in reverse order of precedence (from lowest precedence to highest) due to the top-down nature of the recursive descent method being used. Recursive descent parsers read the production rules from top to bottom, so the operators with higher precedence are applied first in the parse tree as they are closer to the terminals. The left-to-right application of rules in LL (leftmost derivation) parsers like the one built here gives them a peculiar property when parsing formulae. They consider rules that appear further right in a production to be higher in precedence as they are considered later. This property gives disjunction higher precedence than conjunction in the above grammar, even though they technically have the same precedence.

Each production rule, except the last, has a path into the next level below to ensure the recursive descent parser can try other rules if it fails to find a match on a level. Once a parser has reached a certain level, it can either keep parsing at that level, or descend into

the next level if it no matches are found. Once a parser descends to a lower level, it cannot return to the higher higher levels. By preventing the parsing of rules at higher levels than the current one, we can preserve the order of precedence. The only way a parser can rise back to the top level of the production rules is if it encounters a formula surrounded by parentheses.

"No LL(k) grammar is left-recursive" [1]. If a left-to-right parser's grammar were left recursive, it would be infinitely recursive. Parsing the formula $p \wedge q \vee r$ using the above grammar would result in a match being found in the first rule of the L3P production rule (L3P $\wedge$ L3P). Expanding on the first non-terminal of this rule would again find a match in the same rule. This will continue infinitely as none of the symbols are ever reduced to terminals, so none of the tokens are ever consumed.



Hutton suggests adjusting the rule "to be recursive in their right argument only, rather than in both arguments" to eliminate left recursion [7]. This change to the grammar forces the formula to the left of a binary operator to go down a level in the order of precedence and closer to the terminals. This is only possible because of the parser's LL property. Therefore, we can be sure that the instance of a rule being parsed is its first occurrence in the unconsumed input, and there are no more rules at this precedence level to parse on the left side after it has been applied. It also forces all the binary operators to become right-associative. This follows the convention for some operators like $\rightarrow$ but against convention for others like $\wedge$ and $\vee$. Whatever convention the user uses, brackets can always be added around formulae to ensure they are parsed precisely as intended. The final form of the grammar now looks like this:

$$
\begin{aligned}
&L1P \rightarrow L2P \leftrightarrow L1P \mid L2P \\
&L2P \rightarrow L3P \rightarrow L2P \mid L3P \\
&L3P \rightarrow L4P \wedge L3P \mid L4P \vee L3P \mid L4P \\
&L4P \rightarrow \neg L4P \mid \forall Var\, L4P \mid \exists Var\, L4P \mid L5P \\
&L5P \rightarrow T \mid F \mid P(Terms) \mid (L1P) \mid Term = Term
\end{aligned}
$$

Recursive descent parsers are built up of mutually recursive functions, each implementing one of the grammar's production rules. The functions are implemented as parser combinators and

mutual recursion is achieved through the paths that rules hold to the parser at the level below them. The listing below shows the first level, in which biconditionals are parsed. The parser first parses the left side with the L2P parser then attempts to parse a bi-conditional symbol followed by a parse at the L1P level. If this is successful, the function returns a bi-conditional formula; otherwise, it returns the formula parsed at the level below. This *alternative* style of parsing is made possible through the *choice* operator. The *choice* operator can also make it possible to accept multiple different synonyms for a symbol.

```
l1P :: [Symbol] -> Parser Pred
l1P sig = l2P sig >>= \p ->
            do symbol "<->" <|> symbol "↔"
               q <- l1P sig
               return $ p `Bicon` q
          <|> return p
```

The L2P parser is the same as the above, except it parses implications instead. The L3P parser is also very similar, except it parses two different operators. As discussed earlier, the order in which the operators are listed makes a difference in how formulae are parsed. The further down an operator is listed, the higher its precedence is in a formula.

The L4P parser perfectly demonstrates the roles the *do* notation and *choice* operator play in a parser. The *do* notation encapsulates a sequence of operations as a parser, while the *choice* operator can switch between parsers based on the subject being parsed. Expressions like the case expression can be used to process and return values from a *do* notation sequence. Here, case expressions are used to fail a parser if the variable provided for a quantifier exists in the signature, verified using the *findSymbol* function. If the parsed name exists in the signature, then it must either be a constant or a function and not a variable. The inline *fmap* function (<$>) is used to complete the partial construction of quantifiers with the following formula parsed.

```
l4P :: [Symbol] -> Parser Pred
l4P sig = do symbol "NOT" <|> symbol "¬"
             Not <$> l4P sig
          <|> do symbol "ALL" <|> symbol "∀"
                 v <- lowerStr
                 case findSymbol v sig of
                      Nothing -> All v <$> l4P sig
                      _       -> empty
          <|> do symbol "EXISTS" <|> symbol "∃"
                 v <- lowerStr
                 case findSymbol v sig of
                      Nothing -> Exi v <$> l4P sig
                      _       -> empty
          <|> l5P sig
```

The final level in this recursive descent parser parses the terminals and bracketed formulae. The parsers have been arranged in such an order to minimise the number of false positives that occur during a parse and consequently, the number of times the parser has to backtrack. For example, if the bracketed formula parser was before the relation one, the list of terms inside the brackets of the relation would be parsed using the L1P parser. Then, only after the parser inevitably fails, will it return to the relation parser to be parsed correctly. By ordering parsers from most to least specific, these unnecessary errors can be avoided.

```
l5P :: [Symbol] -> Parser Pred
l5P sig = do symbol "T" <|> symbol "TRUE"
             return $ Const True
      <|> do symbol "F" <|> symbol "FALSE"
             return $ Const False
      <|> do r <- capitalisedStr
             case findSymbol r sig of
                  Just (Relation _ arity) -> do
                       symbol "("
                       ts <- listN arity $ termP sig
                       symbol ")"
                       return $ Rel r ts
                  _                       -> empty
      <|> do symbol "("
             p <- l1P sig
             symbol ")"
             return p
      <|> do l <- termP sig
             symbol "="
             r <- termP sig
             return $ l `Eql` r
```

Terms

As with the parsers listed earlier, the term parser lists its parsers from most to least general. A case expression is used to decide whether a lowercase token should be parsed as a variable or a constant based on whether or not it appears in the signature. Number tokens are parsed as free variables.

```
termP :: [Symbol] -> Parser Term
termP sig = do f <- lowerStr
               case findSymbol f sig of
                    Just (Function _ arity) ->
                       do symbol "("
                          ts <- listN arity $ termP sig
                          symbol ")"
                          return $ Func f ts
                    _                       -> empty
           <|> do x <- lowerStr
               case findSymbol x sig of
                    Nothing           -> return $ Var x
                    Just (Constant _) -> return $ ConstT x
                    _                 -> empty
           <|> do Var . show <$> number
```

Sequents

The sequent parser first parses a single-directional entailment with a list of hypotheses. Failing that, it parses an equivalent sequent. Finally, it parses a single-directional entailment with no assumptions.

```
sequentP :: Signature -> Parser Sequent
sequentP syms = do ls <- list $ predP syms
                   symbol "|-" <|> symbol "⊢"
                   r <- predP syms
                   return (([], ls) `Entails` r)
                <|> do l <- predP syms
                       symbol "-||-" <|> symbol "⊣⊢"
                       r <- predP syms
                       return (([], l) `Equivalent` r)
                <|> do symbol "|-" <|> symbol "⊢"
                       r <- predP syms
                       return (([], []) `Entails` r)
```

4.3 Rule application

The *main* function in the *Main* module is the entry point into the program. It prompts the user for a signature, followed by a prompt for the sequent to be proven that uses the signature provided. If the sequent input was a single-directional entailment, *applyRule* would be called to begin the natural deduction process. However, if an equivalent sequent was input, *applyRule* is first called to prove the entailment going forward (from the hypothesis to the

conclusion) before being called again to prove the entailment going in the opposite direction. Being of the *IO* type, user input *IO* operations can be sequenced and processed in *main*.

```
main :: IO ()
main = do sig <- getSymbols
          s <- getSequent sig
          case s of
            (_ `Entails` _) -> do res <- applyRule sig s
                                  putStrLn "Proof complete"
                                  return ()
            ((vs, a) `Equivalent` c) -> do
              res1 <- applyRule sig ((vs, [a]) `Entails` c)
              putStrLn "Forward sequent proven\n"
              res2 <- applyRule sig ((vs, [c]) `Entails` a)
              putStrLn "Reverse sequent proven"
              putStrLn "Proof complete"
              return ()
```

The design chapter discussed how inference rules can be transformed into transition functions for sequent states. Two types of transition functions were identified: those that only return a single output (linear functions) and those that return two outputs (branching functions). The *RuleApplication* data type encapsulates all the possible outcomes from applying an inference transition function to a sequent state. The *LinearApplication* and *BranchingApplication* values are the outputs from linear and branching rules and hold the resulting sequent or sequents. They both represent the result of successfully applying a rule to a sequent. The *InvalidApplication* value is used when an application is unsuccessful and holds the error message output from the failed application. Finally, the *UndoingApplication* will cause a proof state to revert to the immediately preceding one. While not technically the result of a rule application, the *UndoingApplication* is what results from the *Undo* rule from the *Rule* data type being called upon to change a proof state. Thus, it is more accurate to say that the *RuleApplication* data type represents the change a proof state might undergo after being affected by a user's application of a rule.

```
data RuleApplication = UndoingApplication
                     | InvalidApplication    String
                     | LinearApplication     Sequent
                     | BranchingApplication Sequent Sequent
```

The natural deduction process is split over three functions, of which the first two are mutually recursive. They form a hierarchy where a state enters the first level and is advanced by a rule in the third. The *applyRule* function is the first level and is responsible for deciding whether to advance an incomplete proof state or to conclude a complete one. It uses the identity function to make this decision.

```
applyRule :: Signature -> Sequent -> IO Bool
applyRule sig s = do
    if identity s then
        return True
    else applyRule' sig s
```

The next level in the hierarchy (*applyRule'*) is where the core of the recursion takes place. It begins by accepting a rule from a user and applying it to the current sequent state. Notice how the result of this application can be stored using *let*, akin to how variables are used in imperative programs. Also, notice how, unlike any other sequenced operation, the expression used to obtain the result in *let* does not have to be in the same monad as the return type; it is of a *RuleApplication* type here instead of an *IO* type. Once the result of the rule application has been found, a case expression is used to make a decision based on its nature.

If the result of a proof application is invalid, the resulting error message is displayed, and the recursion self-loops, keeping the proof state unchanged. However, if the application is successful and results in a single sequent, the resulting proof state is sent to the first level to either continue the recursion and advance the state or conclude it. The remainder of this case will only continue once the current path being followed comes to an end and *applyRule* returns a boolean value. An application resulting in a pair of sequents first aims to complete the recursion using the first proof state and then moves on to the next once the first resolves to a boolean value.

So far, the only cases provided would advance a proof forward, even if a user no longer wishes to take a proof down the current path. By returning *False* from a state in the proof, the *UndoingApplication* can stop a proof in its tracks and return it to its previous state. The conditionals in the linear and branching cases make this possible, allowing a state to follow a new path if an application returns *False*. As Haskell evaluates expressions lazily, if the first branch of a *BranchingApplication* was undone and returned *False*, the following branch would not be evaluated. The conditional will immediately short-circuit to its first branch, and the recursion will self-loop, giving the state returned to the chance to follow a new path.

```
applyRule' :: Signature -> Sequent -> IO Bool
applyRule' sig s = do
  print s
  r <- getRule sig
  let ruleApl = applyRule'' s r
  case ruleApl of
    InvalidApplication str -> do
      putStrLn $ "Error:" ++ str
      applyRule' sig s
    LinearApplication s1 -> do
       result <- applyRule sig s1
       if not result then
          applyRule' sig s
       else return True
    BranchingApplication s1 s2 -> do
       result1 <- applyRule sig s1
       result2 <- applyRule sig s2
       if not (result1 && result2) then
          applyRule' sig s
       else return True
    UndoingApplication -> return False
```

The final level in the hierarchy (*applyRule"*) maps rules to their corresponding implementations and returns the result of their application to the given state. The *Undo* rule maps straight to the *UndoingApplication* value.

```
applyRule'' :: Sequent -> Rule -> RuleApplication
applyRule'' s rule = case rule of
       (AndIntro p q)   -> andI   s p q
       (AndElimL p)     -> andEl  s p
       (AndElimR p)     -> andEr  s p
        ImpIntro         -> impI   s
       (ImpElim p q)    -> impE   s p q
                          ...
       AllIntro         -> allI   s
       (AllElim p t)    -> allE   s p t
       (ExiIntro p t v) -> exiI   s p t v
       (ExiElim p)      -> exiE   s p
       Pbc              -> pbc    s
       Undo             -> UndoingApplication
```

Below are a few selected listings of the implemented rules and their explanations. These rules comb through the provided inputs, attempting to catch any errors that may prevent them

from being applied to their provided sequents. If no such errors are found, they return the resulting sequent (or sequents in the case of *BranchingApplication*s).

The conjunction elimination rule is one of the more simple rules. It accepts a conjunctive formula from the set of hypotheses and adds either its left or right formula to the set with *setApp*, which appends an element to the end of a set. This rule may fail in two cases: if the given formula does not appear in the set of hypotheses or if the formula provided is not conjunctive. A conjunctive formula is not just a formula involving a conjunction; it is a formula formed of two others combined by a conjunction. A conjunction must be the root operator of the formula in its parse tree. For example, the formula $p \vee (q \wedge r)$ is not conjunctive, while the formula $(p \vee q) \wedge r$ is. Instead of rewriting the left rule, the right rule is written using the left rule and switches the order of the operands in the formula supplied.

```
andEl :: Sequent -> Pred -> RuleApplication
andEl ((vs, as) `Entails` c) (p `And` q)
  | (p `And` q) `elem` as =
        LinearApplication ((vs, setApp p as) `Entails` c)
  | otherwise = InvalidApplication "Proposition not in scope"
andEl (_ `Entails` _) _ =
    InvalidApplication "Conjunctive proposition must be provided"

andEr :: Sequent -> Pred -> RuleApplication
andEr s (p `And` q) = andEl s (q `And` p)
```

Unlike the conjunction elimination rules, the disjunction introduction rules do not follow their inference rules; they accept different inputs to their inference rules. Their inference rule is listed below. It accepts a single formula and adds it onto the left side (for the left version) of a disjunctive formula, which already has another formula present on its right side.

$$\frac{\varphi}{\varphi \vee \psi} \vee \mathrm{I}_{\mathrm{l}} \tag{4.3}$$

This inference rule poses a serious issue as the ψ formula is seemingly pulled out of thin air. To overcome this, the function instead accepts the end formula as its input. The formula provided is then checked to see if it is indeed disjunctive, and its left formula (φ in the rule shown above) is checked to verify its existence in the set of hypotheses. Once the function is sure the provided formula has no problems with the current sequent, it is added to the set of hypotheses and this new sequent is output. The ψ formula is a newly introduced formula that has not been derived from any formula already within the set of hypotheses. Consequently, it may introduce some new variables previously unseen in the proof and these new variables are introduced into the scope by adding them to the set of recognised variables.

```
orIl :: Sequent -> Pred -> RuleApplication
orIl ((vs, as) `Entails` c) (p `Or` q)
  | p `elem` as = LinearApplication
    (mergeSets vs (vars q), setApp (p `Or` q) as) `Entails` c
  | otherwise = InvalidApplication "Proposition not in scope"
orIl (_ `Entails` _) _ =
    InvalidApplication "Disjunctive proposition must be provided"

orIr :: Sequent -> Pred -> RuleApplication
orIr s (p `Or` q) = orIl s (q `Or` p)
```

The lemma introduction function is a branching rule. Its first branch changes the goal of the sequent to the given formula. Once this forumula has been proven, the proof follows the next branch, which adds this formula to the set of hypotheses and changes the goal of the proof back to the original consequent. As a new formula is being used in the proof, its variables are added to the set of recognised variables.

```
lemmaI :: Sequent -> Pred -> RuleApplication
lemmaI ((vs, as) `Entails` c) p = BranchingApplication
    ((mergeSets vs (vars p), as) `Entails` p)
    ((mergeSets vs (vars p), setApp p as) `Entails` c)
```

The inference rule for existential introduction, also known as existential generalisation, appears deceivingly simple from its inference rule:

$$\frac{[\varphi[t/x]]}{\exists x.\varphi}\ \exists\mathrm{I} \tag{4.4}$$

In it, we can identify three separate inputs: the formula φ to be generalised and the term t to replace with the bound variable x. These inputs also need to be tested to ensure there are no conflicts within and between them. Guards are used to catch these conflicts and they are computed in a where clause to improve their readability.

```
exiI :: Sequent -> Pred -> Term -> Variable -> RuleApplication
exiI ((vs, as) `Entails` c) p t (Var v)
  | termNotInPred = InvalidApplication
        "Provided term does not appear in the formula provided"
  | varBound = InvalidApplication
        "Provided variable is already bound"
  | varCapture = InvalidApplication
        "Substituting in provided term will result in variable capture"
  | otherwise = LinearApplication
        ((mergeSets vs $ setApp (Var v) (varsT t), setApp generalisedP as)
            `Entails` c)
  where termNotInPred = t `notElem` terms p
        varBound = Var v `elem` boundVars p
        varCapture = not . null $ varsT t `intersect` boundVars p
        subbedP = sub (Var v) t p
        generalisedP = Exi v subbedP
```

In existential elimination, we see how new free variables are added to a scope. It is also the first function shown here that introduces a local assumption into the scope and shows how it is prepended to the front of the set of hypotheses with *setPre*. The set of hypotheses is a set in that it contains no duplicates. However, as it is implemented using a list, there is an order to the hypotheses. While this order is unimportant for computations on the set and the list truly acts as a set for these computations, the order can be used to give importance to specific hypotheses. Local assumptions are always added to the front of the set to distinguish them from the other hypotheses so the user knows which hypothesis is the local one in a subproof. This is similar to how local hypotheses are listed at the top of a subproof box in Fitch-style notations.

```
exiE :: Sequent -> Pred -> RuleApplication
exiE ((vs, as) `Entails` c) (Exi v p)
  | Exi v p `elem` as = LinearApplication
        ((setApp freeVar vs, setPre subbedP as) `Entails` c)
  | otherwise = InvalidApplication "Propostion not in scope"
  where freeVar = newFreeVar vs
        subbedP = sub freeVar (Var v) p
exiE (_ `Entails` _) _ = InvalidApplication
    "Proposition must be existentially quantified"
```

Chapter 5

Evaluation

In this chapter, we will analyse a couple of executions of the program on different proofs, aiming to highlight different interesting aspects of the program. Following this, we will discuss how the system was tested and evaluate its success. Finally, we will identify different problem areas in the system and provide ways to improve it.

5.1 The proof assistant in action

This section analyses two proof problems from the LICS worksheets.

Figure 5.1 shows an execution for an equivalent sequent in predicate logic. For first-order logic proofs, sequents additionally print their set of recognised variables. In this way, users always know the variables recognised by the current state, so they know what variables are available for use. We can also clearly see the two separate proofs required to solve the equivalent sequent and how the program immediately switches between them. Listed directly below are the two proofs in Fitch notation. If we read these proofs according to the directions the inference rules go in (and the same order that the proof would have been initially written in), we find that this order matches the order in which the rules are applied in Figure 5.1.

1.	$\forall x.P(x)$	hyp
2.	$\exists x.\neg P(x)$	hyp
3.	$a \quad \neg P(a)$	hyp
4.	$P(a)$	$\forall$ E, 1
5.	$\bot$	$\neg$ E, 3,4
6.	$\bot$	$\exists$ E, 2,3-5
7.	$\neg\exists x.\neg P(x)$	$\neg$ I, 2-6

1.	$\neg\exists x.\neg P(x)$	hyp
2.	a	
3.	$\neg P(a)$	hyp
4.	$\exists x.\neg P(x)$	$\exists I$, 3
5.	$\bot$	$\neg E$, 1,4
6.	$P(a)$	PBC, 3-5
7.	$\forall x.P(x)$	$\forall I$, 2-6

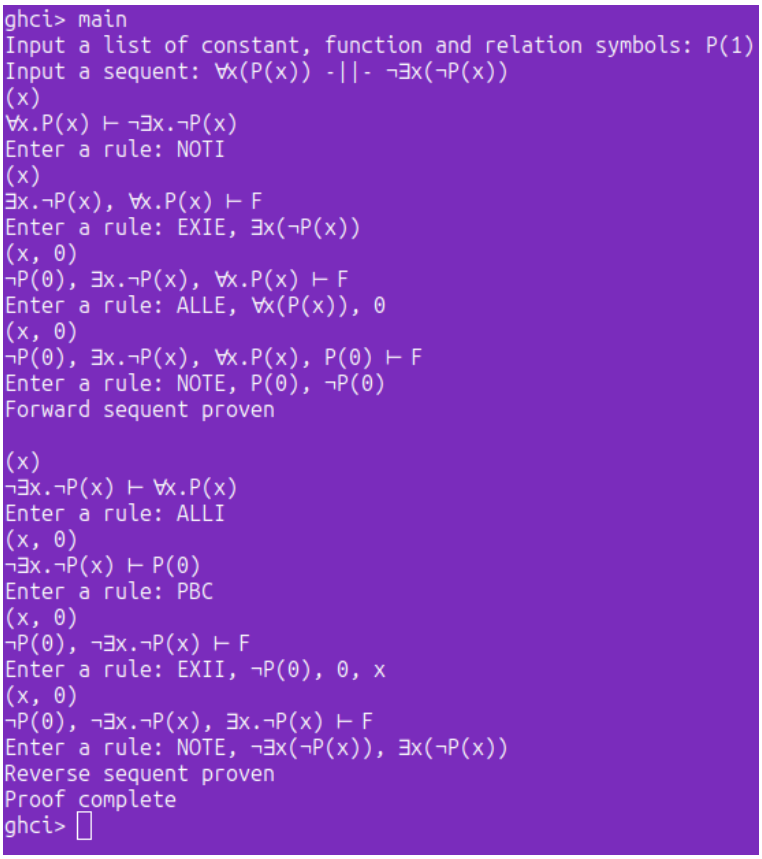


Figure 5.1: Question 4.3(b): $\forall x.P(x) \dashv\vdash \neg\exists x \neg P(x)$

Figure 5.2 below displays a more complex propositional logic proof than others in the worksheets. The complexity comes from the proof becoming 'stuck' when the state $\neg\alpha$, $((\alpha \to \beta) \to \alpha) \vdash \bot$ is reached. To get the proof back on track, the user must realise that they can derive $\bot$ by using negation elimination with $\neg\alpha$ from the set of hypotheses and an α. The problem is that the only way α can be brought into the scope is by deriving it from the hypothesis $((\alpha \to \beta) \to \alpha)$ with an implication elimination and the formula $\alpha \to \beta$, which is currently not in scope either. This is where the lemma introduction rule comes in. It allows a proof to change its goal to any formula the user desires, and by proving that goal, allows the formula to enter into the set of hypotheses. The figure below shows how the lemma introduction rule brings the formula $\alpha \to \beta$ into scope.

Figure 5.2 also shows what happens when syntax errors occur in user input and when the system rejects the application of rules. In both cases, an error message is displayed, and the proof continues as usual, re-prompting the user for whatever it requires. Immediately after the application of bottom elimination, we can see the local hypothesis α being eradicated from the set of hypotheses as the subproof is exited. The *UNDO* rule is also shown here, undoing an application of implication introduction and reverting the proof state.

```
ghci> main
Input a sequent: |- ((a → β) → a) → a
 ⊢ (((a → β) → a) → a)
Enter a rule: IMPI
((a → β) → a) ⊢ a
Enter a rule: UNDO
 ⊢ (((a → β) → a) → a)
Enter a rule: IMPI
((a → β) → a) ⊢ a
Enter a rule: PBC
¬a, ((a → β) → a) ⊢ F
Enter a rule: LEMMAI, a → β
¬a, ((a → β) → a) ⊢ (a → β)
Enter a rule: IMPA
Unexpected character: I
Enter a rule: ANDEL, ¬a
Error: Conjunctive proposition must be provided
¬a, ((a → β) → a) ⊢ (a → β)
Enter a rule: ANDEL, β
Error: Conjunctive proposition must be provided
¬a, ((a → β) → a) ⊢ (a → β)
Enter a rule: IMPI
a, ¬a, ((a → β) → a) ⊢ β
Enter a rule: NOTE, a, ¬a
a, ¬a, ((a → β) → a), F ⊢ β
Enter a rule: BOTE, β
¬a, ((a → β) → a), (a → β) ⊢ F
Enter a rule: IMPE, (a → β) → a, a → β
¬a, ((a → β) → a), (a → β), a ⊢ F
Enter a rule: NOTE, a, ¬a
Proof complete
ghci>
```

Figure 5.2: Question 3.1: $\vdash ((\alpha \to \beta) \to \alpha) \to \beta$

5.2 System testing

Though only two executions are shown, they give a good idea of how the system works. In addition to these problems, the system was tested on many more from the worksheets. In total, the system was tested on fifteen proof problems from the LICS worksheets, which included problems in both propositional and predicate logic. The system was first tested using the exact sequence of rule applications in the worksheet solutions to test for correctness. It was then tested against alternative solutions (a different sequence of rule applications) where possible and with erroneous inputs to test for flexibility. Many inconspicuous features were deemed necessary additions through this testing, either because a proof would be unsolvable without it, like with the lemma introduction rule, or to prevent errors and unexpected behaviour, like with the commutativity of operations.

In addition to the worksheets, the system was tested against a select few problems from the vast collection of problems in the Oxford Natural Deduction Pack [2]. This problem set presented the system with much more challenging questions. It allowed it to test how it handles problems involving constant terms and the equality and biconditional operators,

as they were absent in the worksheets. Around ten problems were tested from this problem set, similar to how they were tested with the LICS worksheets. The problems selected were the longer and more difficult ones from further down in the worksheet to push the system. All but one of the problems tested were successfully solved using the proof assistant. This problem exposed a flaw in the system, which will be discussed in the following section.

All testing was conducted manually due to the interactive nature of the proof assistant. Even then, it was possible to thoroughly test every rule and failure condition within the assistant.

5.3 System flaws

Now that a thorough understanding of the system has been reached, we can begin to discuss the system's flaws. The first can be spotted in Figure 5.2 above. When a syntax error is made, the program outputs quite an unhelpful error message, not describing where or why the parse failed. This is due to the use of alternatives, which prevent descriptive error messages from being thrown from where they occur, as although the parser may fail in one section of an alternative, it ignores the error and carries on to the next. It remains hopeful that it will find a match in another alternative parser, even if this may not be the case. A parser library like Parsec can be used to solve this, which has much more powerful error handling and propagation built into the applicative machinery [9]. This ensures that errors arising from parsers in applicatives are propagated correctly rather than ignored. That said, parser libraries were not used to minimise external dependencies and keep the system as flexible and easily extensible as possible.

Currently, no checks are in place to prevent users from entering mixed quantifiers that quantify over the same variable. These checks encourage users to write formulae whose meanings are not prone to misunderstanding. One way to do this is to include extra logic in the quantifier parsers to output a notice whenever such a formula is input. Another way would be to *rename* the formula as described in [13], but this would mean that the parser can output formulae that defy the user's expectations. A good compromise would be to give the user a notice and then provide the option to rename the formula.

One final issue related to parsing is that users can use variables that are not in the recognised set. Thanks to the comprehensive error checking built into the rules, this will never seriously threaten the program. Even so, the set of recognised variables exists to prevent users from making meaningless steps in a proof. It would be possible to prevent unrecognised variables from being parsed by passing the set of recognised variables into the term parser. However, this would burden all the other parsers that use the term parser with another input besides the signature. This extra baggage that the parsers would have to throw around was decidedly not worth it when the only downside is that the hypothesis set becomes cluttered with unnecessary formulae. These steps can also be undone to remove the newly introduced formulae involving

foreign variables.

In the Existential 22 problem from [2], the solution provided generalises only one of the free variables in the predicate $P(a, a)$ when $\exists$I is applied to it, resulting in the formula $\exists y.P(a, y)$. In the current implementation, the predicate would be generalised to the formula $\exists y.P(y, y)$ as the substitution function it uses does not have such flexibility. In this context, the substitution function replaces and generalises all the free variable a occurrences with the bound variable y, and we have no say in which particular free variables are affected by the substitution. This means that some problems in the [2] pack are either unsolvable or not solvable in the same way as those presented in the solutions when using the system.

Chapter 6

Conclusions

6.1 Project contributions

- Introduced a new way of viewing seqents as data structures for storing the state in a natural deduction proof and inference rules as transition functions over sequent states to advance a proof state.
- Introduced an abstract and versatile framework for natural deduction in Haskell, capable of adapting to different proof systems.
- Designed a recursive algorithm for the natural deduction proof process.
- Designed a powerful and versatile recursive descent parser utilising the monad structure.

6.2 Reflection

In hindsight, using a functional programming language like Haskell was ideal for this project due to its higher-order functions and algebraic data structures. The BNF definitions of syntaxes and recursive functions given in the lecture notes were intuitive to implement because of how similar Haskell code can be written to mathematical notations. The monad structure enabled a powerful general-purpose parser to be implemented in a few lines of code and made the parsing process incredibly simple and intuitive. A monadic parser like this would require hundreds of lines of code and many classes in other langauges, and the parsers themselves could not be written nearly as succinctly as they are in Haskell.

Encapsulating the proof state in a sequent and using recursion to move between them was a clever idea. However, the algorithm implemented carries on indefinitely even if the proof is unsolvable, as it uses the identity inference rule as a stopping case. This is not an issue as it is quite like solving a proof in real life: once one has had enough, they can quit and move on. That said, another helpful feature may have been to save the current state of a derivation so that once they feel like trying again, they can restart from where they left off. The inspiration

for the *undo* rule also came from observing how natural deductions are performed by hand; lines in a proof can be erased, taking the proof back a step.

The only change that comes to mind for a change that would be made if the project were to be rewritten is to have a GUI made for it, no matter how basic. Natural deduction is already a difficult concept to grasp, and having to perform it in a terminal makes it much more so. Initially, a library was selected to implement the GUI. However, this was never done due to the steep learning curve for these complex operations in Haskell and their lack of proper documentation.

6.3 Future work

In addition to rectifying the system's flaws outlined earlier, various ways can be found to improve and extend the system to make it more useful and user-friendly.

A simple addition to the system would be to print the recursive proof tree on the completion of a proof, as shown in 3.1. This should be easy in theory, as the tree is created alongside the proof. However, it is tricky in practice, as formatting large and complex trees with long sequent states is difficult in a command-line interface.

If a graphical user interface was built for the program, recursive proof trees and other proof notations could be drawn graphically, making them much easier to display. It would also significantly improve the user experience as intuitive graphical interactions could replace the text-based interaction. An example would be the ability to revert a proof to any state just by clicking on it rather than manually using the undo rule multiple times to revert to the earlier state. Considering how long the list of inference rules is, it would also be incredibly beneficial to have a page listing all the transition functions, as these can be difficult to remember. Given the time, building a graphical interface for the program would have added the most value to the project and made it feasible to release to students. Automating the proof solving process would be most difficult, but also immensely valuable to users.

To add a new inference rule to the system, it must first be translated into a transition function. It can then be added to the *Rule* data type and have a parser written for it before its implementation is written in the *RuleApplication* module. Adding a mechanism to easily add new rules to the system as well as derive new rules from old ones would make the system useful for longer and more complex proofs rather than just the simple ones given in the worksheets.

Finally, the system can be extended to include new logic systems, such as modal and higher-order logic. This is a complex process, as new formulas, sequent states, rules, and rule application modules need to be written. All inference rules to be implemented must also

be translated into rule transition functions. Parsers must be written for all data types apart from the rule application data type. The logic system may also require auxiliary modules and data types like predicate logic's term and symbol modules. However, the process is much more straightforward as many of the foundations, such as the parser and utility functions, have already been implemented, and plenty of examples can be copied from the propositional and predicate logic modules.

Bibliography

[1] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., USA, 1972.

[2] Alastair Carr. The natural deduction pack, 2018. URL: https://users.ox.ac.uk/~logicman/carr/NDpack.pdf.

[3] Julie Moronuki Christopher Allen. *Haskell Programming from first principles*. Lorepub LLC, 2016.

[4] Lean Prover Community. Natural deduction for propositional logic, Unknown. Accessed: 08/05/2024. URL: https://leanprover.github.io/logic_and_proof/natural_deduction_for_propositional_logic.html#:~:text=In%20natural%20deduction%2C%20every%20proof,from%20B%2CC%2C%E2%80%A6.

[5] Gerhard Gentzen. Investigations into logical deduction. *American Philosophical Quarterly*, 1(4):288–306, 1964. URL: http://www.jstor.org/stable/20009142.

[6] Graham Hutton. Monadic parser combinators, 1996. URL: https://www.cs.nott.ac.uk/~pszgmh/monparsing.pdf.

[7] Graham Hutton. *Programming in Haskell*. Cambridge University Press, second edition edition, 2016.

[8] Stanisław Jaśkowski. On the rules of suppositions in formal logic. 1934.

[9] Daan Leijen and Erik Meijer. Parsec: A practical parser library. 07 2001.

[10] Bryan O'Sullivan. *Real World Haskell*. O'Reilly Media, 1st edition, 2008.

[11] Frank Pfenning. Lecture notes on sequent calculus, 2021. URL: https://www.cs.cmu.edu/~15414/s21/lectures/10-sequents.pdf.

[12] Nicholas Rescher. Reductio ad absurdum. Accessed: 08/05/2024. URL: https://iep.utm.edu/reductio/#:~:text=Reductio%20ad%20absurdum%20is%20a,its%20rejection%20would%20be%20untenable.

[13] Georg Struth. Logic in computer science. 2022. In: Lecture notes.