Automatic Generation of Human Readable Proofs

Xuehan(Maka) Hu

Submitted in partial fulfilment
of the requirements for the degree of

Master of Science

Department of Computer Science
Brock University
St. Catharines, Ontario

Abstract

Declarative sentences are statements constructed from propositions that can be either true or false. To be easily manipulated by a program, we present a formal system for handling such declarative sentences called propositional logic. Currently, widely used methods for automatically generating proofs have readability limitations: the deduction process does not conform to human reasoning processes, or the proof tree generated is too complicated.

The primary objective of this thesis is to design a program that automatically generates human-readable propositional logic proofs using Natural Deduction. Natural Deduction is a calculus for deriving conclusions from a finite set of premises using proof rules. The deduction process is a tree structure with assumptions as leaves, natural deduction rules as nodes, and the conclusion as the root. This calculus models human reasoning very well because it builds proofs incrementally using logical deductions from known facts and assumptions.

Our approach will be capable of proving any valid sentence of Propositional Logic automatically and producing a proof tree in the Natural Deduction calculus.

Acknowledgements

I would like to acknowledge my indebtedness and render my warmest thanks to my supervisor, Professor Michael Winter. His friendly guidance and expert recommendations were crucial during every step of the work.

Motivation

Propositional logic is widely applied in the field of modern computing. As computer science professionals, we want to treat propositions and conclusions formally so that the arguments are sound and can be carefully defended or executed by a machine [6]. Argumentation requires the construction of proofs. Therefore, an automatic theorem prover that performs deduction steps in propositional logic is required.

In the modern computational domain, propositional logic is one of the fundamentals of circuit design. An electronic computer is constructed from circuits made up of logic gates. Every such circuit corresponds to a formula in propositional logic. Conversely, any formula can be implemented by a circuit [1]. Nonetheless, as designs are increasingly complex, errors are costly and dangerous. An automatic theorem prover that can assist us in building circuit designs is required.

However, the issue with current automatic theorem provers is that their proofs are basically not readable and/or do not model human reasoning. Theorem provers tend to generate overly complicated, unreadable proofs. On the other hand, provers use calculi that are well suited for automatic theorem proving but do not model the way humans reason. Thus, we intend to construct an automatic theorem prover with an emphasis on human-readable outputs.

Generating human-readable proofs can benefit various fields that involve Human-Computer Interaction. First, and most clearly, human-readable proofs are desirable for communicating results such as in scientific publications and textbooks. Additionally, an AI system may come to a conclusion based on reasoning in propositional logic. Users of such a system want to understand why the system came up with some conclusion or decision [3]. As an example, an intelligence analyst who gets suggestions from an algorithm for big data analytics. Here we want to know why the algorithm suggests certain things. The proof tells the user why but it needs to be presented in a human-readable form [3].

In this thesis, we will use the Natural Deduction calculus for implementing an automatic theorem prover. Natural Deduction is a logic calculus that models human reasoning very well but is seldom used for automatic theorem provers. We will argue that the proof tree generated by our theorem prover is much better readable than proofs of standard theorem provers.

Contents

List of Tables

List of Figures

Chapter 1

Propositional Logic and Natural Deduction

1.1 Propositional Logic

Declarative sentences are statements constructed by propositions that can be either true or false [10]. The language of propositional logic defines the set of declarative sentences, and it is meaning, i.e. their semantics, formally. This is essential in formulating certain properties and arguments in areas such as law and mathematics [2].

1.1.1 Preliminary Concepts and Terminology

In this section, we want to provide an informal definition of propositional logic. A formal definition will follow in the next chapter. We start with an atomic sentence which is given by propositional variables P and Q. These variables serve as representatives of atomic properties such as

P: I love Brock University.
Q: I get my degree.

Basic sentences can be combined using the following logical connectives [6].
$\neg$: The *negation* of Q, written as $\neg Q$, represents that Q is not true, i.e. "I do **not** get my degree.", or equivalently "It is **not** true that I get my degree."
$\vee$: The sentence $P \vee Q$ is true *iff* at least one of P and Q is true. Also known as the *disjunction* of P and Q[1]. For example, the expression $P \vee Q$ means: "I love Brock

[1]In this thesis we use the abbreviation for 'if and only if'.

University, **or** I get my degree."
$\wedge$: Similarly, the sentence $P \wedge Q$, called the *conjunction* of P and Q, is true iff both P and Q are true, i.e. it represents the statement "I love Brock University **and** I get my degree."
$\rightarrow$: The sentence $P \rightarrow Q$ expresses an *implication*. It is true, if P is true, then also Q is true. In this case, it means: "**If** I love Brock University, **then** I get my degree."
$\leftrightarrow$: $P \leftrightarrow Q$ expresses the fact that P and Q are equivalent. In our example, we get:"I love Brock University **if and only if** I get my degree." However, $\leftrightarrow$ is not part of our logic. We only use this as an abbreviation of (P $\rightarrow$ Q) $\wedge$ (Q $\rightarrow$ P).
As a further example, we can use those connectives to represent the following sentences:

"It is not true that if I love Brock University, then I get my degree.": $\neg (P \rightarrow Q)$
"I love Brock University, and I do not get my degree.": $P \wedge \neg Q$

If we connect them by $\leftrightarrow$, we have:

"It is not true that if I love Brock University, then I get my degree", if and only if "I love Brock University, and I do not get my degree."

$$\neg (P \rightarrow Q) \leftrightarrow P \wedge \neg Q$$

Now, we already know how to convert a declarative sentence into a propositional formula. In the next section, we will use an example to demonstrate the challenges we will encounter. Before that, we will briefly introduce the precedence of the logical symbol, and we will see its application in the following examples.
We all know that multiplication usually binds tighter than addition, i.e. $a + b * c$ usually means $a + (b * c)$. In the same way, to make the proportional formula easier to read, we apply the following standard connectives, namely, [2]:

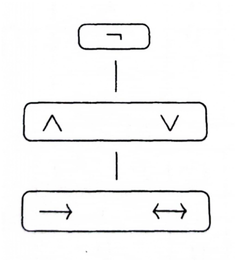


Figure 1.1: Precedence Convention [2]

A connective is placed in Figure 1.1, the higher its precedence indicates higher priority. In our example , $\neg$ binds more tightly than $\wedge$, $\wedge$ tighter than $\rightarrow$ and $\leftrightarrow$ [10]. Therefore, our expression can be read as:

$$((\neg P) \rightarrow Q) \leftrightarrow (P \vee (\neg Q))$$

It should be pointed out that although the symbols $\rightarrow$ and $\leftrightarrow$ have the same precedence, the $\rightarrow$ is inside the brackets. In this case, we will operate the $(\neg P) \rightarrow Q$ first. Now that we recognize some fundamental rules, we are able to give an example of the challenges we will face.

1.1.2 Syntax

The set P can be any set. Its elements are called propositional variables. Usually, we will choose P to be infinite but even a finite set would be sufficient as long as we have enough variables to express the properties at hand.
Propositional logic is based on a set P of propositional variables. The syntax of formulas is defined over the alphabet $P \cup \{(,), \bot, \neg, \wedge, \vee, \rightarrow\}$

Definition 1. [10][6] *Let P be a set of propositional variables. The set* Prop *of propositional formulas is recursively defined by:*
1. Each propositional variable $p \in P$ is a formula, i.e. $P \subseteq \mathrm{Prop}$.
2. $\bot$ is a formula., i.e. $\bot \in \mathrm{Prop}$.
3. if $\varphi \in \mathrm{Prop}$, then $\neg\varphi \in \mathrm{Prop}$.
4. If φ_1, $\varphi_2 \in \mathrm{Prop}$, then
a) $\varphi_1 \vee \varphi_2 \in \mathrm{Prop}$, and
b) $\varphi_1 \wedge \varphi_2 \in \mathrm{Prop}$, and
c) $\varphi_1 \rightarrow \varphi_2 \in \mathrm{Prop}$.

1.1.3 Semantics

In the previous section, we discussed how to use the natural deduction to verify a set of premises $\varphi_1 \cdots \varphi_n$ are valid, denoted by $\varphi_1 \cdots \varphi_n \vdash \psi$. In this section, we defined a new notation.

Definition 2. [10] A truth assignment is a function $v : P \rightarrow \mathbb{B}$ from the propositional variables into the set of truth values $\mathbb{B} = \{T, F\}$.

We want to give meaning to a formula by defining when a formula is valid or true. This is based on a truth assignment that already assigns a truth value to each propositional variable.

Definition 3. [10] Let v be a truth assignment. The extension $\bar{v}$: Prop $\to \mathbb{B}$ of v is defined by:

1. $\bar{v}(p) = v(p)$ for every $p \in P$
2. $\bar{v}(\bot) = \mathrm{F}$.
3. $\bar{v}(\neg\varphi) = \begin{cases} \mathrm{T} & \text{if } \bar{v}(\varphi) = \mathrm{F}, \\ \mathrm{F} & \textit{if otherwise.} \end{cases}$
4. $\bar{v}(\varphi_1 \wedge \varphi_2) = \begin{cases} \mathrm{T} & \text{if } \bar{v}(\varphi_1) = \mathrm{T} \textit{ and } \bar{v}(\varphi_2) = \mathrm{T}, \\ \mathrm{F} & \textit{if otherwise.} \end{cases}$
5. $\bar{v}(\varphi_1 \vee \varphi_2) = \begin{cases} \mathrm{T} & \text{if } \bar{v}(\varphi_1) = \mathrm{T} \textit{ or } \bar{v}(\varphi_2) = \mathrm{T}, \\ \mathrm{F} & \textit{if otherwise.} \end{cases}$
6. $\bar{v}(\varphi_1 \to \varphi_2) = \begin{cases} \mathrm{T} & \text{if } \bar{v}(\varphi_1) = \mathrm{T} \textit{ and } \bar{v}(\varphi_2) = \mathrm{F}, \\ \mathrm{F} & \textit{if otherwise.} \end{cases}$

v is said to satisfy a formula φ iff $\bar{v}(\varphi) = \mathrm{T}$. Additionally, v is said to satisfy a set $\Sigma \subseteq$ Prop of formulas iff v satisfies φ for all $\varphi \in \Sigma$.
Let Σ be a set of formulas and φ a formula. Then φ follows from Σ ($\Sigma \models \varphi$) iff every truth assignment that satisfies all formulas in Σ also satisfies φ.

1.2 Natural Deduction

Around 1877, mathematicians began to examine purely syntactic approaches to logic, using propositional formulae as axioms and adding procedures for constructing new formulas, known as rules of inference. This combination of axioms and rules that allow deriving true formulas out of the axioms and assumptions will be referred to as a propositional proof system [2], or a deduction system.

1.2.1 Motivation

Let us consider the following human deduction process. Suppose we have declarative sentences P and Q given premises:

Example 1. Consider the following declarative sentences P and Q:

P: It rained.

Q: The street is wet. $P \to Q$: If it rained, then the street is wet.

If we know that it rained and that the street is wet if it rained, we may use these two pieces of evidence to conclude that the street is actually wet [6].
The human deduction, as shown by this example, typically involves reasoning from given a set of premises to reach a valid conclusion. It's an incremental procedure that follows logic, which allows inferring properties from premises. What if we would like to have a set of proof rules, like the human deduction, by which we are able to infer one premise from another premise, and conclude a valid conclusion? In this chapter, we will introduce how to use natural deduction to calculate propositional logic.

1.2.2 Notations

The Natural Deduction Calculus consists of several proof rules. They allow us to deduce one formula from one or more previously derived formulas. We can reach a conclusion after incrementally inferring those formulas [6][10].
Suppose we have a set of formulas $\varphi_1, \varphi_2, \varphi_3, \ldots, \varphi_n$, which represented the premises we mentioned earlier. After we apply the rules to the formulas above, we obtain a new set of formulas. After repeating this process we may end with the desired conclusion ψ. We will denote this fact by:

$$\varphi_1 \cdots \varphi_n \vdash \psi.$$

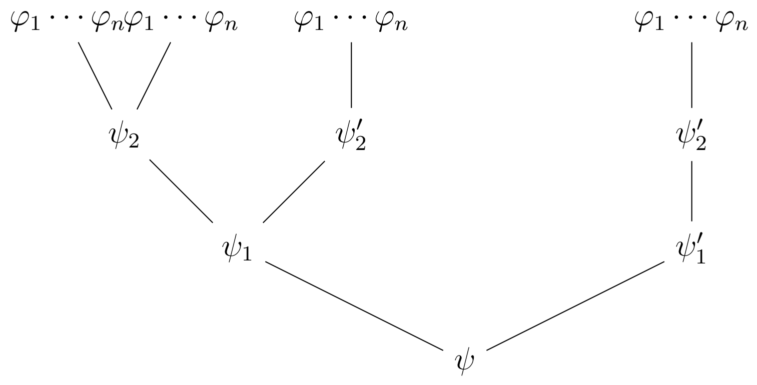

$\varphi_1 \cdots \varphi_n \varphi_1 \cdots \varphi_n$
$\varphi_1 \cdots \varphi_n$
$\varphi_1 \cdots \varphi_n$
ψ_2
ψ_2'
ψ_2'
ψ_1
ψ_1'
ψ

Figure 1.2: The Natural Deduction Tree Structure

In the tree structure shown in Figure 1.2, the leaves $\varphi_1 \cdots \varphi_n$ represent a series of premises or assumptions. We will discuss later when it is necessary to create new assumptions and when it is necessary to discard assumptions. The nodes ψ_1, ψ_1', ψ_2, and ψ_2' indicated intermediate formulas during application of rules. The ψ at the root represents the conclusion we want to achieve. Now that we understand the fundamental structure of proof in natural deduction, the following chapter will introduce calculus formally. But before we get there, let us review our previous Example 3 and see how it is transformed into the proof we need.

1.2.3 Rules

The rules of Natural Deduction mainly consist of introduction and elimination rules for each logical connective. An introduction rule allows one to derive a formula which uses the connective in question and an elimination rule removes the connective from one of the previously derived formulas. In addition to these rules, the calculus of natural deduction adds the proof-by-contradiction rules which is neither an introduction nor an elimination rule.

Example 2. If we have declarative sentences P and $P \to Q$, which P be a assumption: "It rained", and $P \to Q$ be a assumption "If it rained, then the street is wet.", then we can figure out another declarative sentence Q:

Q: The street is wet.

The rule formalizing this reasoning is called implication elimination[2]. [6][10].

Rule 1. Given φ, and $\varphi \to \psi$, we can conclude the ψ.

$$\frac{\varphi \quad \varphi \to \psi}{\psi} \to E$$

With the annotation $\to$ E at the rule, we indicate that this step uses the implication elimination rule. This will be done similarly for all other rules as well.

Rules with Temporary Assumptions

Sometimes, we may need to add temporary assumptions in order to obtain the desired conclusion. Let us consider the following example.

[2]The introduction elimination, also known as arrow-elimination, is the contemporary term for this rule. It is also known by its Latin name, modus ponens.

Example 3. If we want to derive the formula $P \to Q$, we can make a temporary assumption P, and provide a proof of Q using that assumption, i.e., we have a proof tree of the form:

$$\begin{array}{c} P \\ \vdots \\ Q \end{array}$$

Now we can conclude $P \to Q$ and discard the no longer needed assumption P. In this case, we can **discard** the assumption, written as:

$$[P]$$

The overall idea of making or discarding an assumption is that we create a new assumption when we need it to derive the sub-goal or the sub-formula we want to prove and discard it after we achieve the goal.

Rule 2. This **Implication Introduction** rule is written as:

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

As mentioned above, the calculus of natural deduction contains one rule that is neither an introduction nor an elimination rule.

Rule 3. If we can show that the assumption $\neg\varphi$ leads to a contradiction $\bot$, then we can conclude φ [10]. The rule is called proof-by-contradiction, or **PBC** for short, and is formally given by:

$$\dfrac{\begin{array}{c} [\neg\varphi] \\ \vdots \\ \bot \end{array}}{\varphi}\ PBC$$

The remaining rules are similar to the ones introduced above so we will not introduce them in detail. We have listed all the rules in the table below:

	introduction rule	elimination rule
$\land$	$\dfrac{\varphi \quad \psi}{\varphi \land \psi}\ \land\mathrm{I}$	$\dfrac{\varphi \land \psi}{\psi}\ \land\mathrm{E1} \qquad \dfrac{\varphi \land \psi}{\psi}\ \land\mathrm{E1}$
$\lor$	$\dfrac{\varphi}{\varphi \lor \psi}\ \lor\mathrm{I1} \qquad \dfrac{\psi}{\varphi \lor \psi}\ \lor\mathrm{I2}$	$\dfrac{\varphi \lor \psi \quad \begin{array}{c}[\varphi]\\ \vdots \\ \chi\end{array} \quad \begin{array}{c}[\psi]\\ \vdots \\ \chi\end{array}}{\chi}\ \lor\mathrm{E}$
$\to$	$\dfrac{\begin{array}{c}[\varphi]\\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\ \to\mathrm{I}$	$\dfrac{\varphi \quad \varphi \to \psi}{\psi}\ \to\mathrm{E}$
$\neg$	$\dfrac{\begin{array}{c}[\varphi]\\ \vdots \\ \bot\end{array}}{\neg\varphi}\ \neg\mathrm{I}$	$\dfrac{\varphi \quad \neg\varphi}{\bot}\ \neg\mathrm{E}$
PBC	$\dfrac{\begin{array}{c}[\neg\varphi]\\ \vdots \\ \bot\end{array}}{\varphi}\ \mathrm{PBC}$	

Table 1.1: Summary of Natural Deduction Rules for Propositional Logic [10]

If we provide a proof in natural deduction of a formula φ from some assumptions $\varphi_1, \ldots, \varphi_n$, then we would expect that φ also follows from $\varphi_1, \ldots, \varphi_n$ in the sense of Definition 2. This property is called the soundness (or correctness) of the calculus. We mention this result here and refer to Theorem 13 [6].

Theorem 1.2.1 (Soundness). *If $\varphi_1, \ldots, \varphi_n \vdash \varphi$, then $\varphi_1, \ldots, \varphi_n \models \varphi$.*

1.2.4 Completeness

The completeness theorem of the calculus of natural deduction states that every true statement can also be proven, i.e., it is the opposite implication of the soundness theorem.

Theorem 1.2.2 (Completeness). *Let $\varphi_1, \cdots, \varphi_n$ and ψ be propositional formula. If $\varphi_1, \cdots, \varphi_n \models \psi$, then $\varphi_1, \cdots, \varphi_n \vdash \psi$.*

In this section, we want to outline the proof of this theorem because it leads to an algorithm for constructing a proof tree for a true formula.

Completeness and Truth Tables

A truth table consists of several rows, each corresponding to a truth assignment. Since the validity of a formula only depends on the truth values of the propositional variables that actually occur in the formula, the truth table has 2^n rows if the formula uses n variables. The columns of the table list the propositional variables and the formula in question. If needed, subformulas can be added. The values in a column of the table indicate whether the variable or formula is true for the truth assignment corresponding to the row.

p	q	ψ
T	T	T
T	F	T
F	T	T
F	F	T

In the table above ψ is assumed to contain only the propositional variables p and q. The first row of the table corresponds to any truth assignment v with $v(p) = T$ and $v(q) = T$. Since we have two variables, the table has four rows. The column of ψ consists of T entries only, showing that the formula ψ is valid. As a first step, we want to construct a proof of ψ for each row of the table. In order to do so we need to encode that we refer to the second row of the table. On this row, we have:

$$v(p) = T \quad v(q) = F$$

The proof that we produce will be a proof of ψ using assumptions based on p and q. Since $v(p) = T$, we will use p as an assumption. Similarly, since $v(q) = F$, we will use $\neg q$ as an assumption. In general, depending on the truth value that is assigned to the variable in the corresponding row, we will use the variable itself or not of the variable as an assumption.
The following lemma shows that we can extract a proof for each row of a truth table.

Lemma 1.2.3. *[10] Let φ be a formula with propositional variables among $p_1 \cdots p_n$, and v be a truth assignment, define:*

$$\hat{p_i} := \begin{cases} p_i & \text{if } v(p_i) = T \\ \neg p_i & \text{if } v(p_i) = F \end{cases}$$

And we have:
1. $\hat{p_1}, \cdots, \hat{p_n} \vdash \varphi$ *if* $\bar{v}(\varphi) = T$.
2. $\hat{p_1}, \cdots, \hat{p_n} \vdash \neg\varphi$ *if* $\bar{v}(\varphi) = F$.

The lemma is constructive and induces a recursive procedure based on the structure of ψ. In our example, we generate the following proof by induction on ψ.

$$p, \neg q \vdash \psi$$

Since the lemma above, and, hence, the recursive procedure generating a proof, is shown by induction on the formula ψ and subformulas of a true formula need not to be true again, we actually need both cases of the lemma. For example, if ψ is the formula $\neg\psi'$ for some ψ', then ψ' is false in the second row of the table so that the second case of the lemma applies. We obtain a proof $p, \neg q \vdash \neg\psi'$ which is also a proof $p, \neg q \vdash \psi$.
The overall idea of lemmas of completeness is producing proofs for every row, and combing proofs after that. For example, the lemma generates the following proofs for Row 1 and 2 of the table:

$$p, q \vdash \psi$$

$$p, \neg q \vdash \psi$$

The two proofs only differ in the assumptions q resp. $\neg q$. These can be eliminated by combining the two proofs into one proof $p \vdash \psi$ by using the formula $q \vee \neg q$. The following lemma states that fact. Similarly, we can combine rows 3 and row 4, by eliminating those assumptions. In the end, we can generate the proof by the tree structure.

p	q	ψ	
T	T	T	$p, q, \vdash \psi$
$\underline{\text{T}}$	$\underline{\text{F}}$	$\underline{\text{T}}$	$p, \neg q, \vdash \psi$
F	T	T	$\neg p, q, \vdash \psi$
F	F	T	$\neg p, \neg q, \vdash \psi$

$$\begin{tikzcd}
p, q \vdash \psi \arrow[dr, dash] & & p, \neg q \vdash \psi \arrow[dl, dash] & & \neg p, q \vdash \psi \arrow[dr, dash] & & \neg p, \neg q \vdash \psi \arrow[dl, dash] \\
& p \vdash \psi \arrow[drr, dash] & & & & \neg p \vdash \psi \arrow[dll, dash] & \\
& & & \psi & & &
\end{tikzcd}$$

For each line of the truth table, we are going to construct a proof. If the formula has n propositional variables, the truth table has 2^n rows. We assemble those 2^n proofs into a single proof [10].

Chapter 2

Natural approach to Proofs in Natural Deduction

2.1 Example Using Natural Deduction

In this section, we will use an example to show how our Natural Deduction rule helps us generate proofs. But before we get to that, you might be wondering how we know when to use which rule. We only had assumptions and a conclusion when we set up the proof. How do we build the intermediate proofs step by step? More specifically, how do we choose the "correct rule" to get to the desired conclusion goal? Let us start by discussing two ideas that will assist our reasoning steps: Applying deduction rules forward and backward.

2.1.1 Deduction Rules Forward and Backward

In the **backwards** strategy, you begin with the conclusion you desire to start proving and work backwards to the premises. You break down the desired formula into smaller sub-formulas and try to find any connections to the premises. This strategy is usually used in the first step of an introduction. For example, when you write down your conclusion formula at the bottom of the page and look for any rule that could connect it to the desired conclusion formula.
On the other side, with the **forward** strategy, you begin with the premise formulas and work forward to the desired conclusion formulas. This way is more like human reasoning, generating proofs and intermediate steps incrementally by applying Natural Deduction Proof based on known premises and assumptions.
Let us look at an example of how forward and backward induction strategies can be

used.

Example 4. Give a derivation of $\neg p, p \vee q \vdash p$ in natural deduction, or show there is proof using natural deduction.

On the L.H.S.[1] of the $\vdash$, the expression $\neg p, p \vee q$ means there exists a tree[2] with no more than 2 assumptions, i.e., with two assumptions that have not been discarded.

Most of the time, we will do the Natural Deduction application on those formulas on L.H.S. On the R.H.S. [3], the conclusion meaning the formula at the very root of this tree is p.
Next, we begin the induction step. The first step is to **write down everything** we know under the given derivation. We placed our assumptions at the top and our conclusion at the bottom, leaving enough space between them so we could later apply the Natural Deductions rules to generate intermediate steps.
Then, we tried applying the deduction rules. Remember how the deduction rules look: there is a new symbol under the line, and there are sub-trees above, and we have the symbol on the R.H.S. Even if we are unsure which rule applies at this moment, we can write a line above the conclusion q.

$$\frac{p \vee q \quad \neg p}{q}$$

Now, we are considering which rule should apply here. q is variable, and it seems hard to figure out which rule we should follow, as the variable, is already the atomic component. Do not forget we have assumptions formulas here: $p \vee q$, and $\neg p$. Let us try to make any connection between the conclusion q, and those assumptions. So, there are things we can do here:

1. Considering any rule with a single conclusion could help us deal with q.
2. Considering any rule with assumptions could help us deal with $p \vee q$, and $\neg p$.
3. Once we have a list of optional rules from steps 1 and 2, we choose the one that connects them.

[1] L.H.S. is the abbreviation of the left-hand side.
[2] The tree structure see the Figure 1.2.
[3] R.H.S. is the abbreviation of the right-hand side.

rules deal with the conclusion q	rules deal with the assumption $p \vee q$	rules deal with the assumption $\neg p$
$\wedge E_1$		
$\wedge E_2$		
$\vee E$	$\vee E$	
$\rightarrow E$		
P.B.C.		~~$\neg I$~~
		$\neg E$

Here it is, the $\vee E$ rule from whom the conclusion of p assumptions is $p \vee q$. So, we take the formula $p \vee q$ as the leaf and start applying the $\vee E$ **rule backward**. There are certain Natural Deduction rules that we know we will not consider when listing them, such as the Not Introduction. Because we can only use this induction rule backwards, but we cannot do it there as the $\neg p$ is an assumption. However, after we choose $\neg E$, this brings up a new issue: As we do not have assumptions on the L.H.S. and R.H.S. of the relation $\vee$, how can we complete sub-trees that are above the $\vee E$ rule?

~~$p \vee q$~~ $\qquad\qquad$ $\neg p$

$$\dfrac{p \vee q \quad \begin{matrix}[p]\\ \vdots \\ \chi\end{matrix} \quad \begin{matrix}[q]\\ \vdots \\ \chi\end{matrix}}{q}\ \vee E$$

Consider this scenario: You need something to complete a task, but you do not have it. The only thing you can do is - borrow it. Let us "borrow" temporary assumptions p and q. Of course, we must "remember" what we "borrowed". Here we use the add the notation E^1 to indicate those temporary assumptions generated under this rule. Later, we'll have to get rid of these assumptions, as those temporary assumptions have to be discarded to make the proof of truth.

~~$p \vee q$~~ $\qquad\qquad$ $\neg p$

$$\dfrac{p \vee q \quad \begin{matrix}[p]\\ \vdots \\ p\end{matrix} \quad \begin{matrix}[q]\\ \vdots \\ q\end{matrix}}{q}\ \vee E$$

Before we go to the next step, let us see what we have now. We have two temporary assumptions p and q, which will be discarded later, and we have an assumption $\neg p$.

How can we make the connection between them? Here we use similar procedures as before:

1. Considering any rule with a single conclusion could help us deal with the assumption $\neg p$.
2. Considering any rule with a single conclusion could help us deal with the temporary assumptions p.
3. Considering any rule with a single conclusion could help us deal with the temporary assumptions q.
4. Once we have a list of optional rules from steps 1 2 and 3, we choose the one that connects them.

rules deal with the assumption $\neg p$	rules deal with the temporary assumption p	rules deal with the temporary assumption q
$\neg E$	$\vdots$	$\vdots$

It is obvious we can use the $\neg E$ **rules forward** here.

$$\frac{\varphi \quad \neg\varphi}{\perp}\ \neg E$$

Under the rule, you need two sub-trees, one is p and the other one is $\neg p$. After we eliminate $\neg p$ and p by the $\neg E$, it gives us $\perp$ as the conclusion. Currently, we have two proofs:

~~$p \vee q$ $\neg p$ $[p]$~~

$$\frac{p \quad \neg\, p}{\perp}\ \neg E$$

$$\frac{p \vee q \quad \begin{array}{c}[p]\\ \vdots\\ p\end{array} \quad \begin{array}{c}[q]\\ \vdots\\ q\end{array}}{q}\ \vee E$$

Here we are almost done, but we have two things left:

1. There is no connection between the conclusion $\perp$ and the temporary assumption q on the sub-trees of the E proof.
2. On the L.H.S. of the $\neg E$ proof, we have the temporary assumption p, which we no longer utilize.

Now if you look at the rule, you can recognize this is exactly the rule by contraction[4].

$$\begin{array}{c}[\neg\varphi]\\ \vdots\\ \dfrac{\bot}{\varphi}\;PBC\end{array}$$

By taking $\neg$ contradiction of $\bot p$ is truth. By the *P.B.C.* rule, we are supposed to discard the temporary assumption $\neg p$. However, we already have the $\neg p$ as an assumption here. Since we are not "borrowing" it, we do not need to "return" it. Here is the proof tree we generated:

$$\dfrac{p \vee q \qquad \dfrac{\dfrac{\neg p \quad [p]^1}{\bot}\;\neg E}{q}\;PBC \qquad [q]^1}{q}\;\vee E^1$$

The root q is on the R.H.S. of the $\vdash$, and the open assumptions $\neg p$ and $p \vee q$ are the formulas on the L.H.S.

[4]The P.B.C. can apply in almost every situation. However, we only use it when necessary. Here the P.B.C. is the only rule you can apply in this situation.

Chapter 3

Challenges and Progress in Proof Generation

In this section, we want to compare our approach with existing tools and methods for generating proofs automatically. First, we will investigate automatic theorem provers using Prover9 as an example. Second, we will outline how the constructive proof of completeness of propositional logic leads to an algorithm generating a proof in natural deduction automatically. Last but not least, we will demonstrate our approach by an example using the interactive theorem prover Coq before presenting the algorithm in general.

3.1 Automatic Theorem Provers

The resolution calculus is such a deduction system that is usually used by automatic theorem provers because the rules of the system are very well-suited for computer manipulation. In general, a proof by resolution works always by contraction. This is done by converting every formula in conjunction normal form, employing resolution rules and trying to derive contradictions, after assuming that the formula you wish to prove is false. In CNF[1], a formula is a conjunction AND of clauses, where each clause is a disjunction OR of literals, either a variable or its negation [6]. This structure makes it easier to perform certain computational tasks, such as validity checking [6]. A proof by contradiction indicates that the initial formula was valid.
These automated theorem provers are extremely powerful and can automatically generate a proof once provided formulas representing the problem [7]. However, the

[1]The abbreviation for conjunction normal form.

generated outputs are usually not very well readable for a human being.

3.1.1 Example

In order to demonstrate [7] the approach of automatic theorem provers and resolution we will use an example of Prover9.

Example 5. Given a derivation: $\vdash (((p \to q \vee s)) \wedge ((p \wedge r \to s)) \wedge ((s \wedge t) \to (p \vee \neg q))) \to (((p \wedge (q \to r)) \to s) \wedge (((q \wedge s) \wedge t \to p))$ and show there is a proof by the Prover System.

The input for Prover9 looks like this. Here we use "|" represent $\vee$, "&" represent $\wedge$, and "$- >$" represent $\to$.

```
formulas(goals).
    (((p -> (q | s)) & ((p & r) -> s)) & ((s & t) -> (p | -q)))
    -> (((p & (q -> r)) -> s) & (((q & s) & t) -> p)).
end_of_list.
```

In our example, there is only one formula in our (*goals*), so the process of *Prover*9 is clear: the formula takes its universal closure, such as bounding all free proportional variables, then negates [7]. Then the resolution tries to falsify the negated goal, i.e. to show that it is not true.
Here is the output generated by this example:

```
% -------- Comments from original proof --------
% Proof 1 at 0.01 (+ 0.00) seconds.
% Length of proof is 5.
% Level of proof is 2.
% Maximum clause weight is 2.000.
% Given clauses 0.



1 (p -> q | s) & (p & r -> s) & (s & t -> p | -q) -> (p & (q
   -> r) -> s)
& (q & s & t -> p) # label(non_clause) # label(goal).  [goal
   ].
2 -p | q | s.  [deny(1)].
3 -p | -r | s.  [deny(1)].
4 -s | -t | p | -q.  [deny(1)].
5 p | q.  [deny(1)].
6 p | s.  [deny(1)].
7 p | t.  [deny(1)].
8 -q | r | s.  [deny(1)].
9 -s | -p.  [deny(1)].
10 p | -s | -q.  [resolve(7,b,4,b),merge(c)].
11 p | -q.  [resolve(10,b,6,b),merge(c)].
12 p.  [resolve(11,b,5,b),merge(b)].
13 -s.  [back_unit_del(9),unit_del(b,12)].
14 -r.  [back_unit_del(3),unit_del(a,12),unit_del(c,13)].
15 q.  [back_unit_del(2),unit_del(a,12),unit_del(c,13)].
16 $F.  [back_unit_del(8),unit_del(a,15),unit_del(b,14),
   unit_del(c,13)].
```

As is obvious, this result is hard to read. Even though Prover9 incorporates the rules of classical propositional logic, it is still challenging for us to fully understand the middle stages of the reasoning process. Generally, two features make the result difficult to read: the label and the induction steps. The general strategy of induction in the Prover9 system is to rewrite initial formulas, eliminate rewritten formulas, and attempt proof via contradiction. Here we will begin by describing these labels.

Line1 : This line expresses the initial formula we want to prove: ⊢ (((p→q∨s)) ∧ ((p∧r→s)) ∧ ((s∧t) → (p∨¬q))) → (((p∧(q→r)) →s) ∧ (((q∧s)∧t→p)). The label [*goal*] indicates the input formula we are supposed to do with the negations [7]. Since the comment indicates "non_clause", we only refer to "1" when referring to the first line.

- The label [*deny*(1)] appears from lines 2 to 9, showing the CNF translation negating our [*goal*] [7].

Line2 : Here we have (p → q) ∨ s. First we deny the initial formula, so we get: ¬ ((p → q) ∨ s). As we know $\alpha \rightarrow \beta \equiv (\neg\alpha) \vee \beta$ by the implication rule, we can get the ¬ ((p → q) ∨ s) $\equiv \neg((\neg p \vee q) \vee s)$. By the De Morgan's Law, we can get $\neg(\neg p \vee q) \wedge \neg s \equiv \neg\neg p \wedge \neg q \wedge \neg s \equiv p \wedge \neg q \wedge \neg s$, which is the output we generated.

Line3 : Consider this part of our initial formula: (p∧ r)→ s, then we have ¬ (p ∧ r)∨ s by implication, and ¬ p ∨¬ r ∨ s by De Morgan's Law.

Line4 : Similarity, $(s \wedge t) \rightarrow (p \vee \neg q)$ can be written as $\neg(s \wedge t) \vee (p \vee \neg q)$ by implication, and ¬ s ∨ ¬ t ∨ p ∨ ¬ q by De Morgan's Law.

Line5 : The initial formula is (p ∧ (q → r)→ s). First, we take the negation of the initial formula, which is ¬ (p ∧ (q → r)→ s), it equivalent to (¬ p ∨ ¬ (q ∧ r)), and ¬ p∨¬ q ∨¬ r by De Morgan's Law.

Line 6 Let us focus on the part of the formula p∧ (q→ r)→ s and take the negation, we have ¬ (p∧ (¬ q∨ r))∨ s, and we get ¬ p∨ (¬ ¬ q∧¬ r)∨ s, then we get ¬ p∨ (q∧ ¬ r)∨ s. So we can conclude ¬ p∨ q∨ s. As we already have ¬ p∨¬ r∨ s at Line 3, here we only remain the p∨ s.

Line 7 The initial formula is s∧ t → p ∨¬ q, we take its negation, and we can get s∧ t∧¬ p∧ q. So we have t∧ ¬ p, which is t ∨ p by De Morgan's Law.

Line 8 We have p∧(q→ r)→ s, and the negation is p∧ (¬ q∨ r)∧ ¬ s. Then we have ¬ q ∨ r ∨¬ s

Line 9 We focus on the part of the formula s∧ t→ p∨¬ q and take its negation, and we get s ∧ t∧¬ p∧ q, so we have q ∧ r.

- Then we eliminate those assumptions step by step.

Line 10 The formula p ∨¬s ∨ ¬ q derived by eliminating p ∨ t with ¬ s∨ ¬ t∨ p∨¬ q. In line 5, the label [resolve(7, b, 4, a)] indicates that the second literal of clause 7 is resolved with the second literal of clause 4 [7]. [merge(c)] indicates the third literal has been removed since it was the same as the one before it[7].

Line 11 The result of eliminating p∨¬ s ∨ ¬ q and p∨ s is p ∨¬ q.

Line 12 This p derived by eliminating p∨¬ q and p∨ q.

Line 13 If p is true (row 1), then ¬ p∨ s to be true, s must also be true. However, if s is true, then ¬ s is false. As we know p is true and s is true, we conclude: ¬ s is false (row 13).

Line 14 We know p is true (row 12), s is false (because ¬ s is T row 13), and ¬ p∨ ¬r∨ s (row 3). p is true and s is false, ¬ r must be T. Thus, we get r is false.

Line 15 Similarly, we have p is true (row 12) and s is false, and we have ¬ p ∨ q ∨ s is true, so we have q is true.

Line 16 The result leaves us with the conclusion: False. Therefore, we finished the proof.

Constraints of *Prover9*

As is obvious, the *Prover*9 and other resolution calculators are powerful but do not model human reasoning well. Humans typically build proofs incrementally using logical deductions from known facts and assumptions. Furthermore, it uses labels. This does not lead to human-readable proof and cannot easily be converted into plain text.

3.2 Proof by Induction on the Structure of Formulas

As we already know, the Propositional Logic model basically gives each proportional variable a truth value. Propositional Logic is decidable, we can construct a formula and construct a truth table, for each line and produce one proof according to the lemma 1.2.3. Let us consider the following example:

Example 6. Follow the lemma to find the derivation ⊢ (((p→q∨s)) ∧ ((p∧r→s)) ∧ ((s∧t) → (p∨¬q))) → (((p∧(q→r)) →s) ∧ (((q∧s)∧t→p)).

Using the notation below, simplify the truth table for the formula ψ:
Then we generate the truth table based on the valuation of propositional variables. In this example, we only consider this signed valuation: v(p)=F, v(q)=T, v(r)=F, v(s)=T, v(t)=F, and the corresponding row on this truth table is:

$\psi = A \to B$

$A = (A1 \wedge A2) \wedge A3 \quad A1 = p \to (q \vee s)$

$A2 = (p \wedge r) \to s \quad A3 = (s \wedge t) \to (p \vee \neg q)$

$B = C \wedge D \quad C = (p \wedge (q \to r)) \to s$

$D = E \to p$

$E = F \wedge t \quad F = q \wedge s$

A	B	C	D	E	F	ψ
T	T	T	T	T	T	T
			⋯			
T	T	T	T	F	T	T
			⋯			

The proof tree produced by this truth table row is shown below; one branch is omitted. The overall idea is following the lemma 1.2.3, and induction on the structure of the formula. By the induction hypothesis, we get either a derivation $\hat{p_1} \cdots \hat{p_n} \vdash \psi$ or $\hat{p_1} \cdots \hat{p_n} \vdash \neg\psi$ [10]. In our example, we only consider the situation when the ψ is truth. Above those valid formulas $\neg A \vee B \to (A \to B)$, $\neg E \vee p \to (E \to p)$ and $\neg F \vee \neg t \to \neg(F \wedge t)$, are sub-proofs that do not contain any open assumptions.

$$\dfrac{\begin{array}{c}\vdots\\ \neg A \vee B \to (A \to B)\end{array} \qquad \dfrac{\dfrac{\begin{array}{c} omit\\ \vdots \\ C\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \neg E \vee p \to (E \to p)\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}\vdots\\ \neg F \vee \neg t \to \neg(F \wedge t)\end{array} \qquad \dfrac{\neg t}{\neg F \vee \neg t}\,\vee I2}{\neg E}\,\to E}{\neg E \vee p}\,\vee I1}{D}\,\to E}{B}\,\wedge I}{\neg A \vee B}\,\vee I2}{A \to B}\,\to E$$

Let us break down the generation process of this proof tree.

- We begin with the conclusion A $\to$ B. According to the lemma 1.2.3, the structure of the formula is $\varphi_1 \to \varphi_2$: If $\bar{v}(\psi) =$ T, then we have $(\bar{v}(\varphi_1) = F) \vee (\bar{v}(\varphi_2) = T)$. Given that ψ is true, $(\bar{v}(A) = F)$, and $(\bar{v}(B) = T)$, we are able to derive $\neg A \vee B$ using $\vee$I1 or $\vee$I2. We generate the proof by the rule of $\to$ E. On the left-hand side sub-proof, we use the implication to connect the right-hand side sub-proof and the conclusion.

- Then, we do the induction on the formula B. The formula B is true and the structure of the formula is $\varphi_1 \vee \varphi_2$. According to the lemma 1.2.3, we have $(\bar{v}(\varphi_1) = T) \wedge (\bar{v}(\varphi_2) = T)$. Here we have $(\bar{v}(\varphi_1) = T)$ and $(\bar{v}(\varphi_2) = T)$. So we combine those derivations using $\wedge$ I[10].

- Similarly, we do the induction on the formula D. The formula is true, and the structure of the formula is $\varphi_1 \to \varphi_2$. We have $(\bar{v}(\varphi_1) = F) \vee (\bar{v}(\varphi_2) = T)$, so we put the $\neg E \vee p$ on the right-hand side sub-proof. Here we have $\bar{v}(\neg E) = \mathrm{T}$ but $\bar{v}(p) = \mathrm{F}$, we use $\vee I1$ as our next branch.

- Here we find the sub-proof starting with the conclusion $\neg$ E. The formula is true, and the structure is $\varphi_1 \vee \varphi_2$. As stated by the lemma 1.2.3, we have $(\bar{v}(\varphi_1) = T) \vee (\bar{v}(\varphi_2) = T)$. Here we have $\neg$ t is true but $\neg$ F is false. So we apply the $\vee$ I2, and this brings our branch to a close.

Constraints of the proof generated by induction of formula

However, this is only one row of the truth table. As we have 5 propositional variables, we have $2^5 = 32$ rows in our truth table, and each row has a distinct signed valuation. Then we eliminate eliminate p_n in $\hat{p}_1, \ldots, \hat{p}_{n-1}, p_n \vdash \psi$ and $\hat{p}_1, \ldots, \hat{p}_{n-1}, \neg p_n \vdash \psi$ to $\hat{p}_1, \ldots, \hat{p}_{n-1} \vdash \psi$ by the lemma 1.2.3, and combine them into one proof. In the end, we can make a big proof with 582 discard labels and 31 natural deduction proofs used with the $\vee$ E rule.
As can be seen, the proof tree for this method is huge. The construction of this proof uses a complete case distinction over all propositional variables. Although this procedure creates a proof for every true formula, it can hardly be called human-readable.

3.2.1 A Coq example.

Coq is a system for managing formal proofs, it provides a formal language for defining mathematics [8]. The deduction provides tactics of natural deduction proofs, which simplify the deduction process. Here, we will use Coq with the Natural Deduction package [10] to demonstrate our approach because it defines tactics corresponding to exactly one rule of natural deduction. Tactics in Coq are usually more powerful and can do multiple steps at once. We will use this package to show a simple proof example, proving the “human-readable” proof the way we want to:

```
Require Import NaturalDeduction.
Lemma Example (p q r s t :Prop) : (((p -> (q \/ s)) /\ ((p /\
    r) -> s)) /\ ((s /\ t) -> (p \/ ~q))) -> (((p /\ (q -> r)
   ) -> s) /\ (((q /\ s) /\ t) -> p)).
Proof.
1.    Impl_Intro.
2.    And_Intro.
3.    Impl_Intro.
4.    And_Elim_all in H.
5.    And_Elim_all in H0.
6.    Impl_Elim in H and H0.
7.    Or_Elim in H5.
8.    assume (p/\r).
9.    Impl_Elim in H2 and H5.
10.   assumption.
11.   And_Intro.
12.   assumption.
13.   Impl_Elim in H3 and H4.
14.   assumption.
15.   assumption.
16.   Impl_Intro.
17.   And_Elim_all in H.
18.   And_Elim_all in H0.
19.   assume (s /\t).
20.   Impl_Elim in H1 and H5.
21.   Or_Elim in H7.
22.   assumption.
23.   PBC.
24.   Not_Elim in H6 and H0.
25.   assumption.
26.   And_Intro.
27.   assumption.
28.   assumption.
29.   Qed.
```

Let us examine the code sequentially:

- In Lemma TestProperty,"(p q r s t : Prop)" indicates p q, r, s and t are formulas. On the RHS, $\vdash$ (((p$\rightarrow$q$\vee$s)) $\wedge$ ((p$\wedge$r$\rightarrow$s)) $\wedge$ ((s$\wedge$t) $\rightarrow$ (p$\vee\neg$q))) $\rightarrow$ (((p$\wedge$(q$\rightarrow$r)) $\rightarrow$s) $\wedge$ (((q$\wedge$s)$\wedge$t$\rightarrow$p)) is what we want to prove. The output generated is the current stage of the proof.

- Between the 'Proof.', and 'Qed.' is where the proof process is written. The output generated is attached in the comments below each instruction.

Line1 We look at the formula, the structure is $\varphi_1 \rightarrow \varphi_2$, so we apply the $\rightarrow$ I rule backwards. In the code we use Impl_Intro, and generate 1 subgoal. The rule replaces the current goal with (((p$\wedge$(q$\rightarrow$r)) $\rightarrow$s) $\wedge$ (((q$\wedge$s)$\wedge$t$\rightarrow$p)), and add the assumptions H: $\vdash$ (((p$\rightarrow$q$\vee$s)) $\wedge$ ((p$\wedge$r$\rightarrow$s)) $\wedge$ ((s$\wedge$t) $\rightarrow$ (p$\vee\neg$q))).

- After generating hypotheses using the Impl_Intro rule, we start preprocessing these hypotheses, which will be done in lines 2 to 6.

Line2 Here we use And_Intro to replace the current goal (((p$\wedge$(q$\rightarrow$r)) $\rightarrow$s)$\wedge$(((q$\wedge$s)$\wedge$t$\rightarrow$p)) by 2 sub-goals (((p$\wedge$(q$\rightarrow$r)) $\rightarrow$s) and (((q$\wedge$s)$\wedge$t$\rightarrow$p)).

Line3 Similarity, the Impl_Intro rule replaces the current goal (((p$\wedge$(q$\rightarrow$r)) $\rightarrow$s) by s, and adds a new assumption H0: p $\wedge$ (q $\rightarrow$ r).

Line4 The code 'And_Elim_all in H' recursively apply the $\wedge$ E1 and $\wedge$ E2 in the current assumption H: $\vdash$ (((p$\rightarrow$q$\vee$s)) $\wedge$ ((p$\wedge$r$\rightarrow$s)) $\wedge$ ((s$\wedge$t) $\rightarrow$ (p$\vee\neg$q))), and replace it with the two assumption H1:s $\wedge$ t $\rightarrow$ p $\vee\neg$ q and H2: p $\wedge$ r $\rightarrow$ s.

Line5 Similarity, we replace the current assumption H0: p $\wedge$ (q $\rightarrow$ r) to two assumptions H0: p and H3: q $\rightarrow$ r.

Line6 By the Impl_Elim rule, we apply to the two assumptions of the form H :p $\rightarrow$ q $\vee$ s and H0 : p and add the new assumption H5: q $\vee$ s.

- The elimination rule forward, work on assumptions. Since we have done all the reprocessing of our assumptions, and our current formula contains only one propositional variable, we are beginning to apply the elimination rules backwards.

Line7 The code 'Or_Elim in H5', applies to an assumption of the form H5 : q s. It generates two proof obligations with assumptions H4: q resp. H4: s and the current goal.

Line8 Add a new assumption H5:(p $\wedge$ r).

Line9 Similarly, since assumptions H2 : p $\wedge$ r $\rightarrow$ s and H5 : p $\wedge$ r already have, apply the Impl_Elim rule and a new assumption H7: s is generated.

Line10 This branch has been proven, we use the 'assumption.' close our subgoal s.

3.3 Proof Generated by Our Method

Here we will use the same formula to show the proof tree generated by our method, which is more readable and imitates the human reasoning process.

Example 7. Given a derivation $\vdash (((p\rightarrow q\vee s)) \wedge ((p\wedge r\rightarrow s)) \wedge ((s\wedge t) \rightarrow (p\vee\neg q))) \rightarrow (((p\wedge(q\rightarrow r)) \rightarrow s)\wedge(((q\wedge s)\wedge t\rightarrow p))$, and show there is a proof using natural deduction.

The Branch A:

$$\dfrac{\dfrac{\dfrac{[p \wedge (q \to r)]^2}{p}\wedge E1 \qquad \dfrac{[q]^1 \qquad \dfrac{[p \wedge (q \to r)]^2}{q \to r}\wedge E2}{r}\to E}{p \wedge r}\wedge I \qquad \dfrac{\dfrac{[((p \to q \wedge s) \wedge (p \wedge r \to s)) \wedge (s \wedge t \to p \vee \neg q)]^5}{(p \to q \vee s) \wedge (p \wedge r \to s)}\wedge E2}{p \wedge r \to s}\wedge E2}{s}\to E$$

The Left Branch:

$$\dfrac{\dfrac{\dfrac{\dfrac{[p \wedge (q \to r)]^2}{p}\wedge E1 \qquad \dfrac{\dfrac{[((p \to q \vee s) \wedge (p \wedge r \to s) \wedge (s \wedge t \to p \vee \neg q))]^5}{(p \to q \vee s) \wedge (p \wedge r \to s)}\wedge E1}{p \to q \vee s}\wedge E1}{q \vee s\text{[b]}}\to E \qquad \begin{matrix}\text{Branch A}\\ \vdots \\ s\end{matrix} \qquad [s]^1}{s}\vee E^1}{p \wedge (q \to r) \to s}\to I^2$$

$$\dfrac{\dfrac{\begin{matrix}\text{Left Branch}\\ \vdots \\ p \wedge (q \to r) \to s\end{matrix} \qquad \begin{matrix}\text{Right Branch}\\ \vdots \\ (q \wedge s) \wedge t \to p\end{matrix}}{(p \wedge (q \to r) \to s) \wedge ((q \wedge s) \wedge t \to p)}\wedge I}{(p \to q \vee s) \wedge (p \wedge r \to s) \wedge (s \wedge t \to p \vee \neg q) \to (p \wedge (q \to r) \to s) \wedge ((q \wedge s) \wedge t \to p)}\to I^5$$

The other branch is generated analogously.

[b]Here is the left-most branch of this proof-tree.

Recursively generate the proof tree

Here we will demonstrate how our algorithm is used to generate our proof tree recursively. As the methods for generating proof trees are essentially the same, we will only demonstrate the leftmost branch's generation process here, which is the first branch generated by the algorithm.

assumptions, **formula** $((p\rightarrow q \lor s)\land(p\land r\rightarrow s))\land(s\land t\rightarrow p\land \neg q)\rightarrow (p\land (q\rightarrow r)\rightarrow s)\land ((q\land s)\land t\rightarrow p)$
do $\rightarrow$ **I** $((p \land (q \rightarrow r) \rightarrow s) \land ((q \land s) \land t \rightarrow p)$, add new assumption: $((p \rightarrow q \lor s) \land (p \land r \rightarrow s)) \land (s \land t \rightarrow p \lor \neg q)$
do assumptions $\land$ **E1**: $(p\rightarrow q\lor s)\land (p\land r\rightarrow s)$ and $\land$ **E2** $s\land t\rightarrow p\lor \neg q$
do assumptions $\land$ **E1**: $p\rightarrow q \land s$ and $\land$ **E2** $p\land r\rightarrow s$

Our recursion starts with an empty assumption list. In the recursive process, assumptions may be added to the list based on the rule applied. The formula parameter is the formula we wish to prove. At the beginning, it is a formula entered by the user. First, we apply the $\rightarrow$ I rule, and add an assumption from the left-hand side of our current conclusion. We apply this rule because our strategies always want to start with the 'Introduction rules backwards', and this is a binary formula connected by an implication rule. Then we recursively generate the sub-proofs, which are from the right-hand side of our current conclusion.
At the beginning of the next recursion, we have an assumption which structure as $((\varphi_1 \land \varphi_2) \land (\varphi_3))$, we repeatedly apply the $\land$E1 and $\land$E2, to simply the assumption until they are no more change. After that, it generates 3 different assumptions φ_1, φ_2, and φ_3. They are natural deduction proofs composed of corresponding rules and have corresponding sub-proofs above the line.

assumptions: $p\rightarrow q\lor s$, $p\land r\rightarrow s$, $s \land t\rightarrow p\lor \neg q$
formulas: $(p\land (q\rightarrow r)\rightarrow s)\land ((q\land s)\land t\rightarrow p)$
do $\land$**I** left $p\land(q\rightarrow r)\rightarrow s$ right $(q\land s)\land t\rightarrow p$

Then we work on the sub-proof generated from the rule $\rightarrow$ I. Here we have the current formula with the structure $\varphi_1 \land \varphi_2$, we apply the $\land$I rule. The rule recursively generates 2 sub-proofs φ_1 and φ_2 above the line.

assumptions: p→ q∧ s, p∧ r→ s, s∧ t→ p∨ ¬ q
formula: p∧(q→ r)→ s
do → **I** s, add new assumption: p∧(q→ r)
do assumptions ∧ **E1** p and ∧ **E2** q→ r, do→ **E** p.

Now we follow the left-most sub-proof generated from the rule → I. Similarly, we apply the → I rule and generate a new assumption. The rule recursively generates a sub-proof from the right-hand side of the current conclusion. As we have a new assumption formed with $\varphi_1 \wedge \varphi_2$, we apply the ∧E1 and ∧E2 make them become φ_1 and φ_2.
It is imperative to note the next step: Here we have a new assumption p, and we have another assumption in our list (p→ q ∨ s), which was generated at the beginning. We apply the → E rule, and here is the new assumption generated. We omit other components of sub-proofs.

$$\dfrac{\overset{\vdots}{p} \quad \overset{\vdots}{p \to q \vee s}}{q \vee s} \to E$$

formula: s
try ∨ **E**, add new assumption a: q, and find the subProof q.

Now we work on the sub-proof generated by the → I. Here we only have a single propositional variable as our formula, we cannot use the 'Introduction Rule Backward' anymore, so we start the 'Elimination Rule Forward', and try the ∨ E rule. The assumption we just generated using the → E rule can be used here, we can put it on the first branch above the ∨ E rule. Thus, we have located the leftmost branch of the proof tree.
We are looking for the second subproof above the ∨ E. We first perform → E based on our assumptions, we have:

$$\dfrac{\overset{\vdots}{p \wedge r} \quad \overset{\vdots}{p \wedge r \to s}}{s} \to E$$

Then we find our subProof above the formula s. Similarly, as we have q → r in our assumption list, we have:

$$\dfrac{\overset{\vdots}{q} \quad \overset{\vdots}{q \to r}}{r} \to E$$

And this will be added to our assumption list.

```
do → E p∧ r
do →E q

assumptions: q∨ s, s∧ t→ p∧ ¬ q, p, r, q,
formula: p∧ r
do ∧ I left p right r
do → E q

assumptions: q∨ s, s, s∧ t→ p∧ ¬q, p, r, q,
formula: s
∨ E: add new assumption b: s, and finding the subProof b of ∨E
find ∨E: q∨s,s,s
```

For the second branch above the → E, we have the formula p ∧ r. We apply the ∧ I and generate 2 sub-proofs p and r. Then we apply the → E to generate a proof with the conclusion of p again. However, as we already have the same proof in our assumption list, this proof will not be added to the list again.
Then we are looking at the third branch above the → E. As we find the formula s in our assumption, we find the sub-proof and terminate the recursion. The right branch is generated analogously.

Chapter 4

Implementation

In this chapter, we want to outline our implementation. The source code can be found at [5].

4.1 Architecture of “Formula” and “NDProof”

When we think about implementation, we always hope that our program architecture will become obvious - straightforward to implement, read, maintain, and use. In this section, we’ll go over how to use the composite design pattern to achieve our primary goal of making the implementation’s architecture straightforward.

4.1.1 Composite Design Patterns

Main Benefits

The compose pattern is suitable for any tree structure to make everything straightforward. As objects can represent part-whole hierarchies, clients can treat all objects in the composite structure uniformly [4]. In other words, it is easier to implement and use, as it eliminates redundant code for different objects, and the complex structure of the entire tree is hidden.
For the 1st point, the composite mode eliminates the repetitive code written for different types of children, which only requires the component to be written. For the other advantage, as the composite mode hides the tree structure, the client only needs to find the component.
In our implementation, we will apply the composite design pattern in classes “Formula” and “NDProof”.

$(p \vee (q \to (\neg p)))$
$(q \to (\neg p))$
$(\neg p)$

Table 4.1: The Recursion Structure of the the formula $(p \vee (q \to (\neg p)))$.

Formula: Motivation and Applicability

A well-formed formula[a] can be represented as a parse tree [6]. For example, the formula $(p \vee (q \to (\neg p)))$ can be represented as the following tree structure in the figure 4.1.
As we can see from this tree 4.1, the root is the formula, on each layer of the tree, we have a smaller sub-formula. We can give relations between those sub-formulas and formulas by the definition 1 [9] [10].

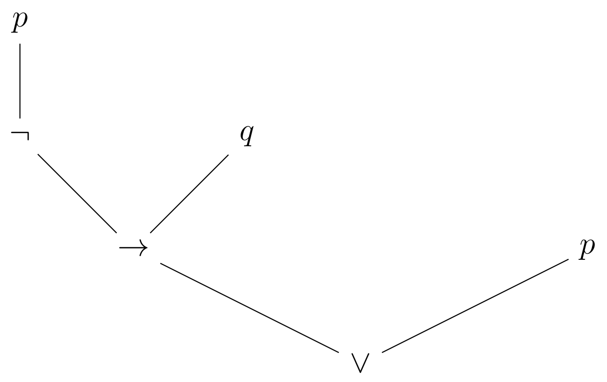


Figure 4.1: The Tree Structure of the the formula $(p \vee (q \to (\neg p)))$.

Formula: Structure and Participants

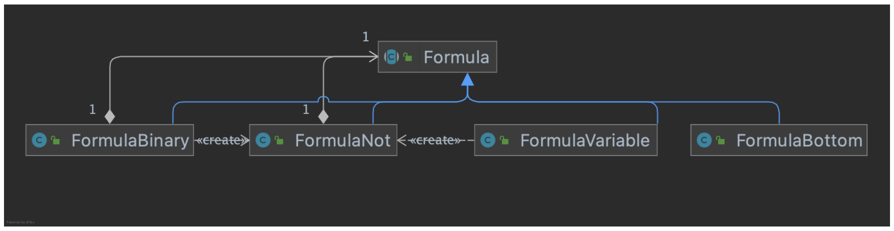


Figure 4.2: Structure of “Formulas”

[a]It will be denoted as $w.f.f.$

Component: *Formulas*
Leaf: *FormulaVariables*, *FormulaBottom*
Composite: *FormulaBinary*, *FormulaNot*
Mode: safe mode
As we already know, a formula is a tree structure. Here we have an abstract class, *Formula*[b], would serve as the superclass of 4 extending classes: *FormulaVariable*, *FormulaNot*, *FormulaBottom* and the *FormulaBinary*, which represent four situations in our definition 1:

$\psi(\varphi) = \{\varphi\}$	$\varphi = p$	FormulaVariable
	$\varphi = \bot$	FormulaBottum

Table 4.2: "Formula" Base Cases

$\psi(\neg\varphi) = \psi(\varphi) \cup \{\neg\varphi\}$	$\varphi = \neg\varphi$	FormulaNot
	$\varphi = \varphi_1 \wedge \varphi_2$	
$\psi(\varphi_1 \bigtriangleup \varphi_2) = \psi(\varphi_1) \cup \psi(\varphi_2) \cup \{\varphi_1 \cup \varphi_2\}$	$\varphi = \varphi_1 \vee \varphi_2$	FormulaBinary
$\bigtriangleup \in \{\wedge, \vee, \rightarrow,\}$	$\varphi = \varphi_1 \rightarrow \varphi_2$	

Table 4.3: "Formula" Recursive Cases

Classes of "Formulas"

In the abstract class **Formula**, we have the following functions and methods:
toStringPrecedence[c]: It implemented in every subclass of Formula is determined by the precedence of operators. (*int*) is the operator's precedence.
isValidFor: It is implemented in every subclass of Formula, determining whether the formula is valid for the specific *truthAssignmemt*, which respective truth values (true or false).
isValid: Check the *truthAssignment* in the formula to see if it is generally true, i.e. $p \vee \neg p$, return true.
createProofTA: We will discuss this in a later section.
getPropVariables: It returns a set of strings so that variables that occur more than once in a formula only appear once in the result.

[b]Formula is an abstract class, which is shown by the icon in front of it.
[c]It is an abstract method, which is shown by the icon in front of it.

For the abstract class formula, it has four extending classes: $FormulaVariable$, $FormulaButtom$, $FormulaNot$, and $FormulaBinary$.

- **FormulaVariable**: This class extends the formula to just a variable, which is a string. An element of this class would represent atomic propositions, i.e. p, q $\cdots$. The constructor signs the inner member $value$ as the string. As we represent an atomic proposition, we only have one inner member.
- **FormulaButtom**: The base case $FormulaButtom$ is a special formula, which is only a symbol of the bottom, written as $\perp$.
- **FormulaNot**: The $FormulaNot$ is not for another formula, e.g. a formula starting with not, and there is another formula in there. It has 1 component which is also the formula. The constructor signs the parameter from the $bodyFormula$, to the Formula.
- **FormulaBinary**: It represents the most common cases: 2 formulas are connected by the binary operation. The constructor has two sub-formulas, $LeftFormula$, $RightFormula$, and $operation$, which connect them.

NDProofs: Structure and Participants

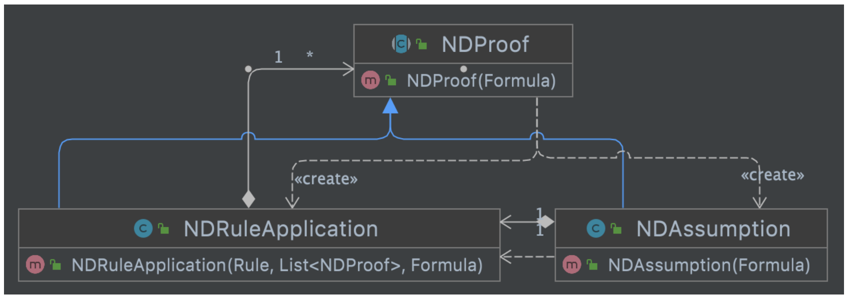


Figure 4.3: Structure of “NDProofs”

Component: $NDProof$
Leaf: $NDAssumption$
Composite: $NDRuleApplication$
Mode: **safe mode**

Classes of "NDProofs"

The abstract class **NDProof** represents a natural deduction proof. It contains one variable: *Formula conclusion*, which represents the conclusion of the current tree. The constructor gets the *conclusion* passed in and stores the *conclusion*.
getConclusion: It is simply the getter return the *conclusion*.
discardAll: This method is for discarding all assumptions in the NDProof. We will explain this in detail in the following chapters.
discard: *discardAll*() calls *discard*() will actually do the work. Also, we will explain this in detail in later chapters.

NDProof has two extending classes: *NDAssumption* and *NDApplication*.

- **NDAssumption** extends the *NDProof* and represents the assumption of the natural deduction proof. It contains an additional boolean variable *isDiscarde*. The constructor calls the super constructor and sets the variable *isDiscarded* to false.
- **NDApplication** extends the *NDProof* and represents the application of the natural deduction rule. It contains 2 variables, *subProofs* and *rule*.

4.2 Proof Tree Generation and Discard

4.2.1 Generates proofs automatically using the new algorithm

In our implementation, we will use *createProofTree*() to generate the proof tree recursively.

Algorithm

We have 2 parameters in this method, assumptions and formula. The parameter **assumptions** is what we need to build the proof tree. It was empty at the beginning and will be generated later when we apply certain natural deduction rules. The parameter **formula** is the formula that we want to prove. It is initially generated by the parser[d]. If we find the formula in the assumption list, we terminate the recursion and return the proof tree.

[d]This program is written in Java using *JParsec*. By importing *JParsec*, we can use the *FormulaParser* to parse user-input formulas.

```
Data: List of assumptions $\alpha$, Formula formula $\varphi$
Result: NDProof proofTree
/* Use $\neg$ E$^*$, $\rightarrow$ E$^*$, and $\vee$ I$^*$, if both $\alpha$ and $\varphi$ = T. Use PBC$^{**}$ if $\alpha$ =
   F.                                                                          */
simplifyAssumptions ($\alpha$);/* We repeatedly do $\wedge$ E there are no more changes
 in $\alpha$.                                                                  */
 do $\rightarrow$ E*
 simplifyAssumptions ($\alpha$);
 if the $\varphi$ is in the $\alpha$ list then return proofTree; /* find the tree */
if $\varphi$ is ($\varphi_1$ operation $\varphi_2$) then
    switch operation do
        case $\rightarrow$ do proofTree = $\rightarrow$I ;
        case $\wedge$ do proofTree = $\wedge$I;
        case $\vee$ do proofTree will try $\vee$I*, $\vee$ E, $\rightarrow$E$^*$, and PBC$^{**}$;
    end
end
else if $\varphi$ is $\neg\varphi'$ then proofTree = $\neg$ I;
else if $\varphi$ is $\varphi'$ then
    proofTree will try $\vee$ E, $\rightarrow$E$^*$, and PBC$^{**}$
    else /* when $\varphi$ is $\perp$                                           */
        proofTree will try $\neg$E*, $\vee$ E
    end
end
```

Strategies

At the beginning of each recursion, we do some pre-processing in our assumptions:

- Make assumptions smaller/simpler. We achieve this by $simplifyAssumptions(\alpha)$. As long as there is an assumption with structure $\varphi_1 \wedge \varphi_2$, we repeatedly do $\wedge$ E1, $\wedge$ E2 until there are no more changes in this assumption.

- Add more information to our assumptions. When we use $\wedge$ E and $\rightarrow$ E, assumptions add more information on its sub-proofs.

Then we select the appropriate rule to apply based on the formula's structure. Under one formula structure, we may have multiple rules to choose from at times. Similar to the example in Chapter 2, we apply the following strategy to model the human reasoning step.

1. Introduction Rule Backward, work on the conclusion.
2. Elimination Rule Forward, work on assumptions.
3. Try a Proof By Contradiction if a direct proof doesn't work.

Other strategies to help us terminate the program:

- In rules $\neg$ E, $\rightarrow$ E and $\vee$ I, we check whether the assumption is true and whether the formula follows. We cannot apply the rule if this is not the case.
- We apply the PBC rule only if $\perp$ is an assumption.
- Avoid adding two identical assumptions to the list.

Complexity Analysis

The complexity of this method is exponential. We will iterate over almost every scenario when generating the proof tree, and we cannot guarantee the depth of the recursion. Also, this is an NP-complete problem. Although our algorithm is not better in run-time than others, our goal is to generate a readable proof tree.

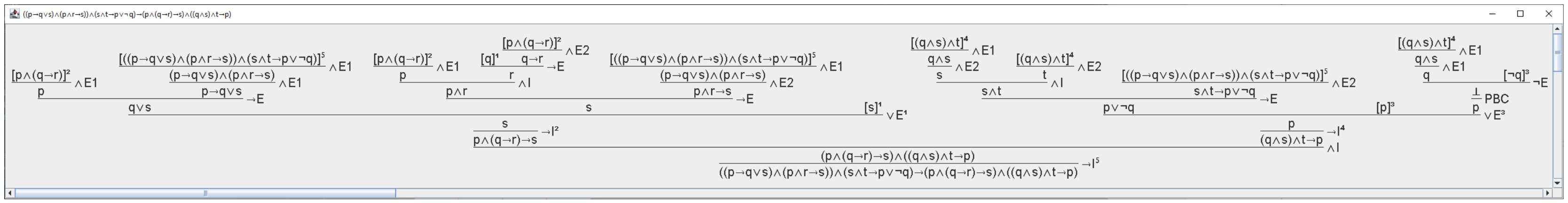

((p→q∨s)∧(p∧r→s))∧(s∧t→p∨¬q)→(p∧(q→r)→s)∧((q∧s)∧t→p)
(p∧(q→r)→s)∧((q∧s)∧t→p)
((p→q∨s)∧(p∧r→s))∧(s∧t→p∨¬q)→(p∧(q→r)→s)∧((q∧s)∧t→p)

Figure 4.4: A readable proof tree generated by our algorithm: $\vdash (((p{\to}q{\vee}s)) \wedge ((p{\wedge}r{\to}s)) \wedge ((s{\wedge}t) \to (p{\vee}\neg q))) \to (((p{\wedge}(q{\to}r)) \to s) \wedge (((q{\wedge}s){\wedge}t{\to}p))$

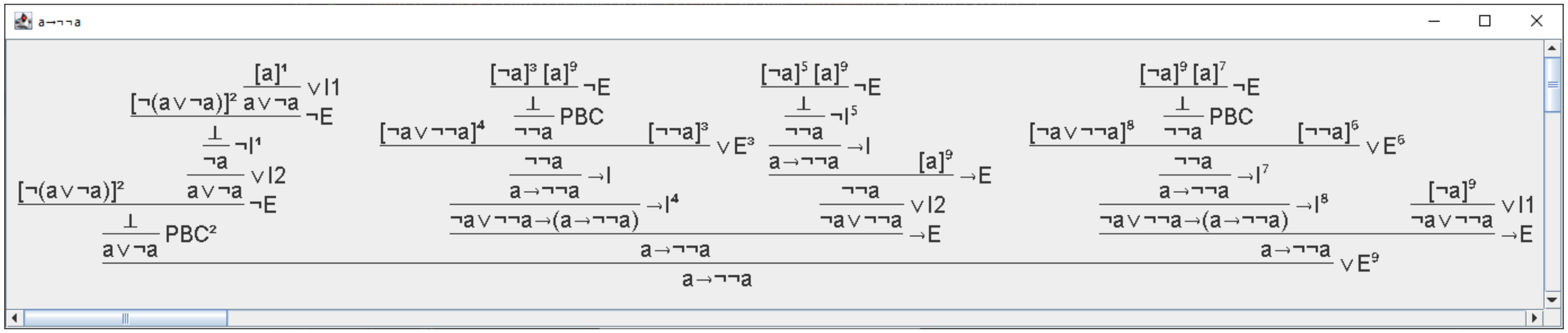


Figure 4.5: The tree for the derivation $\vdash$ a $\to$ ¬¬ a is produced by lemma 1.2.3

Discard Assumptions

We need assumptions to help us complete the proof. When we don't need the assumptions, we need to discard them. In the abstract class ***NDProof***, the method

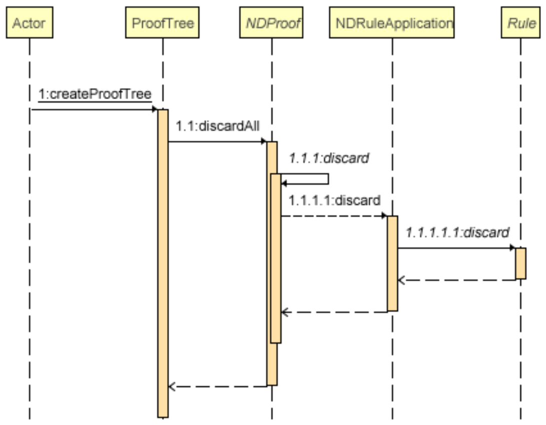


Figure 4.6: The Sequence Diagram of Discard Process of derivation $a \vee b \rightarrow b \vee a$

$discardAll()$ is the method that we will call in order to discard all assumptions in a $NDProof$ according to the rules. The method calls $discard()$ which will actually do the work. $discard()$ gets a $DiscardLabelFactory$ for creating labels, a list of formulas that will be discarded, and a list of $NDProofRules$ which are the rules that discard the assumption. i.e. $formulas.get(i)$ is discarded by rule $discarders.get(i)$. Initially, these lists were empty.
In the class of $NDAssumption$ and $NURuleApplication$, we have implementations of discard():

- In ***NDRuleApplication*** we simply call the method discard of Rule (we will discuss it soon) and store the $DiscardLabel$ that is returned in a variable.

- In ***NDAssumption*** we need to check whether the conclusion is in the list formulas. If the assumption is discarded, we store the $discarder.get(i)$ in a variable of the class.

In ***Rule***, we implemented $DiscardLabeldiscard()$. For rules that do not discard any assumption we simply call discard recursively for each $subProof$ of proof and return null as the $DiscardLabel$. These recursive calls will modify the two lists, so we make a new list each time to ensure that each call utilizes the initial list. For the other rules, we create a new DiscardLabel and add the formulas that need to be discarded in the corresponding sub-proof to the list. The $DiscardLabel$ has one private variable label of type int. This is the label that indicates by which rule an assumption is discarded.

Then we add proof to discarders in the corresponding subProof in which the formula needs to be discarded, and then we do the recursive calls as above.

4.2.2 Generates proof automatically using the completeness theorem

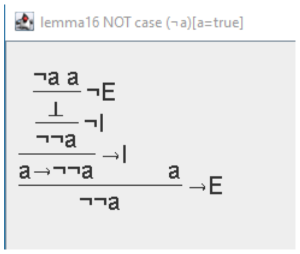
Induction on the structure of formula

In the class Formula, we have an abstract method *createProofTA*. In this section, we will discuss implementing this in the method in each subclass of the formula according to the lemma of completeness.
The lemma says that there is a proof of the formula if the formula is true for the given truth assignment and a proof of not of the formula if the formula is false. In either case, the proof might have open assumptions that are propositional variables resp.[e] not of such a variable depending on whether the truth assignment assigns truth resp. false to the variable. In general, we have two base cases $\varphi = p, \bot$, and four recursion cases $\varphi = \neg\varphi, \varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \rightarrow \varphi_2$.
$\varphi = p$: We have this case in the class *FormulaVariable*. Since this is a propositional variable, the proof is just an assumption p or $\neg p$ depending on whether the *truthAssinment* is true or false for p. As it is, we call the *isValidFor* to get the '*truthAssignment* of these propositional variables, and return "p" or "$\neg p$" in each case.
$\varphi = \bot$: We have implemented this case in the class *FormulaBottom*. We make the $\bot$ as the assumption and produce the $\neg I$.
$\varphi = \neg\varphi$: We have this case in the class *FormulaNot*. According to the Lemma, the propositional variables in φ' are those from φ [10], we distinguish two cases here:

- If $\overline{v}(\varphi) = T$, then $\overline{v}(\varphi') = F$: If the formula $\neg p$ is true for the *truthAssignment*, then we call the method recursively for the body p and the proof returned is the result.

- If $\overline{v}(\varphi) = F$, then $\overline{v}(\varphi') = T$: If the formula is false, then we call the method recursively for the body, create a proof using the Lemma for the body p of the formula, and then we apply to those two proofs as an $\rightarrow E$. Regarding this situation, the output results are shown in Figure 4.7.

[e]The abbreviation"resp" stands for"respectively" It is frequently used to indicate that two lists contain the same items in the same order.


lemma16 NOT case (¬a)[a=true]

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg a \quad a}{\bot}\neg E}{\neg\neg a}\neg I}{a\to\neg\neg a}\to I \qquad a}{\neg\neg a}\to E$$

Figure 4.7: The output of case $\varphi = \neg\varphi$ when $\overline{v}(\varphi') = T$

We have also implemented case $\varphi = \varphi_1 \wedge \varphi_2$, $\varphi = \varphi_1 \vee \varphi_2$, and $\varphi = \varphi_1 \to \varphi_2$ followed by the lemma within the class $FormulaBinary$. The process of proving is similar with the $formulaNot$.

Combining True Trees

The Fig 4.5 shows the derivation $\vdash$ a $\to \neg\neg$ a has a tree generated by the lemma 1.2.3. As we only have 1 propositional variable a, and generated 2 rows of true tables. In this case, we use one $\vee$ E rule to combine our tree. In our example 6, we have 5 propositional variables and generated 2^5 rows of the truth table. In that case, we can make a big proof with 582 discard labels and 31 natural deduction proofs used with the $\vee$ E rule.

4.3 Program Usage

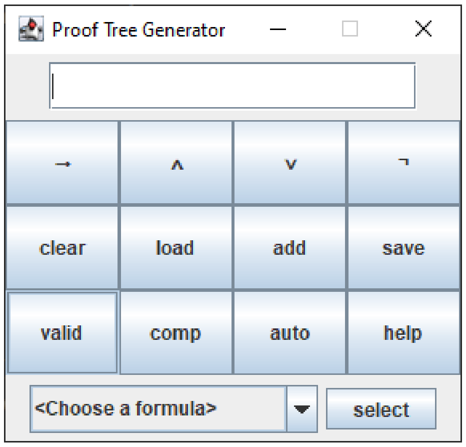


Figure 4.8: The User Interface of the Program

The user interface is able to do the following:

- The dropdown button: There is be a list of formulas in the system from which the user can select a formula.
- The upper input text box: A user can enter a formula, after entering the formula it will be added to the list.
- The save, load, and button: A formula can also be saved, loaded, or added from a text file.
- After selecting a formula in the list the user can do the following:
  - The valid button: Check whether the selected formula is valid.
  - The comp button: Generate and display the proof generated by completeness proof if the formula is valid.
  - The auto button: Generate and display the proof generated by the new method if the formula is valid.
  - A message will appear if the formula is not valid.

Chapter 5

Conclusion and Future Work

At the beginning of this thesis, we aimed for creating a tool for computer science professionals that can address the issue that the argumentation process in the computing research field related to propositional logic used to be complicated and unreadable. We achieved the goal by implementing an automatic theorem prover, which is achieved by generating a proof tree recursively, using the corresponding natural deduction proof either forward or backward, depending on the structure of the formula. In comparison to other tools that generate propositional logic proofs automatically through natural deduction, the proof tree generated by our method is significantly concise and human-readable.
We would like to outline several topics that can be addressed in future work.

1. We can explore in future research transforming the proof tree we generated into natural language sentences. For example, if the proof tree ends with the implication introduction rule and the conclusion $\varphi \to \psi$, this could be translated into the following English text: In order to proof $\varphi \to \psi$, we assume φ and $\cdots$ conclude ψ, where the dots serve as a place holder for the translation of the proof tree above the implication introduction into natural language, and then prove ψ [6].

2. The user interface could be improved. First, a menu item for displaying the truth table could be added. Second, we could allow the user to select one row from the truth table in order to generate the proof for that row of the table using Lemma 1.2.3. In addition, we could let the user manipulate proof trees by hand for even further improving them or produce a proof completely by hand.

3. We could find and add a procedure for converting Prover9 proofs automatically into Natural Deduction. This will allow a user to import such proof.

4. We can consider this method to automatically generate human-readable proofs in first-order logic. The problem we need to overcome is that this logic is not decidable, so the algorithm will not terminate if the formula is false. However, if the formula is actually true, then it should be possible to produce a human-readable proof.

Bibliography

[1] Carmen Brun, Jonathan Buss, Lila Kari, and Anna Lubiw. Cs 245: Logic and computation [lecture slides], 2018. `https://cs.uwaterloo.ca/~cbruni/CS245Resources/lectures/2018_Fall/01_Introduction_post.pdf`.

[2] Stanley Burris. *Logic for Mathematics and Computer Science*. Prentice Hall, Upper Saddle River, NJ, c1998.

[3] Defense Advanced Research Projects Agency (DARPA). Explainable artificial intelligence (xai). Technical report, 2016. Defense Advanced Research Projects Agency.

[4] Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, 1995.

[5] Maka Hu. Automatic-generation-of-human-readable-proofs, 2023. `https://github.com/MakaHu/Automatic-Generation-of-Human-Readable-Proofs`.

[6] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[7] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[8] The Coq Development Team. The coq proof assistant, 2023. Version 8.14.0.

[9] Dirk van Dalen. *Logic and Structure*. Springer, 4th edition, 2004.

[10] M. Winter. Logic in computer science. `https://www.cosc.brocku.ca/~mwinter/Courses/5P02/Logic.pdf`, 2014.