LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

CHAIR OF THEORETICAL COMPUTER SCIENCE AND THEOREM PROVING

Resolution Prover in Rust

Wanda Rosmus

Bachelor's Thesis in Computer Science

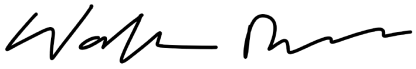
Supervisor: Prof. Dr. Jasmin Blanchette

Advisor: Lydia Kondylidou

Submission Date: 01.08.2024

Disclaimer

I confirm that this thesis type is my own work and I have documented all sources and material used.



Munich, August 1, 2024 Wanda Rosmus

Acknowledgments

First of all, I would like to thank my advisor Lydia Kondylidou, who, despite her own doctorate and other university commitments, still found time for my work, was always patient, gave me helpful feedback and always had an open ear for challenges and problems. Thank you for the opportunity to familiarize myself with this fascinating topic under your guidance! I would also like to thank my family and friends who have constantly supported me.

Abstract

Resolution is a theorem proving technique that uses proof by contradiction to determine the satisfiability of logical formulas. Resolution is often used in automated theorem proving systems, which help to automatically prove mathematical theorems using computer programs. As part of this thesis, I developed a resolution prover for first order logic inputs in the Rust language, which has not existed before. First, a resolution prover for propositional logic inputs was developed, which then served as the basis for the development of a resolution prover for first order logic inputs. In this bachelor thesis, I describe the theoretical foundations and the implementations made in order to develop a resolution prover in Rust for first order logic inputs.

Keywords: Resolution Prover, Rust, First Order Logic, FOL, Propositional Logic, Automated Theorem Proving, Theorem Prover, Unification, CNF, CNF Conversion.

Contents

1 Introduction

1.1 Motivation

In our data-driven and highly automated world, it is crucial to understand the underlying decision-making mechanisms and processes used in automated logical reasoning. One example for such mechanism is resolution, which plays a central role in formal logic and theorem proving. In this thesis paper, I will explain the development of a resolution prover for first-order logic (FOL) and propositional logic inputs and emphasize to the importance of such provers. Before I explain how I created my resolution prover and also how it works I generally want to outline the importance of automated theorem provers and how a resolution prover fits in this topic.

Automated theorem proving (ATP) is a computer-based technique that is used to automatically prove mathematical theorems. Automated theorem proving is based on formal logic and reasoning mechanisms that aim to prove (or disprove) the validity of a specific input, e.g. a mathematical thesis. This involves demonstrating how an assertion (the theorem to be proven) can be logically derived from a set of assumptions or axioms. The practical applications of automated theorem provers are incredibly diverse, as they help to ensure the correctness of (mathematical) systems. A typical example of a practical use of automated theorem provers is the verification of software. There, theorem provers are used to check whether a software program meets certain specifications like correctness or also security requirements. To do this, the actual functionalities and structure of the software must first be formalized, as well as the behavior or properties to be checked. A well-known example of the use of ATP for the verification of software is the Z3 Theorem Prover, which was developed by Microsoft Research and is used in particular to check and guarantee the security of code in Windows operating systems (Microsoft 2024). Automated theorem provers are therefore used in many different ways and can vary greatly in their complexity, whether they are used to prove mathematical theorems in research or in industry to ensure the security of code.

When executing an automated theorem prover, various algorithms and functions are used to control the logical closure of the proof, depending on the requirements and available input. Resolution is a procedure based on the concept of proof of contradiction which is also frequently used in this context since it is a specific method in the field of automated theorem proving. The basic idea of the resolution principle is look for contradictions from given logical statements by using a systematic procedure and thereby to prove the satisfiability or unsatisfiability of a conjecture. In this thesis I have focused on designing and developing an automated theorem prover based on resolution or to be more specific a resolution prover. A Resolution Prover (Resolution Theorem Prover) is an integral tool within computational logic. The primary task of such a prover is to validate or falsify conjectures within a logical framework by executing automated deductive reasoning processes. Resolution provers are crucial in various areas, including theorem proving and formal verification. In theorem proving a resolution prover can assist in automatically proving mathematical theorems by deriving logical consequences from given inputs, e.g. axioms and hypotheses. This form of application is of particular importance in both computer science and mathematics, as in these fields formal proofs ensure the correctness of algorithms and systems.

1.2 The Rust Programming Language

To implement the resolution prover, my supervisor and I selected the Rust programming language (Official Website) because it excels in performance, safety and concurrency. Rust's unique ownership system guarantees memory safety while maintaining high execution speed, making it an ideal choice for developing robust and efficient tools for logical reasoning (Santoso2023). Or in this case an ideal choice for writing a resolution prover.

Rust offers several technical advantages over other languages traditionally used for prover development. You could say that Rust is an answer to a few well-known problems of other popular languages such as null pointer dereferencing and memory safety. Rust addresses these issues with its ownership system, which ensures that memory safety is enforced at compile time (Santoso2023) without the need for a garbage collector (automatic memory management tool to identify and free memory resources that are no longer needed). In addition, Rust's ownership system directly releases memory that is no longer required. This concept of direct memory release is possible due to the use of scopes, or more precisely the memory release at the end of a validity range (Klabnik2023) . This avoids a whole class of common runtime errors. In general, a garbage collector can affect the entire program despite its useful activities due to the use of additional computing power. The unpredictable pauses and the computing power required can impair performance and responsiveness. Rust's system avoids this problem, resulting in a way more predictable performance, which is critical for computational tasks that require high efficiency and low latency, such as theorem proving.

Another advantage of Rust would be its concurrency model, supported by its ownership and borrowing rules which ensures that data races are prevented at compile time (Santoso2023). To be more specific that the compiler checks and guarantees that there will be no conflicting inquiries and executions to data from multiple threads, which enhances both safety and performance. This makes Rust highly efficient for multi-threaded and performance-critical applications.

I have also noticed that the Rust compiler provides exceptionally informative error messages and warnings. These messages guide developers to fix problems quickly and effectively, which is especially beneficial for new programmers or those switching from other languages. The compiler not only identifies errors and warnings, but also offers suggestions on how to fix them, which optimizes development from my personal experience and also the concrete comparison with the error messages of other languages I have previously programmed in.

The benefits of Rust go beyond its technical features. The language is especially supported by an active community of developers who are committed to its continuous improvement. This community is particularly known for its inclusivity and collaborative spirit, which results directly in the provision of extensive documentation and strong support for newcomers. (E.g. the website Learn Rust which contains more than enough material to learn rust. I also used it to learn this programming language. What really stood out for me was that it is also possible to solve small programming tasks interactively in the browser. This has made learning the language quite uncomplicated and flexible.) Since its establishment in 2021, the Rust Foundation has been crucial in overseeing Rust's development and maintenance, (Website Rust Foundation). In addition, Rust benefits from a rapidly growing ecosystem of libraries and tools, including crates (package manager in rust), that cover a wide range of programming requests and needs. This extensive ecosystem facilitates enhanced development and provides reliable, pre-built solutions and libraries for common tasks, making Rust not only a powerful language, but also an accessible language for developers or people taking their first steps in software.

1.3 Objective of the thesis

The primary goal of this thesis is to develop a Resolution Prover in Rust specifically designed for handling inputs in first-order logic (FOL). I approached this goal by first developing a resolution prover for propositional logic inputs. This first step was very helpful in understanding and familiarizing myself with the topic of automated logical reasoning and at the same time learning the programming language Rust, which was new to me. While propositional logic served as the starting point, the main focus and ultimate goal of this thesis is the implementation of a resolution prover that is able to effectively manage and derive conclusions from first-order logical inputs in Rust. By applying the resolution prover to a specific input, the user can determine whether this input is satisfiable or unsatisfiable. As an outlook, I also intend to make the resolution prover I wrote for propositional and first order logic inputs available to the Rust community and all interested individuals as an open source git repository once it is finalized. My contribution is intended to support future developments of Resolution Provers in Rust and to promote joint progress in the field of automated logical reasoning. It is important to note that my resolution prover can only evaluate a certain type of input. In particular, the FOL inputs, sourced from the TPTP (Thousands of Problems for Theorem Provers) website, can currently only be processed one specific type. The exact specifications of the inputs will be explained in a later part of the thesis. Nevertheless, it seems to me that my resolution prover is an important foundation for the development of further provers in Rust. By addressing this topic, I hope to improve the capabilities and applications of Resolution Provers in Rust so that a wide range of inputs can be processed in the future.

1.4 Overview of the structure

In this paper I will first give an overview of the theoretical foundations of propositional logic, the CNF form, first order logic and the resolution process, depending on the logic system of the inputs. This is the basis for the explanation of the concrete implementation of the resolution prover, which will be explained next. Here I differentiate between the resolution prover for propositional logic inputs and the resolution prover for first order logic inputs and explain the most important functionalities for both provers. A fairly large section is also dedicated to the implementation of a CNF converter, which was required for the resolution prover for first order logic. Finally, I will conclude the implementation of both provers with a detailed description of the resolution algorithm, emphasizing in particular the input logic-dependent extensions. I will then focus on evaluating the results using tests and statistics. I will end this work with a conclusion about the implementation and an outlook with ideas for the further development and optimization of the resolution prover.

2 Theoretical foundations

2.1 Propositional logic and CNF

In the initial phase of my work, I studied propositional logic and its efficient representation in Conjunctive Normal Form (CNF) thoroughly. Understanding these basic concepts was crucial, not only for the theoretical foundations, but also for structuring the programming framework and conceptualize the datastructure of the resolution prover that I developed. Propositional logic, a branch of logic, deals with propositions and their connections to other propositions. A proposition

is essentially a declarative statement that can be uniquely identified as true or false. For example, the statement "The current season is summer and the sun is shining" can be true or false depending on the actual weather conditions and the actual current season. In this example we have two propositions "the current season is summer" and "the sun is shining" which are connected with an AND-operator. This field of logic, which works in a binary framework in which each statement is strictly evaluated according to two possible truth values, true (T) or false (F), also provides a clear basis for the application of the logical operators described below (Rautenberg2010). With the help of these operators, such as conjunction, disjunction and negation, we can manipulate the truth values of propositions in order to construct new propositions from existing ones. Depending on the specific application of the operators, we can draw logical conclusions (Harrison2009). Propositions are represented by variables (e.g.P, Q) denoting simple statements. Now I will describe the listed operators in more detail using examples.

The negation operator reverses the truth value of a given proposition. For instance if the proposition P is "The current season is summer" then $\neg P$ would be "The current season is not summer".

A conjunction, on the other hand, is a linking of statements that uses the AND operator $\wedge$ and can only have the truth value true if all linked statements also have the truth value true. A simple example to illustrate this would be, for example, P corresponds to "I am travelling by train" and Q corresponds to "The train WiFi is working". Assuming that the two statements P and Q are correct, the conjunction of these statements $P \wedge Q$ "I am travelling by train and the train WiFi is working" would also be true. However, as soon as one part of the statement is no longer true, the conjunction is also no longer correct. The individual assignments of the truth values of two variables are also shown in Table 1 in the form of a truth table.

A disjunction is also a linking of statements but whereas to conjunction uses the OR operator $\vee$. In this case, the composite statement is true if at least one of the individual statements is true. To stick with the train analogy, let's consider the following example: Let P represent "The next train goes to Berlin," and let Q represent "The next train has available seats." The disjunction of these statements, $P \vee Q$, means "Either the next train goes to Berlin, or it has available seats, or both." This implies that as long as one of these conditions is met - either the destination is Berlin or there are seats available- you have a reason to consider taking the train.

P	Q	$\neg P$	$\neg Q$	$P \wedge Q$	$P \vee Q$	$\neg P \wedge Q$	$\neg Q \vee P$	$Q \Longrightarrow P$	$P \Longleftrightarrow Q$
T	T	F	F	T	T	F	T	T	T
T	F	F	T	F	T	F	T	T	F
F	T	T	F	F	T	T	F	F	F
F	F	T	T	F	F	F	T	T	T

Table 1: Truth Table propositional logic

Having covered the basic concepts of negation, conjunction and disjunction it is also important to understand implication and equivalence.

An implication can be understand as an if-then relationship between two propositions, e.g. $Q \implies P$ is read as "P implies Q". The implication is true, if Q is true or if both values are false. That is because something false can imply something true which makes the whole statement true. Also if something false implies something false it makes the whole statement true. The only case for an implication statement to be wrong is if something with the truth value true implies something false. Then the whole statement is false. An implication can also be represented by AND, OR and NOT operators, e.g. $Q \implies P$ is equivalent to $\neg Q \vee P$. This is really important for converting inputs which contain implications to a form with only NOT, OR and AND operators.

An equivalence is true if both compared propositions e.g. $Q \iff P$ have the same truth value, so both can be false or both can be true for the whole equivalence statement to be true. The statement $Q \iff P$ can also be understand as $Q \implies P$ and $P \implies Q$ and those to implications can be simplified even further to $\neg Q \vee P$ and $\neg P \vee Q$.

With keeping this brief introduction to propositional logic in mind I will now go on to explain the conjunctive normal form. A conjunctive normal form (CNF) is present when there is a conjunction of one or more clauses, where each clause consists of disjunctions of literals, e.g. $((Q \vee P) \wedge (Q \vee \neg P))$ is in CNF (Harrison2009). A Literal can be understand as an proposition or its negation. A CNF can also consist of only one clause with only one literal, e.g. $(\neg P)$ is also CNF because you could interpretate it as a conjunction of disjunctions, which is consistent with the definition of a CNF. Another special case would be a CNF consisting only of one disjunction, e.g. $(P \vee Q)$. The counterpart to CNF is the Disjunctive Normal Form (DNF) which is a disjunction of clauses that consists of conjunct literals, e.g. $((Q \wedge P) \vee (Q \wedge \neg P))$. A single clause with a single Literal can also be interpretated as an disjunction of conjunctions, which is the definition of a DNF. CNF is a form of representation for propositional logic inputs, because every statement in propositional logic can be expressed as a CNF. The CNF plays a central role in my resolution prover, as the resolution algorithm is specially designed to process inputs in CNF. Some of the inputs in my Resolution Prover are already in CNF format and if not, a CNF converter has been implemented to convert the inputs into CNF.

2.2 First Order Logic (FOL)

In the last chapter I described the structure of propositional logic which basically only consists of propositions and the operators NOT, AND, OR, IMPLIES, EQUIVALENT to build statements more complex structures of the propositions. In propositional logic all propositions or statements have a clear assigned truth value which can either be false or true. Propositional logic is relatively limited in its expressive power, as it cannot make any statements about objects relationships to each other. Furthermore, no quantifiers can be used to make more complex statements(Russell2009). First-order logic, on the other hand, represents an extension to propositional logic, as it adds quantifiers, predicates and functions (Graf1996). That is also how first Order Logic makes it possible to represent the properties of objects and the relationships between different objects in detail, which is essential for modeling and understanding complex systems. In First Order Logic, functions are defined as mappings that assign objects to other objects. Predicates in First Order Logic function as relations or properties of objects and can have different arities, which gives them the ability to express complex relationships between one or more objects.(Rautenberg2010)

Similar to propositional logic, first order logic uses logical operators such as AND, OR, NOT, IMPLIES and EQUIVALENT, which allow logical links between statements to be formulated, but with the extended ability to go deeper into the structure of objects and their relationships. To better understand the difference between Propositional Logic and First Order Logic, here is an example of a statement in First Order Logic:

$$\forall x \, (\text{Person}(x) \rightarrow \exists y \, (\text{hasJob}(x, y)))$$

The statement simply says that every person x has an occupation y. The universal quantifier is used to make a statement about all possible instances x of a set. Here, an x is a concrete instance of a person. The predicate $Person(x)$ is a function that returns true if the object x is actually a person. The following implication simply connects two statements. The implication is true or false depending on the assignment of the truth values of the two statements (see table 1). The existential quantifier is used to indicate the existence of at least one object that fulfills a certain property or relationship. Here it is used to declare that there is at least one occupation y that person x practises. This predicate checks whether person x has profession y. It is a relational function that takes two arguments: a person and an occupation. It returns "true" if there is a relationship between these two objects that x has profession y. If you now consider how you could express this FOL statement with the possibilities of propositional logic, you would quickly realize that this can become very complicated. First of all, we have to think about how we can represent the statement without using quantifiers. Since the statement is about all possible instances of a person, we have to formulate a single statement for each individual person P when using propositional logic structures:

$$P_1 \rightarrow (J_{1,1} \vee J_{1,2} \vee \cdots \vee J_{1,n})$$

...

$$P_m \rightarrow (J_{m,1} \vee J_{m,2} \vee \cdots \vee J_{1,n})$$

Here P_m stands for the mth instance of a person and n for the nth job that this person can have. If we were to use propositional logic instead of FOL to model this situation, there would be problems with the scalability of the statements in practice. While theoretically one can even make such a statement for every single person of a considered set and thus dispense with FOL (even if it is much more complex), there may well be statements that cannot be modeled with propositional logic because they refer to an infinite set. A simple mathematical example would be $\forall x \, \exists y \, (y = x^2)$. This statement says that for all possible numbers x, there is a number y that is the square of x. If we now consider the set of natural numbers for x and y, then we have an infinite set here. This can no longer be represented with propositional logic.

2.3 Resolution in logic systems

In the following section, I would like to explain the basics of the resolution algorithm, starting with a general explanation based on propositional logic inputs and then explaining which changes and extensions are necessary to apply resolution to first order logic inputs.

2.3.1 Performing Resolution on propositional logic inputs

The resolution algorithm is about deriving a proof by contradiction (Robinson1965). In general, the input for the resolution algorithm consists of a certain number of axioms that are valid and a conjecture that needs to be proved or disproved. In the context of propositional logic, both the axioms and the conjecture are in a CNF clause form. The resolution algorithm is applied to this entire clause set, whereby the conjecture is added to the clause set in a negated form before the resolution algorithm is applied. When applying the resolution, two clauses from the clause set are resolved with each other. Resolving two clauses is possible if there is a complementary literal pair in both clauses, i.e. if a literal occurs in the negated form in one clause and in a non-negated form in the other clause (Bachmair2001). An example of this would be $clause1 : (P \vee Q)$ and $clause2 : (\neg P \vee R)$, where P and $\neg P$ represent the complementary literal pair. When resolving two clauses, the complementary literal pair is removed and the remaining contents of the clauses, i.e. the remaining literals, are merged and form a new clause, the so-called resolvent. To stay with the example above: Resolving the clauses $(P \vee Q)$ and $(\neg P \vee R)$ would produce the new clause $(Q \vee R)$ as a result. This new clause is now added to the remaining clause set and can now also be used for other resolving options. This resolving step is often presented as follows (Bachmair2001):

$$\frac{\begin{array}{c}(P \vee Q)\\(\neg P \vee R)\end{array}}{(Q \vee R)}$$

It is important to note that only one complementary literal pair is removed per resolving step. With these two clauses $(P \vee Q)$ and $(\neg P \vee \neg Q)$, there are two complementary pairs: P, $\neg P$ and Q, $\neg Q$. However, only one pair may be considered and removed per resolving step. It is therefore not possible to remove both complementary pairs, which would lead to an empty clause, but the result of the resolving is either $(P \vee \neg P)$ or $(Q \vee \neg Q)$. In principle, each clause can be resolved (even multiple times) with another clause, provided there is a complementary literal pair. This process is repeated until no further new resolvings are possible in the clause set or until a so-called empty clause is found, which represents a contradiction (Robinson1965). An empty clause arises when two resolving clauses each contain a literal of a complementary pair, e.g. $clause1 : (Q)$ and $clause2 : (\neg Q)$ would lead to an empty clause if these two clauses were resolved, because as explained above, when two clauses are resolved, the complementary pair is removed and the remaining clause content is merged. However, there is no remaining clause content here, i.e. no further literals in either clause, which is why the result here is an empty clause. As soon as an empty clause is created in the resolution process, the algorithm stops and the result is that the input is unsatisfiable. If, on the other hand, all possible resolution combinations have been made and the clause set does not produce an empty clause, i.e. no contradiction, then the input is satisfiable (Robinson1965). Satisfiable is understood to imply that it is possible to assign truth values to all variables or literals of a clause set so that all clauses of the formulas become true. This is the case if no empty clause can be found, as this would represent a contradiction (Russell2009). If, on the other hand, the resolution algorithm generates an empty clause, then the input is unsatisfiable. This is to be understood as a clause set being unsatisfiable, as there is no possible truth value assignment that makes the entire clause set true. An empty clause is created in the resolution process because all possibilities of assigning truth values to the variables end in a contradiction (Russell2009). This

means that the original assumptions are inconsistent with the negated conjecture. The resolution process does not directly assign truth values to individual literals and clauses, but attempts to uncover structural contradictions. If you now have two clauses (P) and $(\neg P)$, then P would have to be both true and false at the same time in order to make both original clauses true. But this is not possible, which shows the contradiction at precisely this point. The resolution algorithm therefore basically simulates the processing of the truth value assignments by gradually resolving clauses with complementary literal pairs until an empty clause is found or until no further new resolutions are possible (Robinson1965). If finally no empty clause is found, this means that there is at least one assignment of the truth values of the literals that makes the entire clause set true. If there is an empty clause, there is therefore no assignment that would make the entire clause set true.

Example. Application of the resolution algorithm to an input that is satisfiable: The input is in a CNF form and consists of the clauses:

$$Clause1 : (P \vee Q)$$

$$Clause2 : (\neg Q \vee R)$$

$$Clause3 : (\neg P \vee \neg R)$$

Clauses that are being resolved	Result of the resolving process (resolved clause)
1, 2	$4 : (P \vee R)$
2 ,3	$5 : (\neg P \vee \neg Q)$
1 ,3	$6 : (Q \vee \neg R)$
1, 5	$7 : (R \vee \neg R)$, $8 : (Q \vee \neg Q)$
4, 5	$9 : (R \vee \neg Q)$
1, 9	$10 : (P \vee R)$
3, 4	$11 : (P \vee \neg P)$, $12 : (R \vee \neg R)$

Table 2: Satisfiable Input Clauses

All possible combinations that can be resolved have been run through. This did not result in an empty clause, i.e. a contradiction. This input is therefore satisfiable. As each resolved clause is also added to the clause set, the new clause set is now the following:

$$((P \vee Q), (\neg Q \vee R), (\neg P \vee \neg R), (P \vee R),$$

$$((\neg P \vee \neg Q), (Q \vee \neg R), (R \vee \neg R), (Q \vee \neg Q),$$

$$((R \vee \neg Q), (P \vee R), (P \vee \neg P), (R \vee \neg R)$$

The clauses $(P \vee \neg P)$, $(R \vee \neg R)$ and $(Q \vee \neg Q)$ are tautologies, which means that these clauses are always true, no matter what the truth value assignment of the individual literals is. If the literal P has the assignment true, then the clause $(P \vee \neg P)$ is true because of the literal P and because it is a disjunction, which means that at least one variable must be true for the whole expression to be true. If now P has the assignment false, then $\neg P$ would be true and therefore the whole expression is true again (Harrison2009). Strictly speaking, these tautologies do not contain any useful information regarding the detection of contradictions by means of resolution. For this reason, tautologies that can arise in the resolution process are often discarded and not added to the entire clause set. This is because if one were to resolve a tautology, e.g. $(P \vee \neg P)$ with another clause, e.g. $(\neg P \vee R)$, then the second clause would emerge identically from the resolving process, i.e. still $(\neg P \vee R)$. For the sake of completeness, I have listed them here anyway, but it is important to understand that they have no influence on the detection of contradictions.

Example. Application of the resolution algorithm to an input that is unsatisfiable: The input is in a CNF form and consists of the clauses: $1 : (P \vee Q)$ $2 : (\neg P \vee Q)$ $3 : (P \vee \neg Q)$ $4 : (\neg P \vee \neg Q)$ The table 3 shows the resolution options and the resolution process:

Clauses that are being resolved	Result of the resolving process (resolved clause)
1, 2	$5 : (Q)$
1, 3	$6 : (P)$
1, 4:	$7 : (P \vee \neg P)$, $8 : (Q \vee \neg Q)$
4, 5:	$9 : (\neg P)$
6, 9:	$\{\}$ empty clause

Table 3: Unsatisfiable Input Clauses

Not all possible resolving options are shown in this table, as an empty clause was discovered in the process and the other clauses that could otherwise be resolved no longer have any significance. The empty clause is created by resolving clauses 6 and 9, i.e. P and $\neg P$. By applying resolution, it was determined here that the given input is unsatisfiable for all possible assignments of the literals.

2.3.2 Performing Resolution on first order logic inputs

If you want to use resolution on a first order logic input, there are a few special aspects to be aware of. The fundamental principle is identical: two clauses can be resolved if there is a complementary literal pair. If an empty clause occurs during the resolving process, then the given input is unsatisfiable. If, on the other hand, all resolvents have been formed and no empty clause has been generated, then the input is satisfiable. However, the first order logic inputs no longer consist of simple literals and clauses, but can contain very different types, such as functions, terms, quantifiers, predicates, etc., which is why it is important to first convert the input into a CNF form (Russell2009). This already involves many complex steps, for example when dealing with quantifiers, which I will explain in a later chapter. Let us now assume that the first order logic input is in CNF form. Examples of clauses are e.g. $(f(X) \vee g(Y) \vee \neg h(X))$ and $(NOT\, f(X) \vee \neg g(Z) \vee h(skolemconstant(X))$. We now have the predicates f, g and h and their term contents, which varies. If we now resolve these two clauses

in the same way as before, this would work, for example, with the complementary pair $f(X)$ and $\neg f(x)$ and the resolvent would be $(g(Y) \vee \neg h(X) \vee \neg g(Z) \vee h(skolem_constant(X))$. When selecting a complementary pair, please note the following: The predicate name must be identical and so must the content of the term! For example, you could not resolve $\neg h(X)$ and $h(skolemconstant(X)$, as the content is different here, although the predicate itself would be a complementary pair. This is an important characteristic in the application of resolution to first order logic inputs. However, there are also cases in which resolution can still be applied to complementary predicate pairs with different terms. For this purpose, unification is used. In unification, a variable in the term of a predicate is substituted by a more suitable content, with an unifier (Robinson1965). For example, the two clauses above could also be resolved using $g(Y)$ and $\neg g(Z)$ if the content of $\neg g(Z)$ were replaced by a Y. After substituting Z with Y both clauses can be resolved and the complementary pair $g(Y)$ and $\neg g(Y)$ is removed. Unification can be understood as resolving two clauses by substituting something in those clauses. A substitution is to be understood as a mapping σ that systematically replaces variables in terms with other terms. It is used to unify two terms in different clauses, but with the same predicate designation, so that they appear identical. After applying substitution, a complementary pair can now be resolved as part of unification (Robinson1965). A substitution σ is typically represented as a set of pairs $\{x_1/s_1, x_2/s_2, ..., x_n/s_n\}$ with $i = 1, ..., n$, where x_i is a variable which is substituted by a variable s_i. Applying the substitution to a term t is called $t\sigma$ and means that each occurrence of x_i in t is replaced by s_i (Robinson1965). However, as soon as a specific substitution is carried out, it must also be carried out on the entire clause set so that it remains consistent. With larger input files, this can require a great deal of effort and computing capacity, which is why such operations must be carefully considered. What must also be taken into account with unification is that not all complementary predicate pairs can be unified without further ado. The semantic of these pairs must be compatible. For example, $\neg h(X)$ and $h(skolemconstant(X))$ could not be unified, as the structure and semantic of the term content is fundamentally. Especially skolem constants or skolem functions acquire a special treatment to be unified. I have also written specific functions for this in the implementation of my Resolution Prover. Substitutions must therefore be type-appropriate so that the semantics of the terms involved are not violated, i.e. the variables and terms to be substituted must be compatible (Robinson1965). For example, a skolem function, which can arise in the CNF conversion of first order logic inputs, cannot be replaced by any variable, as the skolem function represents certain quantifier dependencies and this information is no longer represented by a substitution with a variable. It must therefore always be checked that the substitution is permitted and also beneficial in the respective context of the specific application. Another important point to note with unification is the avoidance of circular substitution, also known as occur check (Knight1989). It is important to note here that a substitution must not be circular, which means that it must not contain a definition that would indirectly refer to itself. An example of this would be the substitution $\sigma = \{X/f(X), f(Y)/Z\}$, as the substitution of X by $f(X)$ would result in a loop that would expand X endlessly into $f(f(f(...)$ and on and on. I have also implemented a function for this in my resolution prover. Not let us consider another example to visualize the unification process: $Clauses : (p(f(X), Y) \vee q(Y)), (\neg p(Z, g(W)) \vee r(Z))$ If we want to resolve those two clauses, we need to find a substitution σ, so that $p(f(X), Y)\ \sigma = p(Z, g(W))\ \sigma$. We can resolve those clauses by substituting $f(X) = Z$ and $Y = g(W)$, so the substitution mapping can be formal described as$\sigma = \{Z/f(X), Y/g(W)\}$. After applying the substitution we now have the clauses: $(p(f(X), g(W)) \vee q(g(W))), (\neg p(f(X), g(W)) \vee r(f(X)))$, which can be resolved with the resolvent $q(g(W)) \vee r(f(X))$ as result. While the actual core of the resolution process for first

order logic is identical to the application of resolution for propositional logic, the more complex structure of the inputs requires the implementation of additional mechanisms such as checking the term contents and the application of unification, i.e. the substitution and subsequent resolution of two clauses. Since I have implemented a resolution prover for both propositional logic inputs and first order logic inputs, the exact differences will be explained in more detail later using pseudo code.

3 Resolution Prover in Rust

3.1 Software architecture: Overview of program design and structure

For the implementation of my Resolution Prover I used the programming language Rust, version 1.72.1. My project was managed with git and can be viewed under the following link github.com/wrosmus/resolution-prover-in-rust/ As mentioned before, I implemented two resolution provers, first for propositional logic inputs and then for first order logic inputs. In terms of approach and structure, both provers are quite similar. For both provers, I created a data structure to represent the input. I have tried to keep the data structure as simple as possible and at the same time sufficient for all possible cases so as not to complicate the further processes and algorithms even more. While the data structure in the prover for propositional logic only describes literals, clauses and CNFs, in the prover for first order logic it is extended by terms, predicates, formula types, sentence types and formulas, which is consistent with the description of the more complex structure of first order logic inputs. Both provers have a pest parser, which brings the input files into a structure defined in the data structure so that the input can finally be processed by the resolution algorithm. The resolution prover for first order logic inputs also has an additional CNF converter, which converts the parsed input into a CNF form so that the resolution algorithm can process the input. The implementation of the resolution algorithm is also somewhat more complex here, as mechanisms such as unification have been added. When executing the prover for propositional logic, the command line can be used to choose between various sat (satisfiable) or unsat (unsatisfiable) input files, which are then read in and executed. With the prover for first order logic inputs, one can choose between three different inputs, two of which are satisfiable and one unsatisfiable.

3.2 Parser and Grammar: Description of Implementation and challenges

3.2.1 Parser for propositional logic inputs

For the implementation of the parser I used Pest, version 2.7.5. Pest is a parser library in Rust, which can be used to simplify parsing activities. Based on the Parsing Expression Grammar principle, parsing rules are defined in the form of a grammar and stored in a separate ".pest" file. These defined rules can then be processed by the Pest macro and used accordingly within the parser. There is a function in the parser that accepts the input file as an input string and parses it using the Pest grammar. Further functions are then used to iterate over the individual tokens according to the grammar rules, which are then processed and finally converted into a CNF formula according to the data structure. I have defined this grammar for the resolution prover with propositional logic inputs in order to be able to map the input:

programm file: dimacs_grammar.pest

```
file = { SOI ~ (comment ~ "\n"*)* ~ problem ~ "\n"* ~
(clause ~ "\n"*)* ~ clause? ~ EOI }
comment = _{ "c" ~ (!"\n" ~ ANY)* ~ "\n" }
problem = _{ "p" ~ ("cnf" | "sat") ~ n ~ m }
clause = { (lit ~ "\n"?)+ ~ "0" }
lit = @{ "-"? ~ !"0" ~ ASCII_DIGIT+ }
n = @{ ASCII_DIGIT+ }
m = @{ ASCII_DIGIT+ }
WHITESPACE = _{ " " }
```

The input benchmarks used here all have the same structure: Lines with comments start with a *c* and are ignored or skipped by the parser. The same applies to lines beginning with a *p*, as these represent the problem and the number of CNF clauses in the input file. This is followed by the clauses actually to be processed, whereby the end of a clause is marked by a ”0”. The parser processes these rules and outputs a vector ”CNFFormula” at the end, which contains the parsed CNF formulas as a closed list. This vector is then passed to the resolution algorithm.

Example. Here is an illustration using the satisfiable input **xor.cnf** which is formatted as a dimacs input file:

```
c sat
p cnf 2 2
1 2 0
-1 -2 0
```

After parsing, the clauses from the input file are in the following structure:
$Clauseliterals : [Positive(”1”), Positive(”2”)]$
$Clauseliterals : [Negative(”1”), Negative(”2”)]$

3.2.2 Parser for first order logic inputs

While the implementation of a parser for propositional logic inputs was comparatively less complex, the creation of a parser for first order logic inputs was much more demanding. Here, files from the TPTP website were used as input. The formulas to be parsed were no longer just mapped as one clause per line, but consisted of multi-line nested formulas with operators, formula names, sentence types, etc., which meant that the parser had to be fundamentally rebuilt. The frame of the input file is very similar to the previous definition, because here too there is a ”Start of input” followed by a part containing either ”content” or a ”comment” followed by the end of the file, ”EOI”. The comments can also be neglected here. The formulas to be parsed now follow the pattern that they start with an ”fof” (first order formula), followed by a formula name and the type of formula (formula role), for example an axiom or the conjecture. This is followed by the actual formula, which can be nested as required. Some formulas are also preceded by existential or universal quantifiers, whereby the quantifiers can also occur within or between expressions. The formulas can then consist of single predicates or more complex nesting with implications, equivalences and other operators. At this

point I would like to point out that the defined grammar can only be used for a subset of input files on the TPTP website, i.e. only those that follow the exact structure defined in the grammar. For other inputs, further modifications must be made here. Here is an excerpt from the grammar for first order logic inputs, which describes the formulas:

```
fof_formula = { "fof" ~ "(" ~ identifier ~ "," ~ formula_role ~ "," ~
(WHITESPACE | NEWLINE)* ~ formula ~ ")." }
A formula, on the other hand, is defined as follows:
formula = { quantified_formula | atomic_formula | negation ~ formula }
quantified_formula = { quantifier ~ "[" ~ variable_list ~ "]" ~
":" ~ (WHITESPACE | NEWLINE)* ~ operation }
operation = { equality | inequality }
atomic_formula = { predicate ~ "(" ~ term ~ ("," ~ term)* ~ ")" }
predicate = { letters ~ (letters | numbers | "_")* }
```

A term can in turn be a variable, a function or a constant. The complete grammar definition is even more complex and would go beyond the context here. This can be found in my code in any case. Nevertheless, I wanted to show here roughly how quickly the complexity of the inputs has increased, with which the resolution prover for first order logic inputs now has to work. The parser written for this basically works in the same way as with the propositional logic inputs, but here the handling of the individual grammar rules is much more complex. There were therefore numerous problems with the implementation, which could not be completely solved until the end. In particular, there were discrepancies between the application of the defined grammar in the online pest editor pest.rs and between the framework of the resolution prover. After numerous tests and restructurings were carried out in the parser, but without solving the problem, I decided to exclude the parser here and to convert the input files manually (without parser) into the format predefined in the data structure. This was probably not an elegant solution, but due to the consistent and comparatively short input files it was also not a complicated interim solution and finally I wanted to concentrate on the main task, the design of a resolution algorithm. The further development and parser for first order logic inputs is therefore definitely considered as future work. The input files were then put into this structure by me (as they would have been by the parser):

```
ParsedInput {
      clauses: Vec::new(),
      types: Vec::new(),
      sentences,
    }
```

The individual sentences were then presented in this structure:

```
pub struct Sentence {
    pub sentence_type: SentenceType,
    pub name: String,
    pub formula: Formula,
}
```

Example. Visualization of the parsing structure of the first formula of the input file SYN055+1. Representation in the input file:

```
fof(pel25_1,axiom,
    ? [X] : big_p(X) ).

Parsed Structure:
Sentence {
                sentence_type: SentenceType::Axiom,
                name: "pel25_1".to_string(),
                formula: Formula::Quantified(
                    Quantifier::Exists,
                    vec!["X".to_string()],
                    Box::new(Formula::Predicate(Predicate {
                        name: "big_p".to_string(),
                        arity: 1,
                        terms: vec![Term::Variable("X".to_string())],
                    })),
                ),
            }
```

All the ”parsed” input files can also be found in the cnf_converter.rs file. The input files are processed there in the ”parsed” structure in order to generate CNF clauses.

3.3 Conversion to CNF

For the resolution prover, which processes first order logic inputs, I also had to implement a CNF converter, as the parsed inputs could not be used for the resolution algorithm. The resolution algorithm is designed to process CNF clause inputs. To convert first order logic formulas into a CNF form, I implemented the following steps: (Russell2009)

1. Standardizing variables
2. Eliminate implications and equivalences
3. Convert formula into a negation normal form (NNF)
4. Convert formula into a prenex normal form (PNF)
5. Skolemization
6. Final conversion to CNF

I would now like to explain these six steps in more detail by the use of pseudo code descriptions.

3.3.1 Standardizing variables

The first function called is rename_variable(). The corresponding pseudo code can be found in: "Algorithm 1 Clean Formula" and "Algorithm 2 Rename a Variable". This is required if variables have to be renamed within an expression which could lead to conflicts in the further course of processing and interpretation (Russell2009). A simple example here would be the following: $\forall x\ \exists y\ (P(x) \wedge \forall x\ (Q(x,y) \wedge R(y)))$ This formula has a nested quantifier $\forall x$, which is used within another quantifier $\forall x$ with the same variable name. This could lead to confusion, as the inner variable x hides the outer variable x. These scope conflicts are recognized by the clean_formula() function and the variables in question are renamed using the rename_variable() function so that they can be distinguished. In this example, a clear renaming could look as follows: $\forall x\ \exists y\ (P(x) \wedge \forall x_2\ (Q(x_2,y) \wedge R(y)))$. Here, a _2-ending is added to the nested universal quantifier expression for the relevant variable X, which makes this variable distinguishable. The use of these two functions ensures that there are no name conflicts within a formula.

3.3.2 Eliminate implications and equivalences

After the formulas have been checked for possible name conflicts, the next step is to resolve implications and equivalences (Russell2009), if they exist. The corresponding pseudo code can be found in: "Algorithm 1 Eliminate Implications and Equivalences". An equivalence expression $Q(Y) \Leftrightarrow P(Y)$ can also be expressed as $(Q(Y) \rightarrow P(Y)) \wedge (P(Y) \rightarrow Q(Y))$, i.e. as two implications connected by an AND. An implication $P(X) \rightarrow Q(X)$, on the other hand, can also be represented as $\neg P(X) \vee Q(X)$. In order to bring a given input into CNF, it is essential to resolve implications and equivalences and represent them with the NOT, AND, OR operators. The function eliminate_implications_and_equivalences() ensures this at this point. Equivalences are first converted into implications and implications, e.g. $P(X) \rightarrow Q(X)$ are converted into the form $\neg P(X) \vee Q(X)$, i.e. into a disjunction in which the left literal of the previous implication expression (here $P(X)$) is negated. To guarantee the entire conversion, the function is called in a recursive way and applied to the entire nested structure of the formula. As quantifiers may also be present at this point in the CNF conversion, this function has been designed so that quantified expressions are also handled correctly. This guarantees that the same logic is also applied recursively to the content, i.e. the scope, of the corresponding quantifier during the conversion. The eliminate_implications_and_equivalences function is executed within a loop until no more changes are made to the formula, which is the case if there are no more implications or equivalences.

Algorithm 1 Eliminate Implications and Equivalences

Require: A formula *formula*

Ensure: Returns a formula where implications and equivalences are resolved into NOT, AND, OR

```
current_formula ← formula.clone()
loop
    new_formula ← MATCH(current_formula) current_formula Implies(lhs, rhs)
    neg_lhs ← Not(lhs.clone())                          ▷ Convert implication to disjunction
    new_rhs ← rhs.clone()
    return Or([neg_lhs, new_rhs])
     Iff(lhs, rhs)
    lhs_implies_rhs ← Implies(lhs.clone(), rhs.clone())
    rhs_implies_lhs ← Implies(rhs.clone(), lhs.clone())
    return And([lhs_implies_rhs, rhs_implies_lhs]) ▷ Convert equivalence into two implications
     Not(sub_formula)
    return Not(ELIMINATEIMPLICATIONSANDEQUIVALENCES(sub_formula))
     And(sub_formulas)
    new_sub_formulas ← Map(ELIMINATEIMPLICATIONSANDEQUIVALENCES, sub_formulas)
    return And(new_sub_formulas)
     Or(sub_formulas)
    new_sub_formulas ← Map(ELIMINATEIMPLICATIONSANDEQUIVALENCES, sub_formulas)
    return Or(new_sub_formulas)
     Quantified(quantifier, vars, sub_formula)          ▷ Apply recursively within quantifiers
    new_sub_formula ← ELIMINATEIMPLICATIONSANDEQUIVALENCES(sub_formula)
    return Quantified(quantifier, vars, new_sub_formula)
    return current_formula.clone()
    if new_formula == current_formula then
        break
    end if
    current_formula ← new_formula
end loop
return current_formula
```

3.3.3 Converting formula into Negation Normal Form (NNF)

After ensuring that all implications and equivalences have been resolved into expressions that only contain the operators AND, OR, NOT, the formula is now converted into a negation normal form (Russell2009). The corresponding pseudo code can be found in: "Algorithm 2 Convert to Negation Normal Form (NNF)" and "Algorithm 3 Process Negation to achieve Negation Normal Form".The NOT operators "¬" are pulled into the expression as far as possible until they are directly in front of an elementary logical expression that no longer contains nested AND/OR structures, for example a predicate $P(X)$(Harrison2009). It should be noted that when applying negation to parenthesized expressions, the operators are reversed due to the "pushing in", for example this expression $\neg(P(X) \vee (Q(X))$ would look like this in the negation normal form: $(\neg P(X) \wedge \neg Q(X))$, as the De Morgan laws are applied here. Here, by applying the negation to the parenthesized

expression, the OR operator is converted into an AND operator, which also applies analogously to the application of a negation to a parenthesized AND expression, which would then change into an OR expression. This process of pushing negation inward is the central component of the formation of a negation normal form and is carried out by applying the De Morgan laws to the formula. The De Morgan laws describe exactly this process of reversing the operators in the case of a negation(Harrison2009). Formally speaking, the De Morgan laws can be described as follows (Harrison2009): The negation of a conjunction of two statements is equivalent to the disjunction of the negations of these statements: $\neg(P(X) \wedge Q(X)) \equiv \neg P(X) \vee \neg Q(X)$. Similarly, the negation of a disjunction of two statements is equivalent to the conjunction of the negations of these statements: $\neg(P(X) \vee Q(X)) \equiv \neg P(X) \wedge \neg Q(X)$. The application of De Morgan's laws applies not only to variables and operators, but also to quantifiers (Russell2009). For example, the negation of an existential quantifier $\neg(\exists x\, P(x))$ leads to $\forall x\ \neg P(x)$ and vice versa for universal quantifiers. These mathematical principles are performed recursively by the functions to_negation_normal_form() and process_negation() to ensure that each subformula of a more complex formula is handled correctly with regard to negation. After using the functions to_negation_normal_form() and process_negation() to form an NNF, the function is_fully_negation_normalized() is also used to check whether all negations have actually been sufficiently included in the nested expressions or whether there are any underlying expressions that contain AND/OR operators.

Algorithm 2 Convert to Negation Normal Form (NNF)

Require: A formula *formula*
Ensure: Returns the formula in negation normal form
formula Not(sub_formula)
return PROCESSNEGATION($self, sub\_formula$) ▷ Process negations recursively
And(formulas)
$nnf\_formulas \leftarrow$ MAP($self, ToNegationNormalForm, formulas$)
return And($nnf\_formulas$) ▷ Apply NNF to each subformula in conjunction
Or(formulas)
$nnf\_formulas \leftarrow$ MAP($self, ToNegationNormalForm, formulas$)
return Or($nnf\_formulas$) ▷ Apply NNF to each subformula in disjunction
Quantified(quantifier, vars, sub_formula)
$nnf\_sub\_formula \leftarrow$ TONEGATIONNORMALFORM($self, sub\_formula$)
return Quantified($quantifier, vars, nnf\_sub\_formula$) ▷ Handle quantifiers recursively
return $formula$

Algorithm 3 Process Negation to achieve Negation Normal Form

Require: A formula *formula*
Ensure: Returns the formula with negation processed according to De Morgan's laws

```
formula Not(inner)
  return ToNegationNormalForm(self, inner)                    ▷ Eliminate double negation
And(formulas)
  negated_formulas ← Map(self, ToNegationNormalForm, Map(Not, formulas))
  return Or(negated_formulas)                                 ▷ Apply De Morgan's laws: ¬(P ∧ Q) ≡ ¬P ∨ ¬Q
Or(formulas)
  negated_formulas ← Map(self, ToNegationNormalForm, Map(Not, formulas))
  return And(negated_formulas)                                ▷ Apply De Morgan's laws: ¬(P ∨ Q) ≡ ¬P ∧ ¬Q
Quantified(quantifier, vars, sub_formula)
  new_quantifier ← InvertQuantifier(quantifier)
  negated_sub_formula ← ToNegationNormalForm(Not(sub_formula))
  return Quantified(new_quantifier, vars, negated_sub_formula)
                                                              ▷ Invert quantifier and apply negation inside
  return Not(formula)                                         ▷ Base case: Negate simple expressions directly
```

3.3.4 Converting Formula into Prenex Normal Form (PNF)

The next transformation section deals with creating prenex forms of the formulas (Kroetzsch2017). The prenex normal form (PNF) is a form of representation of formulas in which all quantifiers $\exists$, $\forall$ are moved to the beginning of the formula. The main process of PNF formation takes place here in the function to_prenex_normal_form(), which uses the auxiliary functions extract_and_collect_quantifiers() and apply_collected_quantifiers() to control the extraction and shifting of the quantifiers. As the scope uniqueness of the quantifiers was ensured in the first step by possibly renaming variables, this should continue to be ensured at this point by moving the quantifiers. The quantifiers are extracted by recursively iterating through all subformulas. An example of the formation of the PNF would be the following: Let $(\forall x\, P(x)) \wedge \forall y\, (\neg Q(y) \vee R(y))$ be a formula in NNF. After extracting and shifting the quantifiers, this formula would look like this in PNF: $\forall x\, \forall y\, ((P(x)) \wedge (\neg Q(y) \vee R(y)))$. The step of forming the prenex normal form is very important to prepare the formulas for the skolemization process, which is explained in the next section.

3.3.5 Skolemization

Now we have the formulas in a prenex normal form, which also contains the negation normal form. However, the PNF still contains quantifiers that cannot be handled by the resolution algorithm. Therefore, it must now be ensured that the quantifiers can be removed without changing the semantics of the expressions. This process is called skolemization. In particular, this involves eliminating existential quantifiers by generating a unique Skolem constant or a skolem function for each variable that is bound by an existential quantifier (Russell2009). The difference in the use of a skolem constant or a skolem function is whether an existential quantifier has no surrounding universal quantifiers or whether it does. If an existential quantifier has no surrounding universal quantifiers, then a unique skolem constant representing the statement of the existential quantifier can

be introduced quite simply and the quantifier can be removed. However, if surrounding universal quantifiers are present within the skolemization of an existential quantifier, then a skolem function must be generated which indicates this dependency and whose arguments are determined by the variables of the universal quantifiers. If there are only universal quantifiers in the formulas that are not in the scope of influence of existential quantifiers, then they remain during the skolemization and are deleted afterwards since they can be ignored during the skolemization process if they are not followed by existential quantifiers. Let us now look at the following example to explain skolemization in more detail:

$$\exists y \, (P(x) \wedge \neg Q(y))$$

Here, there is only the existential quantifier $\exists y$ and no preceding universal quantifier, which means that the introduction of a skolem constant is sufficient to eliminate the existential quantifier. This is done by replacing the y in $\neg Q(y)$ with a unique skolem constant, for example "c". After skolemization, the formula would therefore look as follows: $P(x) \wedge \neg Q(c)$. Let us now look at this formula: $\forall x \, \exists y \, (P(x) \wedge Q(y,x))$. Here there is an existential quantifier that depends on a preceding universal quantifier. Therefore, in the context of skolemization, the existential quantifier $\exists y$ must be represented by a skolem function, which represents the dependence of the existential quantifier on the all quantifiers. Here, the variable of the function is determined by the variable of the universal quantifier. We therefore replace y with $f(x)$ here. The skolemized formula then finally reads: $\forall x \, (P(x) \wedge Q(f(x),x))$, whereby the universal quantifier can then be removed in the next step without having to carry out special skolemization steps. Skolemization is therefore specifically about removing existential quantifiers. If, on the other hand, it is a matter of removing an all quantifier, then this is possible without further ado as long as it does not lie within the sphere of influence of an existential quantifier. This is because an universal quantifier basically means that a statement applies to all individuals in a formula. Since we ensure that each clause is true regardless of the specific binding of a variable represented by an universal quantifier when forming CNF formulas, the universal quantifier can be omitted here, since the statement made by the universal quantifier is true for all instances of the variable anyway. The corresponding pseudo code can be found in: "Algorithm 4 Transform to Skolem Normal Form".

Algorithm 4 Transform to Skolem Normal Form

Require: A formula *formula*, set of active quantifiers *active_quantifiers*
Ensure: Returns the formula in Skolem normal form

```
    formula Quantified(Exists, vars, sub_formula)
  skolem_terms ← MAP(vars, GenerateSkolemTerm, active_quantifiers)
  skolemized_formula ← REPLACEVARSWITHTERMS(sub_formula, vars, skolem_terms)
  return TOSKOLEMNORMALFORM(skolemized_formula, active_quantifiers)
                                  ▷ Replace existential quantifiers with Skolem terms
    Quantified(ForAll, vars, sub_formula)
  new_active_quantifiers ← active_quantifiers ∪ vars
  result ← TOSKOLEMNORMALFORM(sub_formula, new_active_quantifiers)
  return Quantified(ForAll, vars, result)   ▷ Keep universal quantifiers, update active set
    And(formulas) — Or(formulas)
  processed_formulas ← MAP(formulas, ToSkolemNormalForm, active_quantifiers)
  return And/Or(processed_formulas)   ▷ Apply transformation recursively to subformulas
    Not(sub_formula)
  processed_formula ← TOSKOLEMNORMALFORM(sub_formula, active_quantifiers)
  return Not(processed_formula)
  return formula                      ▷ Literals and base cases are unchanged
```

3.3.6 Final Conversion into CNF

The final step of the CNF conversion process is to convert the skolemized formulas into a CNF form. Before executing this step, all remaining universal quantifiers are removed (Russell2009). We then have a quantifier-free formula, which is now examined for its structure, i.e. the distribution of OR or AND operators. For this purpose, the function recognize_structure() was first implemented, which analyzes the current structure of the skolemized formula (Corresponding pseudo code: Algorithm 5 Recognize Structure and Convert to CNF). In some cases, a CNF already exists here and the CNF conversion process is successfully completed. However, it is also possible that a DNF or another mixed form is present here. If a DNF is present, a conversion to CNF can be ensured quite simply by using reverse distribution (Kroetzsch2017). The dnf_to_cnf() function is called for this purpose. The corresponding pseudo code can be found in: "Algorithm 6 Convert DNF to CNF", "Algorithm 7 Apply Distributive Law" and "Algorithm 8 Apply Reverse Distributive Law". Consider the following example:
Clause in DNF: $(P(X) \wedge Q(X)) \vee (R(X) \wedge S(X))$
The distributive law is defined as follows:

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

If we now consider

$$A = P(X) \wedge Q(X), B = R(X), C = S(X)$$

with

$$(P(X) \wedge Q(X)) \vee (R(X) \wedge S(X)) = A \vee (B \wedge C),$$

then the distributive law can be applied here as follows:
First distribute A over $(B \wedge C)$: $(A \vee (B \wedge C)) = (A \vee B) \wedge (A \vee C)$.
Since $A = P(X) \wedge Q(X)$, $B = R(X)$ and $C = S(X)$, one distributes $P(X) \wedge Q(X)$ over $(R(X) \wedge S(X))$:

$$(P(X) \wedge Q(X)) \vee (R(X) \wedge S(X)) = ((P(X) \wedge Q(X)) veeR(X)) \wedge ((P(X) \wedge Q(X)) \vee S(X))$$

We now transform this expression further according to the distribution and get the following CNF form as a result:

$$(P(X) \vee R(X)) \wedge (Q(X) \vee R(X)) \wedge (P(X) \vee S(X)) \wedge (Q(X) \vee S(X))$$

However, it is also possible that there is a more complex nested form which is neither in CNF nor in DNF. In this case, the much more complex apply_distributive_law function() was developed, which first analyzes individual subformulas and executes the corresponding distribution or reverse distribution function. With comparatively small inputs, the application of the distribution can be used in a resource-saving manner, but as soon as the formulas become more complex and nested, the complexity of the distribution can increase very quickly, which means that this approach is very inefficient. The Tseytin transformation could be used for this instead, which I will briefly touch on later. Now that all the steps for creating CNF formulas have been completed and the formula is finally available in CNF, there is one last check for the presence of tautologies. A tautology in a clause means that this clause is always true, regardless of the assignment of the truth values for the literals in this clause. An example of a tautology would be $(P \vee \neg P)$. This clause is always true and therefore has no influence on the finding of empty clauses in the resolution process. In order to reduce the number of clauses before the resolution process without losing semantics, tautologies that may have been generated during the CNF conversion process are therefore removed at this point.

Algorithm 5 Recognize Structure and Convert to CNF

Require: A formula *formula*
Ensure: Returns the formula in CNF form
  if IsCNF(formula) **then**
    print "Formula is already in CNF"
    return $formula.clone()$
  else if IsDNF(formula) **then**
    print "Formula is a pure DNF. Converting using distribution."
    return DnfToCnf(formula) ▷ Convert DNF to CNF using distributive laws
  else
    print "Formula is neither pure CNF nor DNF, applying special distribution."
    return ApplyDistributiveLaw(formula)
      ▷ Apply complex distribution methods to convert to CNF
  end if

Algorithm 6 Convert DNF to CNF

Require: A formula *formula*
Ensure: Returns the formula converted from DNF to CNF

```
if IsOr(formula) then                                   ▷ Check if the formula is in DNF (OR of ANDs)
    combined ← []                                       ▷ Initialize an empty list to store combinations
    for each disjunct in formula do
        if IsAnd(disjunct) then                         ▷ Check if the disjunct is an AND formula
            new_combined ← []                           ▷ Initialize a new list for this round of combinations
            for each existing_clause in combined do
                for each conjunct in disjunct do
                    new_clause ← existing_clause + [conjunct]
                                                        ▷ Combine existing clauses with current conjunct
                    new_combined.append(new_clause)
                                                        ▷ Add the new clause to the list of combined clauses
                end for
            end for
            combined ← new_combined                     ▷ Update the combined list with the newly combined
clauses
        else
            for each clause in combined do
                clause.append(disjunct)                 ▷ Add the disjunct to each existing clause
            end for
        end if
    end for
    cnf_formula ← Map(Formula::Or, combined)
    return Formula::And(cnf_formula)                    ▷ Return the CNF formula
end if
```

Algorithm 7 Apply Distributive Law

Require: A formula *formula*
Ensure: Returns the formula with applied distributive law
  if IsOr(formula) **then** ▷ Check if the top-level operator is OR
    $(and\_formulas, other\_formulas) \leftarrow$ Partition(formula, Formula::And) ▷ Separate AND formulas from others
    if $and\_formulas$.is_empty() **or** $other\_formulas$.is_empty() **then** ▷ Check if one of the lists is empty
      return SimplifyOrClauses(formula) ▷ Simplify if no distribution is possible
    end if
    $distributed \leftarrow$ DistributeOrOverAnd($and\_formulas$, $other\_formulas$) ▷ Distribute OR over AND if possible
    return ApplyDistributiveLaw($distributed$) ▷ Recursively apply distributive law to result
  else if IsAnd(formula) **then** ▷ Check if the top-level operator is AND
    $results \leftarrow$ Map(ApplyDistributiveLaw, formula)
    ▷ Apply distributive law to each sub-formula
    return SimplifyAndClauses($results$)
    ▷ Simplify the results of the distribution
  else
    return $formula.clone()$ ▷ Return the formula unchanged if it's neither AND nor OR
  end if

Algorithm 8 Apply Reverse Distributive Law

Require: A formula *formula*
Ensure: Returns the formula with applied reverse distributive law
  $and\_lists \leftarrow$ CollectAndLists(formula)
  ▷ Gather lists of AND sub-formulas from the main formula
  if and_lists.is_empty() **then** ▷ Check if there are no AND sub-formulas
    return Formula::Or($formula$.to_vec())
    ▷ Return the formula as an OR of its components if no ANDs are found
  end if
  $cartesian\_product \leftarrow$ CartesianProduct(and_lists)
  ▷ Compute the cartesian product of the AND lists to form new OR formulas
  $new\_subformulas \leftarrow$ Map(Formula::Or, $cartesian\_product$)
  ▷ Convert each product into an OR formula
  return SimplifyAndClauses($new\_subformulas$) ▷ Simplify with commutative law

3.3.7 Example for converting a first order logic input into CNF

Now that I have explained the roughly divided six transformation steps for CNF conversion, I would like to explain this using a concrete example, which I also use as input in the Resolution Prover. This example is the third formula of the input file **SYN056+1**:

```
fof(pel26,conjecture,
    ( ! [X] :
        ( big_p(X)
       => big_r(X) )
  <=> ! [Y] :
        ( big_q(Y)
       => big_s(Y) ) ) ).
```

This formula is first passed to the CNF converter in the following structure:

```
"Iff(Quantified(ForAll, ["X"], Implies(Predicate
(Predicate { name: "big_p", arity: 1, terms: [Variable("X")] }),
Predicate(Predicate { name: "big_r", arity: 1, terms: [Variable("X")] }))),
Quantified(ForAll, ["Y"], Implies(Predicate
(Predicate { name: "big_q", arity: 1, terms: [Variable("Y")] }),
Predicate(Predicate { name: "big_s", arity: 1, terms: [Variable("Y")] }))))".
```

The sentence type is then analyzed, which in this example is the conjecture. In the resolution, the conjecture is the assertion that is checked for correctness by attempting to show a contradiction. The conjecture is therefore added to the rest of the clause set (often axioms) in negated form. This is also the next step performed by the CNF converter, which is why the formula in the next step is as follows:

```
Negated conjecture:

Not(Iff(Quantified(ForAll, ["X"],
Implies(Predicate(Predicate { name: "big_p", arity: 1,
terms: [Variable("X")] }),
Predicate(Predicate { name: "big_r", arity: 1, terms: [Variable("X")] }))),
Quantified(ForAll, ["Y"], Implies(Predicate
(Predicate { name: "big_q", arity: 1, terms: [Variable("Y")] }),
Predicate(Predicate { name: "big_s", arity: 1, terms: [Variable("Y")] })))))
```

The six transformation steps are now used. First, the clean_formula function is used to check whether there is a need to rename variables if they would lead to conflicts. This is not the case here, which is why the negated conjecture is passed on to the next function eliminate_implications_and_equivalences, in which the implications and equivalences are resolved. There are a total of two implication expressions here, which are set to be equivalent to each other. The breaking down of the implications and equivalences using AND, OR, NOT operators finally converts the formula into the following:

```
Without implications and equivalences:

Not(And([Or([Not(Quantified(ForAll, ["X"],
Or([Not(Predicate(Predicate { name: "big_p", arity: 1,
terms: [Variable("X")] })),
Predicate(Predicate { name: "big_r", arity: 1,
```

```
terms: [Variable("X")] }))))),
Quantified(ForAll, ["Y"], Or([Not(Predicate
(Predicate { name: "big_q", arity: 1, terms: [Variable("Y")] })),
Predicate(Predicate { name: "big_s", arity: 1,
terms: [Variable("Y")] }))])),
Or([Not(Quantified(ForAll, ["Y"], Or([Not(Predicate
(Predicate { name: "big_q", arity: 1, terms: [Variable("Y")] })),
Predicate(Predicate { name: "big_s", arity: 1,
terms: [Variable("Y")] })]))),
Quantified(ForAll, ["X"], Or([Not(Predicate(Predicate
{ name: "big_p", arity: 1, terms: [Variable("X")] })),
Predicate(Predicate { name: "big_r", arity: 1,
terms: [Variable("X")] }))))))))))
```

Once this step has been completed, the formula must now be converted into a negation normal form, i.e. the negations are drawn into the bracketed expressions and applied to the literals, quantifiers and operators. This is quite a challenge for this function, as it is currently very complex and nested. Nevertheless, the algorithm can generate a correct NNF using a recursive application of the pushing negation inwards mechanism, which now looks like this:

```
Negation Normal Form:

Or([And([Quantified(ForAll, ["X"], Or([Not(Predicate
(Predicate { name: "big_p", arity: 1,
terms: [Variable("X")] })),
Predicate(Predicate { name: "big_r", arity: 1,
terms: [Variable("X")] })])),
Quantified(Exists, ["Y"], And([Predicate
(Predicate { name: "big_q", arity: 1,
terms: [Variable("Y")] }),
Not(Predicate(Predicate { name: "big_s", arity: 1,
terms: [Variable("Y")] }))))]),
And([Quantified(ForAll, ["Y"], Or([Not(Predicate
(Predicate { name: "big_q", arity: 1,
terms: [Variable("Y")] })),
Predicate(Predicate { name: "big_s", arity: 1,
terms: [Variable("Y")] })])),
Quantified(Exists, ["X"], And([Predicate
(Predicate { name: "big_p", arity: 1,
terms: [Variable("X")] }),
Not(Predicate(Predicate { name: "big_r", arity: 1,
terms: [Variable("X")] }))))])])).
```

In the next step, the prenex normal form (PNF) is formed, i.e. the quantifiers are moved to the beginning of the expression, which generates this formula:

```
Prenex Normal Form:
```

```
Quantified(Exists, ["X"], Quantified(ForAll, ["Y"],
Quantified(Exists, ["Y"], Quantified(ForAll, ["X"],
Or([And([Or([Not(Predicate
(Predicate { name: "big_p", arity: 1,
terms: [Variable("X")] })),
Predicate(Predicate { name: "big_r", arity: 1,
terms: [Variable("X")] })]),
And([Predicate(Predicate { name: "big_q", arity: 1,
terms: [Variable("Y")] }),
Not(Predicate(Predicate { name: "big_s", arity: 1,
terms: [Variable("Y")] }))])]),
And([Or([Not(Predicate(Predicate { name: "big_q", arity: 1,
terms: [Variable("Y")] })),
Predicate(Predicate { name: "big_s", arity: 1,
terms: [Variable("Y")] })]),
And([Predicate(Predicate { name: "big_p", arity: 1,
terms: [Variable("X")] }),
Not(Predicate(Predicate { name: "big_r", arity: 1, terms:
[Variable("X")] }))])])]))))))
```

Afterwards the formula is then skolemized in PNF. During skolemization, the existential quantifiers are first replaced by a skolem constant or a skolem function, depending on whether the existential quantifier is in the scope of an universal quantifier. After the replacement the existential quantifiers are deleted from the formula. The first existential quantifier ”Exists, [”X”]” is not in the scope of an universal quantifier, which is why a skolem constant is used here for all occurrences of the variable X. This skolem constant is unique and is designated by the function as skolem_constant_2, whereby the ending 2 guarantees the uniqueness here, as other skolem constants have a different ending. After the skolem constant has been used for all occurrences of the variable X, the corresponding existence quantifier is removed. The next quantifiers are then considered. This is followed by an universal quantifier, which does not require skolemization and is therefore skipped but not deleted. This is followed by another existential quantifier ”Exists, [”Y”]”, which is in the scope of the previous universal quantifier ”ForAll, [”Y”]”. Therefore, a skolem function must be generated here as part of the skolemization, which represents the dependency of the previous all quantifier. This is done by creating a skolem function, in this case skolem_function_3 (here too, the ending ensures the uniqueness of the created function) and inserting it into the corresponding terms that currently contain the variable Y. The Y then becomes the argument of the skolem function. Since this formula now only contains universal quantifiers and all existential quantifiers have been successfully skolemized, the skolemization process is complete at this point. In the next step, the remaining universal quantifiers are removed, which leads to this quantifier-free formula:

```
Quantifier-Free-Formula:

Or([And([Or([Not(Predicate(Predicate { name: "big_p", arity: 1,
terms: [Constant("skolem_constant_2")] })),
Predicate(Predicate { name: "big_r", arity: 1,
```

```
terms: [Constant("skolem_constant_2")] })]),
And([Predicate(Predicate { name: "big_q", arity: 1,
terms: [Function { name: "skolem_function_3", arity: 1,
args: [Variable("Y")] }] }),
Not(Predicate(Predicate { name: "big_s", arity: 1,
terms: [Function { name: "skolem_function_3", arity: 1, args:
[Variable("Y")] }] }))])]),
And([Or([Not(Predicate(Predicate { name: "big_q", arity: 1,
terms: [Function { name: "skolem_function_3", arity: 1, args:
[Variable("Y")] }] })),
Predicate(Predicate { name: "big_s", arity: 1,
terms: [Function { name: "skolem_function_3", arity: 1, args:
[Variable("Y")] }] })]),
And([Predicate(Predicate { name: "big_p", arity: 1,
terms: [Constant("skolem_constant_2")] }),
Not(Predicate(Predicate { name: "big_r", arity: 1,
terms: [Constant("skolem_constant_2")] }))])])])
```

The actual CNF form is then created in the last step of the conversion. This is done by means of the distribution, which is quite complicated for larger and more complex formulas, such as the formula we have here. I will now try to break down all the steps as simply as possible: First of all, it is established here that the formula in question is neither already in CNF nor in DNF. The formula is therefore broken down into sub-formulas and combined using the distributive laws in such a way that a CNF form is created. This part here is processed using the distributive law (function apply_distributive_law()):

```
Applying distributive law to formula:

Or([And([Or([Not(Predicate(Predicate { name: "big_p", arity: 1,
terms: [Constant("skolem_con2")] })]), And([Predicate
(Predicate { name: "big_q", arity: 1,
terms: [Function { name: "skolem_function_3", arity: 1,
args: [Variolem_function_3", arity: 1, args: [Variable("Y")] }] }))])]),
And([Or([Not(Predicate(Predicate { name: "big_q", arity: 1,
terms: [ { name: "big_s", arity: 1,
terms: [Function { name: "skolem_function_3", arity: 1,
args: [Variable("Y")] }] })]),
And([Predicate(dicate { name: "big_r", arity: 1,
terms: [Constant("skolem_constant_2")] }))])])])
```

The other part of the formula is processed using the function function reverse_distributive_law(). For the clever combination of the individual expressions, the Cartesian product must also be formed here. These processes are now repeated as often as necessary until finally a CNF form is available which contains the following clauses:

```
Converted CNF Formula:
```

```
Clause 1: { ~big_p(skolem_constant_2), big_r(skolem_constant_2),
            ~big_q(skolem_function_3(?Y)), big_s(skolem_function_3(?Y)) }
Clause 2: { big_p(skolem_constant_2), ~big_s(skolem_function_3(?Y)) }
Clause 3: { ~big_r(skolem_constant_2), big_q(skolem_function_3(?Y)) }
Clause 4: { ~big_r(skolem_constant_2), ~big_s(skolem_function_3(?Y)) }
Clause 5: { big_p(skolem_constant_2), big_q(skolem_function_3(?Y)) }
```

The CNF conversion process is now complete and the generated clauses can be passed to the resolution algorithm together with the remaining clauses of the input.

3.4 Resolution algorithm

3.4.1 Implementation for propositional logic inputs

As already explained, I first programmed a resolution prover for propositional logic inputs as part of this bachelor thesis. The input files here already consisted only of clauses in CNF form, which after parsing are passed to the resolution algorithm as a vector (structure in Rust) with a list of all the clauses. The resolution algorithm is kept very simple here. The main resolution function resolution() iterates through the entire clause set of the input. The first clause is initially designated as ”Clause a” and compared with every other clause in the clause set. The aim of the comparisons is to find a complementary literal pair so that the two clauses can be resolved. This check for the existence of a complementary literal pair is carried out using the function find_resolvent() function. This function checks the two clauses to be compared and resolves them if a complementary pair is present. When resolving, the complementary pair is removed and the remaining clause contents are added to a new clause, the resolvent. By storing the resolvent in a hash set, it is also checked for possible duplicates, which are removed before the resolvent is added to the existing clause set. The new resolvent is then added to the existing clause set, provided that there is not yet a clause identical to the resolvent. The avoidance of duplicates is also easily implemented here by using a hash set. All in all, the system iterates through the existing clause set extended by resolvents until no more new resolutions are possible or until an empty clause is found. The functionalities of the resolution algorithm are shown in pseudo code ”Algorithm 9 Main Resolution Algorithm for CNF Formulas”.

Algorithm 9 Main Resolution Algorithm for CNF Formulas

Require: A CNF formula *cnf_formula*
Ensure: Returns a list of clauses or an error if unsatisfiable

```
unprocessed_clauses ← queue from cnf_formula.clauses
                                        ▷ Initializes the queue with CNF clauses
resolved_clauses ← empty hash set   ▷ Tracks clauses that have been processed or resolved
new_clauses ← empty hash set        ▷ Temporarily stores new clauses formed by resolution
while unprocessed_clauses is not empty do   ▷ Continue until all clauses are processed
    clause_a ← pop front from unprocessed_clauses
                                        ▷ Retrieve and remove the next clause for processing
    for each clause_b in unprocessed_clauses do
                                        ▷ Compare clause_a against all remaining clauses
        if clause_a is not clause_b then                ▷ Avoid self-resolution
            resolvent ← FINDRESOLVENT(clause_a, clause_b)   ▷ Attempt to resolve clause_a
and clause_b
            if resolvent is not None then
                                        ▷ Check if a resolvent was successfully formed
                if resolvent.literals is empty then
                                        ▷ Check for an empty clause
                    return Error "Unsatisfiable"
                end if
                if not in resolved_clauses and not in new_clauses then
                        ▷ Ensure resolvent is not already processed or scheduled for processing
                    add resolvent to new_clauses
                                        ▷ Add new resolvent to new clauses
                    add resolvent to resolved_clauses
                                        ▷ Mark resolvent as processed
                end if
                add clause_a and clause_b to resolved_clauses
                                        ▷ Mark both original clauses as processed
            end if
        end if
    end for
    if new_clauses is not empty then
                                        ▷ Check if new clauses were generated in this iteration
        add all from new_clauses to resolved_clauses
                                        ▷ Merge new clauses into resolved clauses for tracking
        add all from new_clauses to unprocessed_clauses
                                        ▷ Save new clauses for further resolution
        clear new_clauses
                                        ▷ Clear new clauses for next loop iteration
    end if
end while
return list of resolved_clauses
                                        ▷ Return all resolved clauses as the result of the algorithm
```

Example. Resolution with propositional logic inputs: Here is another short example, which is also a benchmark in the resolution prover:

```
File: unit.cnf
c sat
p cnf 3 3
1 2 3 0
1 -2 0
-1 0

This input returns the following clause set after the execution
of the resolution algorithm. The input is also satisfiable,
as no empty clause could be generated.

Clause { literals: [Positive("1"), Negative("2")] }
Clause { literals: [Positive("1"), Positive("2"), Positive("3")] }
Clause { literals: [Positive("3")] }
Clause { literals: [Negative("2")] }
Clause { literals: [Positive("3"), Positive("2")] }
Clause { literals: [Negative("1")] }
Clause { literals: [Positive("3"), Positive("1")] }
```

3.4.2 Implementation for first order logic inputs

If the first order logic inputs are available in a suitable CNF clause form, then the resolution algorithm previously explained for propositional logic inputs can basically also be applied to the extended first order logic inputs, but with some new specifications and special features. There is again a main resolution algorithm here, which controls the resolution process. Again, two clauses from the clause set are compared and the find_resolvent() function is called to check for a possible resolution. After a pair of clauses has been resolved, I have added a new check here, which compares the resolvent with the current main clause ”Clause a” again for possible resolvents, in order to bring about a potential empty clause more quickly. After this new check, the clauses are added to the clause set and the next clause is considered as ”Clause a” and checked with the entire clause set for possible resolutions, including the previously created resolvents. So far, there is no significant difference to propositional logic, except that when checking for a complementary pair, not only the name of the predicate is considered and compared, but also the terms contained therein. A special feature that is added when using first order logic inputs in the resolution algorithm is the use of unification. I have used the principle of delayed unification (Bhayat2023) for this, which means that unification is only carried out in the last step if necessary. I have implemented two resolution steps for this. First, resolution is performed as usual and a clause set is returned that contains all clauses that were created in the resolution process as well as the initial input clauses. This clause set is then used for the second resolution step, which I call resolution with unification. The aim here is to check whether further unification steps can be carried out as efficiently as possible and preferably with as few substitutions as possible. When implementing the resolution algorithm for first order logic inputs, it was important for me to find a clear separation of interests and tasks with

regard to the resolution of two clauses, as there can potentially be several ways of resolving two clauses and unification is a much more complex operation, as in the case of a substitution, this must be applied to the entire clause set. I have therefore decided to carry out resolution without unification in the first resolution step, as empty clauses could already be found here. For the purpose of completeness, the second step then uses unification to search for further resolution possibilities. Since the first step of the resolution for first order logic inputs is very similar to the implementation for resolution for propositional logic inputs, I will dispense with the pseudo code at this point, because the main difference here is the unification, which will now be explained in more detail (see "Algorithm 10 Resolution with Unification" and "Algorithm 11 Unify and Resolve Two Clauses"). In the second step, the previously generated clause set is iterated through and further resolvents are found by means of unification. Here, we only look for resolving options that require a substitution, as all other resolving options have already been explored in the first resolving step. Here, the resolution_with_unification() function is the main control function, as it controls the iteration through the clauses and calls the corresponding helper functions for further processing, including the unify_and_resolve() function, which is the pedant to find_resolvent from the first resolution step. Here, two specific clauses are checked for unification possibilities. To do this, it is first determined whether they are a complementary predicate pair. The term content is then compared and checked for a possible substitution. It is also checked here whether the term content is a skolem constant or a skolem function, as these require extra treatment and cannot simply be substituted by any variables as part of the unification. The apply_substitutions function is responsible for the actual substitution. The unify() function finally performs the actual unification and also ensures that this error is avoided by calling the occurs_check function.

Algorithm 10 Resolution with Unification

Require: Vector of clauses *final_clauses*
Ensure: Returns a sorted list of resolved_clauses or an error if unsatisfiable

```
Sort final_clauses by the number of literals and the first predicate name
                                                    ▷ Initial sorting to optimize processing
unprocessed_clauses ← queue from final_clauses
                                                    ▷ Initialize queue for systematic processing
resolved_clauses ← new HashSet
                                                    ▷ Store resolved clauses in HashSet to avoid duplication
while not unprocessed_clauses.is_empty() do
                                                    ▷ Process until all clauses are examined
    clause_a ← pop front from unprocessed_clauses
                                                    ▷ Take the first clause for resolution
    new_clauses ← new HashSet
                                                    ▷ Temporary storage for new clauses formed in this iteration
    for each clause_b in unprocessed_clauses do
        resolvent ← UNIFYANDRESOLVE(clause_a, clause_b)
                                                    ▷ Attempt to unify and resolve clause_a with clause_b
        if resolvent is not None then
            if resolvent.literals is empty then
                return Error "Unsatisfiable with unification"
                                                    ▷ Empty clause found, CNF is unsatisfiable
            end if
            new_clauses.insert(resolvent)
                                                    ▷ Add the new resolventl
        end if
    end for
    insert clause_a into resolved_clauses
                                                    ▷ Move clause_a into resolved clauses
    if not new_clauses.is_empty() then
        extend resolved_clauses with new_clauses
                                                    ▷ Add new clauses to the set of resolved clauses
        extend unprocessed_clauses with new_clauses
                                                    ▷ New clauses need to be processed in future iterations
    end if
end while
final_unified_clauses ← list from resolved_clauses
                                                    ▷ Convert resolved clauses set into list
sort final_unified_clauses by the number of literals
                                                    ▷ Sort the final clauses for further processing
return final_unified_clauses
                                                    ▷ Return the sorted list of resolved clauses
```

Algorithm 11 Unify and Resolve Two Clauses

Require: Two clauses *clause_one* and *clause_two*
Ensure: Returns a new clause if unification is successful, otherwise *None*

```
for each lit_one in clause_one.literals do
    for each lit_two in clause_two.literals do
        if lit_one.predicate() == lit_two.predicate() and lit_one.is_negation_of(lit_two) then
▷ Check if literals can be unified and resolved
            substitutions ← new HashMap
            subst ← UNIFYTERMS(lit_one.get_terms(), lit_two.get_terms(), substitutions)
            if subst is not None and not CONTAINSSKOLEM(substitutions) then
                new_literals ← union of clause_one.literals and clause_two.literals without
lit_one and lit_two
                APPLYSUBSTITUTIONS(new_literals, subst)
                                                    ▷ Apply substitutions to the remaining literals
                return new Clause with new_literals
                                                    ▷ Return the new resolvent
            end if
        end if
    end for
end for
return None
                                                    ▷ Return None if no resolvable unification found
```

4 Evaluation and tests

4.1 Environment

The inputs of both resolution provers were tested in the following environment:

- Operating System Windows 10
- CPU i7 8th generation, 4 cores + hyperthreading
- 16 GB RAM
- Intel(R) UHD Graphics 620
- 256 GB SSD

This software versions were used:

- Visual Studio Code as IDE version 1.91.1
- Rust version 1.72.1
- Pest Parser library version 2.7.5

4.2 Tests and benchmarks

In order to check the correct implementation of the resolution algorithm, SAT competition benchmarks were used for the propositional logic resolution prover, of which some files are satisfiable and some are unsatisfiable. The complexity of the inputs is determined by the structure and number of clauses, which varies in the input files from at least 2 up to 112 clauses. The first order logic input files were taken from the website TPTP.org (Thousands of Problems for Theorem Provers). This is a constantly growing comprehensive library of problems for automated theorem provers, which are all stored in a standardized format (TPTP language format). This also facilitates the design of provers, as they can easily interpret various problems in the library. The grammar of the TPTP format is well described on the website and ensures a standardized representation of the individual problems and problem classes that are used as input for the prover. The exact benchmarks used can be found in the implementation of the respective prover in the Benchmarks folder.

Since the development of a resolution prover for the entire TPTP problem library would definitely go beyond the scope of this work, if possible at all, I have only concentrated on a problem subset, which I have taken from this category: TPTP Benchmarks SYN In my final implementation, the three input files SYN044+1.p, SYN055+1.p and SYN056+1.p are implemented for the use of the resolution prover, as I found these three files the most meaningful for explaining the functionality of my prover. All three input files have the same basic structure: The existing formulas are given in fof format (first order logic formula) and are either axioms or a conjecture. Each input file contains a varying number of axioms and always a conjecture. My CNF converter was designed precisely for the application of CNF conversion to formulas in fof format. Accordingly, other input files can also be processed, provided they also follow this structure. Of course, in addition to these three inputs, many other input files have also contributed to the development of the prover, which can be viewed

in the Git project in earlier versions of the program. Nevertheless, this total quantity of used and suitable input files represents only a small fraction of the TPTP problem library.

In order to be able to understand the process of the resolution algorithm, I have added additional informative logging at the relevant points in the code, which can be activated if required. This allowed me to try out and test various inputs and always analyze the executed steps of the algorithm. I wrote concrete tests when designing the parser, among other things, as the asserted output can be described very clearly here. In addition, I also wrote tests for the resolution algorithm, which receive either satisfiable or unsatisfiable inputs and accordingly check the respective assertion ”satisfiable” or ”unsatisfiable” after the resolution has been carried out. However, hard coding the exact output of the resolution of certain input files as part of a test does not make sense at this point, which is why I did not do this, but only looked at the general output ”satisfiable” or ”unsatisfiable”. Depending on the order of the clause combination or resolvent formation, the output of the resolution can theoretically look different after each run. Especially with an unsatisfiable input, the output can vary depending on how quickly and skillfully an empty clause was detected. The implemented tests for the propositional logic prover can be found under src/lib.rs. For testing the first order logic prover, I decided to use extensive logging, although concrete tests were also written when designing the parser. These can be found directly in parser.rs. I used these tests to check the recognition of the formula structures. However, as this parser did not work completely until the end, I would just like to refer to it at this point.

5 Conclusion

5.1 Objectives achieved: Reflection of the results in the context of the objective of the thesis

In this thesis I developed two resolution provers in the programming language Rust, one for propositional logic inputs and one for first order logic inputs which also corresponds to the objective of this thesis. The main focus was on the development of the resolution prover for FOL inputs, which is based on the other prover, but implements some significant changes and extensions. The challenge here was on the one hand to convert the existing inputs from a first order formula format into a CNF clause form suitable for the resolution algorithm and on the other hand to handle these more complex clauses correctly in the context of resolution. For this purpose, functions were implemented that can perform unification, i.e. resolution by means of substitution, on this FOL input. The principle of delayed unification was applied here to ensure that unification is integrated into the existing resolution process in a regulated manner. With the development of these two provers and the plan to make the program code available to the Rust community in the future, a further contribution has been made to the development of theorem provers in the Rust language. In the process of this project, I was also able to gain insight into many other exciting mechanisms and optimization possibilities, which I will explain at the end as an outlook.

5.2 Challenges and learning outcomes

This project definitely had some challenges, not least because I was new to Rust as a programming language and also because this topic of theorem proving in this depth was new to me. In the

beginning, the development of the parser in particular took up a lot of time and resources, although in the end the FOL parser was not fully functional and a workaround was created. Here I definitely took away the learning that I should not put so many resources into parser development if the actual goal of the work is something else. Another important component is the CNF converter. Although the main goal of this work is the development of a resolution algorithm, the development of the CNF converter ultimately took at least as much time and even made up a significantly larger part of the program code. At this point, one could therefore have looked for FOL inputs that are already available in a CNF structure in order to be able to concentrate mainly on optimization possibilities and additional heuristics in the actual resolution. However, I personally still considered the development of the CNF converter to be important in the context of this thesis, as I was able to deal explicitly with the special features and characteristics of first order logic and therefore gain even deeper insights into theoretical computer science. Furthermore, I would have liked to spend more time on testing and implement more automated tests. Nevertheless, I learned a lot of new knowledge during this thesis, both in terms of methodology and content. I will hopefully be able to apply this new knowledge and experience to future projects and possible continuations of this project.

5.3 Possible extensions

During the development of the two provers, I constantly came across opportunities for improvement and enhancements. I was able to implement some of them straight away, but there are also some things that could be considered as future work. I would like to explain some ideas here. A tautology check could be implemented in the entire resolution process, as clauses that represent a tautology have no use in the context of the resolution and can be removed from the clause set in order to keep them as small as possible. An initial approach has already been implemented for this, but would need to be expanded and is therefore currently marked as a TODO. A further optimization could be made in the CNF converter in the section of the application of the distributive laws, because for significantly larger input clauses it can be a considerable effort to bring the clause into a CNF using distribution mechanisms. At this point, you could therefore consider implementing a Tseytin transformation. Roughly speaking, formulas are first converted into equations and then into clauses. Newly introduced variables represent certain sub formulas. However, it is important to note that the result of this transformation is not an equivalent result to the previous formula, but an equisatisfiable result, which is why the introduction of this transformation must be analyzed in detail for the respective context of the task. There are also a few ideas that can be tried out in the context of resolution to make this algorithm more efficient. For example, mechanisms could be implemented that try to detect a contradiction as efficiently as possible, for example by preferentially processing smaller clauses. In my current implementation, every clause is compared with every other clause in the clause set for the sake of completeness, but suitable heuristics and methodologies could be used to implement a preselection that would possibly lead to an empty clause more quickly, which could save a large number of resolution steps. Another goal that I would like to implement in the future is to finalize the FOL parser so that even more inputs can be entered and processed in a more agile way in the future and do not have to be explicitly hard coded. At the same time, further suitable inputs could be found and extensions created in the code to be able to process even more diverse problem classes from TPTP instead of just first order formulas as is currently the case.

6 Source Code

The complete implementation of both resolution provers can be viewed on GitHub by following this link Git Repository for Resolution Prover in Rust . A current version of Rust must be installed in order to run the resolution prover. After navigating into the source (src) folder the code can be started with ”cargo run”. There is then an interactive prompt on the command line to navigate to the desired input file. After selecting the preferred input, it is processed by the resolution prover and the user receives the result ”satisfiable” or ”unsatisfiable”, together with a list of the resolution steps performed and in the case of ”satisfiable” also a list with the new clause set.

List of Algorithms

List of Tables

References

BACHMAIR, LEO; HARALD GANZINGER; DAVID MCALLESTER; und CHRISTOPHER LYNCH. 2001. *Resolution theorem proving*, 19–99. Elsevier. URL http://dx.doi.org/10.1016/B978-044450813-3/50004-7.

BHAYAT, AHMED; JOHANNES SCHOISSWOHL; und MICHAEL RAWSON. 2023. Superposition with delayed unification. *Automated deduction – cade 29*, hrsg. von Brigitte Pientka und Cesare Tinelli, 23–40. Cham: Springer Nature Switzerland.

GRAF, PETER. 1996. *Term indexing*, *Lecture Notes in Computer Science*, Ausg. 1053. Springer. URL https://doi.org/10.1007/3-540-61040-5.

HARRISON, JOHN. 2009. *Propositional logic*, 25–117. Cambridge University Press.

KEVIN ANDRIAN SANTOSO, OEY; CATHERINE KWEE; WILLIAM CHUA; GHINAA ZAIN NABIILAH; und ROJALI. 2023. Rust's memory safety model: An evaluation of its effectiveness in preventing common vulnerabilities. *Procedia Computer Science* 227.119–127, 8th International Conference on Computer Science and Computational Intelligence (ICCSCI 2023). URL https://www.sciencedirect.com/science/article/pii/S1877050923016757.

KLABNIK, STEVE, und CAROL NICHOLS. 2023. *The rust programming language: 2nd edition*. San Francisco, CA: No Starch Press. https://doc.rust-lang.org/stable/book/title-page.html [Accessed: (30.07.2024)].

KNIGHT, KEVIN. 1989. Unification: a multidisciplinary survey. *ACM Comput. Surv.* 21.93–124. URL https://doi.org/10.1145/62029.62030.

KRÖTZSCH, MARKUS. 2017. Lecture notes: Funktionen und normalformen. https://iccl.inf.tu-dresden.de/w/images/b/b0/TheoLog2017-Vorlesung-17-overlay.pdf [Accessed: (30.07.2024)].

MICROSOFT. 2024. Z3. https://www.microsoft.com/en-us/research/project/z3-3/ [Accessed: (30.07.2024)].

RAUTENBERG, WOLFGANG. 2010. *A concise introduction to mathematical logic*. Springer New York. URL http://dx.doi.org/10.1007/978-1-4419-1221-3.

ROBINSON, J. A. 1965. A machine-oriented logic based on the resolution principle. *J. ACM* 12.23–41. URL https://doi.org/10.1145/321250.321253.

RUSSELL, STUART, und PETER NORVIG. 2009. *Artificial intelligence*. 3 Ed. Upper Saddle River, NJ: Pearson.