RESEARCH INSTITUTE FOR SYMBOLIC COMPUTATION

JOHANNES KEPLER UNIVERSITY LINZ

A Saturation-Based Automated Theorem Prover for RISCAL

Viktoria Langenreither

December 2025

RISC Report Series No. 25-11

ISSN: 2791-4267 (online)

Available at https://doi.org/10.35011/risc.25-11

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Author
Viktoria Langenreither, BSc
11809679

Submission
Research Institute for Symbolic Computation

Thesis Supervisor
A.Univ.-Prof. DI Dr.
Wolfgang Schreiner

December 2025

A Saturation-Based Automated Theorem Prover for RISCAL

Master's Thesis

to confer the academic degree of

Diplom-Ingenieurin

in the Master's Program

Computational Mathematics

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

The aim of this thesis is to integrate a automated theorem prover for first-order logic with equality in the RISC theorem proving (RISCTP) software. The software currently already provides a built-in first-order prover. The MESON prover (model-elimination, subgoal-oriented) follows a goal oriented approach. The idea is to provide a different procedure (knowledge-based) to further extend the functionality of RISCTP, since different prove designs lead to a fast proof search for different examples. One way to achieve this is by using the saturation principle, i. e. the prover generates from a set of first-order clauses in a systematic way all logical consequences until a refutation (empty clause) is found.

In this thesis we give a throughout theoretical representation of different techniques for first-order provers, based on a comprehensive literature research. In accordance to this, we developed the design of our prover. As most state of the art provers, it is based of the give-clause-algorithm using the DISCOUNT loop to organize the proof search. The inference system is based on a variation of the superposition calculus, using the so-called pure superposition for the equality reasoning (all literals represent equalities). To further extend this prover an interface to an external SMT solver is integrated as well as axioms encoding the special theories of integers and arrays. The implementation of this prover is done as an extension of the RISCTP software using the Java programming language. To evaluate the prover, we used examples that were already defined for evaluating the MESON prover.

Kurzfassung

Ziel dieser Arbeit ist die Integration eines automatisierten Theorembeweisers für Prädikatenlogik erster Stufe mit Gleichheit (first-order logic with equality) in die RISC theorem proving Software (RISCTP). Die Software bietet bereits einen integrierten Prädikatenbeweiser erster Stufe. Der MESON Beweiser (model-elimination,subgoal-oriented) verfolgt einen zielorientierten Ansatz. Die Idee besteht darin, eine alternative (wissensbasierte) Strategie bereitzustellen, um die Funktionalität von RISCTP zu erweitern, da unterschiedliche Beweisdesigns zu unterschiedlich schnellen Beweissuchen für verschiedene Beispiele führen. Dies kann unter anderem durch die Anwendung des Sättigungsprinzips (saturation principle) erreicht werden: Der Beweiser generiert systematisch aus einer Menge von Prädikatenklauseln alle logischen Konsequenzen, bis ein Widerspruch (leere Klausel) gefunden wird.

In dieser Arbeit präsentieren wir eine umfassende theoretische Darstellung verschiedener Techniken für Beweiser. Auf Basis der Erkenntnisse dieser umfassenden Literaturrecherche wurde der Beweiser entwickelt. Wie die meisten modernen Beweiser basiert er auf dem sogenannten give-clause-Algorithmus und nutzt die DISCOUNT-Schleife zur Organisation der Beweissuche. Das Inferenzsystem basiert auf einer Variante des Superpositionskalküls. Um dem Beweiser auch den Umgang mit (Un-)Gleichheit zu ermglichen, wird eine Taktik verwendet, bei der alle Literale bewusst zu einer Gleichheit geformt werden. Zur Erweiterung des Beweisers wurde eine Schnittstelle zu einem externen SMT-Solver integriert, um auch spezielle Theorien wie Integer verwenden zu können. Alternativ werden Axiome zur Verfügung gestellt, die die Theorien der Integer und Arrays kodieren und dem Beweisproblem hinzugefügt werden können. Die Implementierung des Beweisers erfolgte als Erweiterung der RISCTP-Software unter Verwendung der Programmiersprache Java. Die Evaluaierung des Beweiser erfolge anhand von Beispielen, mit welchen bereits der MESON Beweisedr evaluiert wurde. Dadurch wird auch die Vergleichbarkeit der Beweiser gewährleistet.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

The majority of the world is nowadays formed by the application of various software systems causing a huge dependency on software products. Hence it is a necessity that the reliability of software increases steadily. This causes problems in quality control techniques like code inspection or software testing, since those cannot deliver the required level of reliability. The only possible way to reach this level is the application of formal methods.

The technique of formal specifications plays an important role in the area of formal methods. Formal specifications of software denotes the method of describing software formally i.e. by a language based on mathematics and logic. Although formal specifications can be a key component for obtaining a reliable software, they are often neglected, because formal specifications are not necessarily easy to define and are often experienced as a burden. To mitigate these difficulties and improve the reputation of formal methods, the RISC Algorithm Language RISCAL [34] was designed.

RISCAL was developed as a specification and verification tool with the goal to simplify the process of formally specifying problems and verifying the correctness of algorithms (software verification). It is based on a variation of first-order predicate logic and it supports the specification of types, functions, predicates, theorems, and procedures. Parametrized entities can be enriched with invariants, pre- and postconditions. In RISCAL all variables are restricted to finite domains, which enables effective evaluation of all specifications and annotations through automated model checking, by executing the programa and checking all possible inputs for the annotations. Over time, RISCAL has been linked to the RISCTP theorem proving interface, which provides access to external SMT solvers (as for example Z3 [10], cvc5[7] or Vampire [45]) and automated theorem provers. That allows to also prove the validity of these formulas for arbitrary (all possible) values of the model parameters.

The aim of developing RISCTP was to extend the RISCAL model checker with theorem proving capabilities. RISCTP complements RISCAL by offering a dedicated language and tools for specifying and solving first-order logic proof problems. The RISCTP language is a simplified version of the RISCAL language (closely following the SMT-LIB standard, extended with additional expressive features). RISCTP supports a wide range of data types and theories (such as algebraic data types, integer arithmetic and functional arrays) and applies systematic preprocessing steps to prepare proof problem for automated reasoning. Furthermore, RISCTP already has a built-in prover for first-order logic called MESON (model-elimination, subgoal-oriented). By such an internal prover it may, for example become possible to make proofs and their attempts transparent, for instance in order to make it easier to understand why a proof attempt has failed.

One of the main goal of this thesis is to extend the RISCTP software by an additional prover for first-order logic using a new proof strategy. The idea behind this is to create different options, since the results for different examples vary depending on the proof-technique. Currently, only the MESON prover is available in RISCTP, which is a goal oriented proof method. The focus in this thesis is to design and implement a knowledge oriented prover. There are different approaches to design an efficient first-order prover and there are already several good ones available, e.g., Vampire [45] or the E prover [44]. Provers might be based on the saturation principle i. e. the prover generates all logical consequences from a set of first-order clauses (in a systematic way) until the unsatisfiability of the clause set can be shown. The algorithm continues until a refutation, i. e. a contradiction (the empty clause), is found. As most state of the art theorem provers, also Vampire and E use the so-called resolution method [13] [27]. The central piece of the propositional procedure is the resolution rule, which derives from two clauses $p \vee C_1$ and $\neg p \vee C_2$ the resolvent $C_1 \vee C_2$. Additionally, the first-order resolution principle uses unification to resolve the most general forms of the clause directly [29]. One way to ensure theorem proving involving equality is to combine this resolution method with paramodulation [13]. This simple and natural approach introduced by Robinson and Wos [28] is an inference rule that extends the classical equality substitution rule. Another possibility for handling equality is the superposition calculus. This was first introduced by Bachmair and Ganzinger in 1991 [2] and represents a specialization of the resolution method and paramodulation. One big advantage is, that this technique leads to a smaller search space and its compatibility with a wide variety of redundancy elimination criteria. Additionally it is possible to integrate reasoning capabilities for special theories (as for example integers or arrays) in the prover e.g. with an automatization of the theory [16] or a SMT solver [5].

Building on that, the thesis contains a thorough, systematic representation of the theoretical background. This includes a description of first-order logic with equality (definition and notions used in the thesis) and the resolution calculus. To enable the reasoning with equality, the inference rule of paramodulation is investigated. The theory of paramodulation can be further extended to superposition inferences. Different options to create inference systems are shown. The goal is to create a refutation complete system, therefore an inference system consisting of two superposition rules and equality resolution is defined and it is sketched how the system can be proven to be refutation complete. Additionally, axioms for encoding the special theories of integers and arrays are defined.

Based on these theoretic results, we extracted the substantial and most promising parts for designing a superposition prover. The prover is designed as a refutation-based, saturation calculus using the given-clause-algorithm and the so-called DISCOUNT loop as search organization (systematic way to generate clauses). The inference rules are explicitly defined and generating inferences are restricted to maximal literals only. The equality reasoning is done by intentionally making every literal to become an (in)equality, by adding “equals true” respectively “not equals true” (like the E theorem prover [38]).

This designed prover was then implemented as an extension to RISCTP using the Java programming language. The prover got evaluated using the same test examples as for the evaluation of the already included MESON prover. This enables comparisons between these two provers.

This software is going to get integrated in RISCTP in the very near future and can then be downloaded from https://www3.risc.jku.at/research/formal/software/RISCTP/.

This thesis is organized as follows: Chapter 2 gives an overview of the corresponding software RISCAL and RISCTP as well as the historical development of first-order logic proving techniques. Chapter 3 introduces the theoretical background of first-order logic with equality and explains important tools for first-order logic theorem proving. In Chapter 4 the design of the prover along with its implementation is presented. Subsequently, a demonstration is provided together with benchmarks in Chapter 5. The thesis is finished with a short Chapter 6 about possible future work on this topic and with a conclusion (Chapter 7).

Chapter 2

State of the Art

This chapter discusses the RISC Algorithm Language RISCAL and the RISCTP theorem proving interface as well as the history and development of proving techniques for first-order logic (including equality). Specifically, we give an overview about the saturation principle and the resolution calculus, options for reasoning about equality, and decision procedures for special theories.

2.1 RISCAL and RISCTP

Formally verifying computer programs can be a key component for obtaining reliable software. Nevertheless this is often neglected, because formal specifications are not necessarily easy to define and are often experienced as a burden. Especially for students trying to learn this technique it can be a very frustrating process. To successfully overcome this burden and improve the reputation of formal methods, the RISC Algorithm Language RISCAL [34] was designed. RISCAL consists of a specification language and an associated software system essentially intended to simplify the process of formally specifying problems and verifying the correctness of algorithms with respect to these specifications. The software is implemented in Java and it is freely available as open source under the GNU General Public License at [34].

The RISCAL specification language is based on a statically typed variant of first-order predicate logic, designed to develop mathematical theories and to enable the specification of algorithms [37]. RISCAL specifications consist of declarations of types, constants, functions, predicates, theorems and procedures [46]. Parametrized entities may be annotated by invariants, pre- and postconditions. The basic idea behind RISCAL is to use automatic techniques to detect problems in the program, the specification or its annotations, which would prohibit a successful verification [46]. This is done before an actual attempt to prove the verification conditions. In order to achieve this, in RISCAL all program variables and mathematical variables are restricted to finite types, i.e., all types have an upper bound specified by a formal model parameter, which can be instantiated with any suitable value. This enables the system to effectively evaluate all specifications and annotations during an execution of the program and to check the annotations for all possible inputs [46] [33].

The RISCAL software has always (since version 1) supported model checking of programs (for finite domain instances) by doing a runtime assertion check of all possible executions. However, this only ensures annotations that are not too strong; it is still possible that these annotations are too weak [46]. Version 2 solves this problem by including a verification condition generator. This derives certain verification conditions from the annotations. The validity of those ensures the correctness of a program. These conditions are formulas, therefore it is possible to check them in a finite model by the RISCAL model checker [46]. Additionally there is a translation of the RISCAL language to the SMT-LIB logic QF_UFBV of unquantified formulas over bitvectors with uninterpreted sort and function symbols (by Franz-Xaver Reichel [25]). In Version 3 the specification language got extended by the concept of (finite) state transition systems, whose states are updated by repeatedly (nondeterministically) executing atomic actions. Furthermore, there exists a model checker that allows to verify system executions specified in LTL (by Ágoston Sütő [43]). Recently, RISCAL was linked to the RISCTP theorem proving interface [35] that provides connections to several SMT solvers and external theorem provers (for example Z3, cvc5 or Vampire [45]) that allow to also prove the validity of these formulas for arbitrary (all possible) values of the model parameters (version 4). For more information see [46] and [33].

The RISCTP Theorem Proving Interface is included in RISCAL to provide theorem proving capabilities. RISCTP consists of two parts: a language for specifying proof problems and an associated software for solving them [36]. As RISCAL, the RISCTP software is based on Java; it provides an API for the integration of other software [35]. Since RISCTP interacts with external SMT-Solvers and theorem provers (Z3, cvc5 and Vampire), it is designed after the language of the SMT-LIB standard [3]. Also RISCTP is open source and freely available at [35]. Among other features, the software as well provides the following concepts: a typed variant of first order-logic, algebraic data types, and the special theories of functional arrays with extensionality and integer arithmetic (SMT-LIB theories “ArrayEx” and “Ints”) [36]. The RISCTP language intends to simplify the connection of specification and verification systems to theorem provers. Nevertheless, RISCTP provides several more features that are not implemented in SMT-LIB. That includes overloading function names, subtypes contained by formulas, tuples with a generic tuple type constructor and choose expressions [36]. The syntax of RISCTP is similar to that of the RISCAL language, which is near to the traditional mathematical and logical one and therefore enables the user to write down proof problems in a very practical way [36].

RISCTP provides several different declarations — types, constants, functions, predicates, axioms and theorems. These declaration types can be summed up into the three categories of types, functions and formulas; constants and predicates are considered as functions, respectively, functions without any arguments or functions with result type Bool. For all types, RISCTP supports the binary predicates `=` and `~=` which denote “equality” and “inequality”. In contrast to the first-order logic and the most modern theorem proving languages, which distinguish between “terms” and “formulas”, the specific RISCTP language has only one category of “expressions”. RISCTP offers different type constructions: declared types, defined types, subtypes, Booleans, integers, maps, tuples and algebraic datatypes. See [36] for a detailed description.

When RISCTP is applied to a proof problem, the system first derives an internal representation for proceeding in several steps, starting with type checking the specification. To guarantee that the specification is well-formed, “type-checking theorems” might be automatically generated by the system; their validity has to be proven at some point. After the type checking process all overloaded functions and predicates get an internally unique name and subtypes get replaced by the defining types (predicate parameters and quantified variables are constrained by the subtype predicates) [36]. If there occurs some choose expression, it gets replaced by a new so-called “choice function” (the result is not defined but it is constrained by an axiom). The last step is called specification pruning: axioms get deleted until only those items are left in the specification that are significant for proving the indicated theorems. All those steps are required for preparing the proof problem for the proof method `smt` (satisfiability modulo theories) which is based on the SMT-LIB language and makes usage of theories for integer numbers and functional arrays with equality. A detailed description of this method can be found in section 3.3 in the user manual [36]. For the first-order logic proof method `fol`, the system has to use all the declarations, definitions and axioms that have been generated during the type-checking process to explicitly define the theories of maps, tuples and algebraic data types. It is assumed that there is built-in knowledge of the theories of equality and integer arithmetic. For those objects only declarations get generated (no axioms and definitions). Currently only the so-called *MESON* prover is implemented yet. It is based on a combination of of multiple proof strategies, but its core is mainly based on the MESON proof strategy (model-elimination, subgoal-oriented). The idea is to apply backward reasoning to repeatedly achieve simpler subgoals (from the goal to be proven). This process is repeated, until the subgoals are simple enough to be proven from the available knowledge. A detailed explanation on how the proof procedure works is given in Chapter 3 of [36]. Additionally, the prover is able to reason with equality and special theories such as integers, arrays and algebraic data types and tuples.

For more detailed information on RISCTP with concrete examples we refer to the user manual [36] which can be downloaded from the RISCTP home page [35].

2.2 Resolution and Saturation

There are different approaches to design an efficient first-order prover and there are already several good ones available, e.g., Vampire [45] or the E prover [44]. Provers might be based on the *saturation principle*, i.e., the prover generates from a set of first-order clauses, that represents the negation of the formula to be proved, all logical consequences, in a systematic way, until it is possible to show the unsatisfiability of the clause set. Many of these theorem provers use the so-called *resolution method* [27] [13].

The starting point of this method is the propositional resolution rule, which derives from two clauses $p \lor C_1$ and $\neg p \lor C_2$ the resolvent $C_1 \lor C_2$. This is also called *ground resolution* which was introduced by Martin Davis and Hilary Putnam in 1960 [8]. In fact, it is possible to derive the empty clause by just applying the resolution rule exhaustively to an unsatisfiable set of clauses, resolving on each literal in turn. Even though logically not necessary, some improvements are useful for the efficiency (chapter 3.11 of [13]).

However, to apply ground resolution in first-order logic, appropriate instances of the first-order clauses have to be generated. A better idea than “guessing” appropriate instances is to apply unification. The first-order resolution principle of Robinson (1965) applies unification to enable resolving the most general forms of the clauses directly [29]: suppose there are two clauses $C[x_1, \ldots, x_n] = \ldots \vee P(s_1, \ldots, s_m) \vee \ldots$ and $D[y_1, \ldots, x_n] = \ldots \vee \neg P(s'_1, \ldots, s'_m) \vee \ldots$. Then they have an instance to which propositional resolution can be applied if an appropriate ground instantiation θ is applied to unify the set $S = \{(s_i, s'_i) \mid i = 1, \ldots, m\}$. Therefore also a most general unifier (MGU) of S may be used instead of θ.

However, one problem is that, if resolution is performed on the instantiated clauses, the original result might not be obtained by a further instantiation. The reason is that the MGU might be too general and therefore it prevents certain literals to become identified, but this identification may be essential for the propositional proof to succeed. This problem can be solved by an additional “factoring” rule that unifies some subsets of the literals in the same clause — the notional ground instance may thus identity them (“lifting lemma”) (chapter 3.11 of [13]).

In general the resolution calculus introduced by Robinson can be described by the two inference rules sketched above, “*(binary) resolution*” and “*(positive) factoring*” (chapter 2 of [27]):

$$\frac{C \vee A \qquad D \vee \neg B}{(C \vee D)\sigma} \text{ (binary) resolution}$$

$$\frac{C \vee A \vee B}{(C \vee A)\sigma} \text{ (positive) factoring}$$

where σ is the most general unifier of the atomic formulas A and B. It is also possible to combine the two inference rule into a single one. This is easier to implement and restricts the formation of factors. A disadvantage is, that it often avoids the recomputation of factors for many different resolutions, because all factors are generated separately. This proving method is refutation complete, i.e., a contradiction (the empty clause) can be deduced from any unsatifiable set of clauses. Proofs for that can be found for example in chapter 2 of [27] or in chapter 3.11 of [13].

Unfortunately, is not enough for automatic theorem proving to design a sound and complete inference system . It is important to understand how to organize the search for a refutation. In particular, the problem occurred that the bare resolution method can quickly generate large amounts of clauses without reaching a proof. One of the most efficient resolution strategies is *ordered resolution*. Assume that there is a partial ordering on terms and literals. Based on that it is possible to define ordered inferences with some special properties: a subset of the literals is marked as maximal, in case of ground clauses the greatest literal is marked as maximal (then the order is total) and the inference rule is restricted it a way, that they only apply to maximal literals (chapter 2 of [27]). A further improvement of the inference system is achieved by defining a selection function. Intuitively this works in the following way: If there is at least one selected literal in a clause, compute only inferences that involve a selected literal and if a clause has no selected literal, only those inferences get computed that involve a maximal literal (chapter

2 of [27]). An early attempt to reduce the search space was for example due to Robinson in 1965 — the so-called *hyperresolution* [30]. Instead of eliminating (negative) selected literals in a clause successively, this method introduces a single inference rule to combine the steps (chapter 2.6 of [27]). Further improvements and alternatives can for example be found in chapter 2 of [27].

The most recent solution to effectively organize the proof search (and "minimize" the search space) is the so-called *saturation principle*. Saturation denotes the concept laying behind proof-search algorithms using the resolution and superposition inference system [16]. Searching for a contradiction proceeds by saturating the given set of clauses (this means the inference rules get applied systematically and exhaustively). This method helps to solve the problems described in the paragraphs above. Sadly, a straightforward saturation which just exhaustively applies inference rules to previously derived clauses could still easily produce many thousands of clauses without reaching a poof and is therefore hopelessly inefficient. Nevertheless, generating all possible clauses is the key idea of saturation-based theorem proving. By using a powerful concept for redundandy elimination and a good saturation algorithm, this concept can be powerful [16].

Saturation-based theorem proving in its modern form was invented by John Alan Robinson in 1965 [29], when he introduced the resolution calculus. In general, saturation-based theorem proving refers to a process with two levels of data structure manipulation — the level of deduction and the level of theorem proving derivations [27]. At the level of deduction new formulas are generated by (systematically and exhaustively) applying specified inference rules (resolution and superposition inference systems) with the goal of obtaining a contradiction (respectively the empty clause). Additionally, some analysis is done on the current set of formulas to identify the most promising inference rules to apply next [16]. At the second level, the formulas represent the logical information that is actually available to the prover at this moment and is used to determine further proving steps. Soon the practical experience has shown that it is more efficient to use "global" techniques for redundancy elimination and refinements, so on the current collection of formulas, instead of performing refinements on the deductive inferences. In general a good strategy for selecting the inference is a key component for an efficient saturation algorithm. To preserve the completeness the selection process has to be fair, i.e. every possible inference has to be selected at some step of the algorithm (see [16] for more details).

There exist many different saturation algorithms, e.g., the Otter saturation algorithm [26], the limited resource strategy algorithm or the Discount algorithm. Those three are for example used in the successful first-order theorem prover Vampire [16]). All three of them belong to the family of so-called *given clause algorithms* which achieve fairness by using clause selection for implementing inference selection. These kind of algorithms differ between clauses previously selected for inferences, called active clauses, and those not previously selected (waiting to be selected), denoted as passive ones.

- Any saturation algorithm that uses passive clauses for simplification is called an *Otter saturation algorithm* (named after the theorem prover Otter [19]), which was designed from big research results in resolution theorem proving in Argonne National Laboratory. One interesting property is, that if the initial set of clauses is maximally simplified (with respect to itself), then the set of persistent clauses (all active and passive clauses) is also maximally simplified (with respect to itself) when starting a new iteration. A typical behaviour of such algorithms is also that only in a few seconds the total number of clauses can get very big, but the percentage of active clauses is small and decreases steadily [26].

- The *limited resource strategy algorithm* is a variation of the Otter saturation algorithm and was introduced in one of the early versions of Vampire [45]. It can only be used when the user sets a time limit. The main idea is to attempt, finding passive and unprocessed clauses wiht no chance of getting selected (regarding the time limit) and discard them [16]. The clause selection implemented in Vampire is based on the two parameters "age" and "weight", favouring older and lighter clauses. Important is the fundamental difference to redundancy. Redundant clauses are removed without compromising completeness, while "unreachable" clauses only make sense with regards to a certain time limit. For the identification of these clauses, Vampire measures the time spent to process one clause (usually this time increases, since the sets of active and passive clauses are growing) and occasionally estimates how the proof search pace develops towards the time limit [16]. This algorithm is potentially the best saturation algorithm available in Vampire, although Vampire is the only theorem prover that implements this algorithm so far [16]. Since the number of passive clauses is typically much larger than the active ones the inference pace may become dominated by simplifying passive clauses.
- The *Discount algorithm* solves this problem by making some passive clauses truly passive by not using them for the simplifying process. One problem that occurs with this strategy is that simplifying inferences between a passive clause and a new one performed by the Otter algorithm may produce a valuable clause, which the Discount algorithm might not find. This strategy was first implemented in the theorem prover Discount and is now for example also used by the theorem provers Vampire [45], E [44] and Waldmeister.

Even tough the most complicated part of the proof search is the use of the inference system (as resolution for example), preprocessing the input formulas is also very important. This is fundamental when the input problem contains deep formulas or a large number of formulas (might be the key to solve this problem). Useful preprocessing steps are (incomplete list) [16]: Simplify the formula if is contains any $\top$ or $\bot$, apply so-called pure predicate elimination, Skolemization, replace the equality axioms and delete tautologies.

2.3 Reasoning About Equality

The equality predicate in first-order logic can be handled in many different ways. One rather easy possibility is to modify the input formulas by adding equality axioms (chapter 9.3 of [11]). Any *normal* interpretation (i.e., "equality means equality") has to satisfy the axioms reflexivity, symmetry and transitivity, asserting that equality is an equivalence relation [13]:

1. $\forall x.\ x = x$

2. $\forall x, y.\ x = y \Rightarrow y = x$
3. $\forall x, y, z.\ x = y \wedge y = z \Rightarrow x = z$

Additionally there are also axiom schemes asserting congruence for each n-ary function f as well as each n-ary predicate p, which have to be satisfied [13]:

4. $\forall x_1, \ldots, x_n, y_1, \ldots, y_n,.\ x_1 = y_1 \wedge \ldots \wedge x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots y_n)$
5. $\forall x_1, \ldots, x_n, y_1, \ldots, y_n,.\ x_1 = y_1 \wedge \ldots \wedge x_n = y_n \Rightarrow p(x_1, \ldots, x_n) \Leftrightarrow p(y_1, \ldots y_n)$

Remark. There is one instance of the schemes for every function symbol f and every predicate symbol p. This approach has several drawbacks — every time equality is used, axioms have to be provided. This generates a large search space and the method is therefore hopelessly inefficient. See chapter 4.1 of [13] for more information.

An approach that gets rid of the equivalence axioms $1-3$ is called *equivalence elimination* (chapter 4.8 of [13]). It can be shown that a binary relation R is an equivalence relation if and only if it satisfies the axiom

$$\forall x, y.\ R(x, y) \Leftrightarrow R^*[x, y]$$

where $R^*[x, y]$ denotes the formula $(\forall z.\ R(x, z) \Rightarrow R(y, z))$. Thus we can replace each instance of $R(s, t)$ in a formula by $R^*[s, t]$; then the corresponding axioms are no longer needed (chapter 4.8 of [13]). This approach got generalized in 1994 by Ohlbach, Gabbay and Plaisted [20].

An earlier method for equivalence elimination due to Brand in 1975 (chapter 4.8 of [13]) similarly eliminates the symmetry and transitivity axioms but keeps the reflexivity axiom. This has the advantage that the expensive transformation may only get applied to positive occurrences of $R(s, t)$, while negative occurrences $\neg R(u, v)$ can be left alone. With R being the equality relation and a formula in clausal form, this way of eliminating symmetry and transitivity is called Brand's *S-modification* respectively *T-modification* [13]. Furthermore, Brand developed the idea of eliminating the congruence axioms by repeatedly pulling out non-variable immediate subterms t of functions and predicate symbols (other than equality). To do this, the following equivalences are of use:

1. $f(\ldots, t, \ldots) = s \Leftrightarrow \forall w.\ t = w \Rightarrow f(\ldots, w, \ldots) = s$
2. $s = f(\ldots, t, \ldots) = s \Leftrightarrow \forall w.\ t = w \Rightarrow s = f(\ldots, w, \ldots)$
3. $P(\ldots, t, \ldots) \Leftrightarrow \forall w.\ t = w \Rightarrow P(\ldots, w, \ldots)$

This process is repeated until the function symbols and predicate symbols only occur as arguments to the equality predicate. The method is called *Brand's E-modification* (chapter 4.8 of [13]).

A quite different possibility for handling equality is to equip one of the existing first-order theorem proving techniques with additional rules for equality. Wos, Robinson, Carson and Shalla introduced in 1967 the first equality-based inference rule called *demodulation* [47]. The core idea is to use unit equality clauses (e.g. $x+0 = x$) as rewrite rules to simplify other clauses. Since this method is not complete, a more general rule called *paramodulation* was invented later in 1969 by G. Robinson and Wos [28]. Speaking superficially, this is a simple and natural approach which extends the classical equality substitution rule. Together with the standard resolution rule, paramodulation results in a theoretically (refutation) complete method for reasoning with equality. Proofs for the completeness can for example be found in [28] or in chapter 4.9 of [13]. One idea to prove refutation completeness of paramodulation is to add the reflexivity axiom (definitely necessary) and all functional reflexive axioms of the form $f(x_1, \ldots, x_n)$ (one for each function symbol f). For this proof idea the following observation is used: a hyperresolution proof (assuming the equality axioms) can be imitated by resolution and paramodulation (together with the functional reflexive axioms) [13]. Only after a long time it became clear, that simple reflexivity $x = x$ is enough to ensure refutation completeness. Brand presented in 1975 [6] a very similar argument, based on his equality transformations (described above). Additionally to the sufficiency of simple reflexivity he showed that paramodulation can be restricted in many other ways as well without loosing refutation completeness.

The inference rule is as follows:

$$\frac{C \vee s = t \qquad D \vee P[s']}{(C \vee D \vee P[t])\sigma} \text{ Paramodulation}$$

Here σ denotes the most general unifier of s and the indicated term instance s' and $s = t$ may also mean $t = s$ (chapter 4.9 of [13]). This rule represents a generalization of rewriting i.e. rewriting may be applied in both directions, equations can also be used for rewriting if they occur in a disjunction with other literals and full unification is used. It is possible to show that paramodulation together with resolution and factoring is even refutation complete, but it is necessary to add at least the reflexivity axiom to the input problem. It is even possible to show refutation completeness if functional reflexive axioms of the form $f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$ for each function symbol f get added too (chapter 4.9 of [13]). This approach for handling equality is much more efficient than adding equality axioms, even in its initial form. Over time some significant refinements have been made as for example introducing ordering notions from term rewriting (chapter 4.9 of [13]). For more information, see also chapter 7 of [27].

Another possibility, recognized as one of the most powerful for proof problems with equality, is *superposition.* This was first introduced by Bachmair and Ganzinger in 1991 [2] and represents a specialization of the resolution method and paramodulation. The superposition calculus consists of an inference system (respectively a family of systems) for first-order logic with equality, called the superposition inference system. This system is parametrised by a simplification ordering and a selection function [2]. Such ordering have the advantage, that they formalise a notion for a clause "being simpler" than another one. An example of a simplification ordering is the so-called *Knuth-Bendix ordering.* The selection function selects a non-empty subset of literals in every non-empty clause (we underline selected literals). One possibility, sometimes used in the superposition calculus, is to select all maximal literals with respect to the simplification ordering. All the inference rules (in the superposition inference system) can in general be divided in generating and simplifying ones. The inference system extends the rules of resolution and factoring by the three rules of equality resolution, equality factoring, and superposition given below [16]:

$$\frac{\underline{l = r} \vee C_1 \qquad \underline{L[s]} \vee C_2}{(L[r] \vee C_1 \vee C_2)\sigma} \qquad \frac{\underline{l = r} \vee C_1 \qquad \underline{t[s] = t'} \vee C_2}{(t[r] = t' \vee C_1 \vee C_2)\sigma} \qquad \frac{\underline{l = r} \vee C_1 \qquad \underline{t[s] \neq t'} \vee C_2}{(t[r] \neq t' \vee C_1 \vee C_2)\sigma}$$

Here σ is a most general unifier of l and s, s is not a variable, $r\sigma \not\geq l\sigma$ ($\geq$ is the fixed simplification ordering of the inference system), $L[s]$ is not an equality literal, and $t'\sigma \not\geq t[s]\sigma$.

The superposition inference system is sound and refutationally complete, if the selection function is well-behaved, i.e., it selects either a negative literal or all maximal literals [2]. Soundness denotes the property, that if the empty clause can be derived from an input set S, then S is unsatisfiable. Conversely, refutationally complete means that from an unsatisfiable set the empty clause is definitely derivable ([2]). One big advantage is, that this technique leads to a smaller search space and its compatible with a wide variety of redundancy elimination criteria. For example the E theorem prover [44] is a successful superposition-based prover. For a detailed description we refer to chapter 2 and chapter 7 of [27].

2.4 Decision Procedures for Special Theories

Standard theorem proving methods such as resolution are good at processing quantifiers, but have essentially no support for theory reasoning (such as integers, reals, arrays, lists, . . .). It is quite a hard task to combine first-order theorem proving with special theories. One rather easy way of combining first-order logic with theories is by adding a first-order (potentially incomplete) axiomatisation of the theory. These special defined axioms have to be additionally added to the input formulas. Special theories available in Vampire (integrated with this approach) are for example the integers, reals and arrays [16].

Another way of supporting special theories is integrating SMT (Satisfiability Modulo Theory) solvers in the prover [9]. In SMT only certain/special fragments of first-order logic are used, obtained by constraining them syntactically and/or semantically; these fragments are denoted as *theories*. For example typically only formulae without quantifiers are considered. There are a lot of different theories that are actually used in practice, some example may e.g. be found on the webpage [3]. To better understand SMT, we discuss two theories, which are of particular interest:

- Linear Integer Arithmetic (“Ints”): Here only interpretations are considered that use the domain of integers and assign the usual semantic meaning to the addition and subtraction function as well as the equality, less-equal and less predicates. If further restricted, only to those formulae without quantifiers, this theory is called Quantifier Free Linear Integer Arithmetic — which is even decidable [5].
- Functional Arrays with extensionality (“ArrayEx”): Here the basic understanding of an array is described. If a value v gets written to position i of an array a, afterwards reading position i of a should return the value v. Also this theory may be restricted further to formulas theory without quantifiers which makes the theory decidable [5].

The problem of SMT is highly related to satisfiability in first-order logic. In first-order logic, arbitrary interpretations are considered, while regarding SMT only specific interpretations matter concerning the satisfiability. This gets clear when considering the example $1 + 1 < 2$. If the usual mathematical understanding is assumed, then the formula does not hold, obviously. But if $<$ gets interpreted as the $\geq$ predicate and the other symbols get the usual meaning, then the example is satisfiable. Considering the theory of integers, the formula is again unsatisfiable (modulo this theory), since all symbols have the expected meaning. This small example gives already an insight into why satisfiability modulo theories can be of interest.

Unfortunately, showing the satisfiability (or validity) of a first-order formula is quite difficult, since the problem of satisfiability is undecidable (there exists no procedure to detect if an arbitrary first-order formula is satisfiable) and the problem of validity is semi-decidable (there are procedures that determine if a formula is valid, but this process might not terminate for invalid ones) [5]. On the contrary, the satisfiability problem of SMT is decidable for several theories. Decidable theories are for example the theory of fixed size bit vectors or the theory of linear integer arithmetic. But not all theories are decidable, for example the theory of arrays is not (only the quantifier free fragment is) [17].

Some state of the art SMT solvers are for example Z3 [10], CVC5 [7] or Boolector [4]. Z3 (also integrated in RISCAL [25]) was developed by Nikolaj Bjørner together with others at Microsoft Research. The solver supports various logics as well as different input languages (including SMT-LIB). Additionally it provides access to some programming languages (like C++, Java and Python) via an API.

The Satisfiability Modulo Theory Library (SMT-LIB) is an international initiative founded in 2003 by Silvio Ranise and Cesare Tinelli [3]. The main goal is to facilitate the research and development in SMT. In particular, SMT-LIB provides standard descriptions of background theories used by SMT solvers and a common, standardised language for the input and output of SMT solvers.

Since 2005 there exists the Satisfiability Modulo Theories Competition (SMT-COMP) [41], an annual competition of SMT solvers, based on benchmarks of the library. The good acceptance of SMT-LIB becomes apparent trough the numerous solvers supporting SMT-LIB and the large number of participants at the SMT-COMP.

Chapter 3

First-Order Logic With Equality

In this chapter we are going to introduce the terminology and basic notions on which our prover is based. Additionally we discuss the notions of inference systems and proofs as well as orderings and selection functions. We explain important tools for first-order theorem proving such as Skolemization and unification. As a main part, we explain the resolution calculus and different strategies for reasoning about equality. Here we focus on paramodulation and superposition calculi. On top we discuss the refutation completeness of these techniques. Lastly, we name different methods to integrate special theories such as integers or arrays into a first-order logic theorem prover.

3.1 Basic Notions

In this thesis we consider classical first-order predicate logic with equality. We allow all standard logical connectives ($\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$) and quantifiers ($\forall, \exists$) in the language. Additionally we assume that the language contains the propositional constants $\top$ (always true) and $\bot$ (always false). Together with disjoint, countable sets of variables, function symbols and predicate symbols this is referred as the *first-order alphabet*. In our case, the alphabet contains in addition the binary predicate symbol $=$. Throughout the thesis, we denote variables by x, y, z; constants by a, b, c, d; function symbols by f, g, h and predicate symbols by p, q, r. We mainly try to follow the notions and definitions from [16].

The language of first-order logic consists of terms and formulae:

Definition 3.1.1. *Terms* are defined by the BNF grammar rule $t \coloneqq x \mid f(t_1, \dots, t_n)$ for a n-ary function symbol f, a variable x, and $n \geq 0$. A term without variables is called *ground term*. We denote terms by l, r, s, t (chapter 2 of [27]).

Definition 3.1.2. *First-order formulas* are defined by $F \coloneqq \bot \mid \top \mid p(t_1, \dots, t_n) \mid (\neg F) \mid (F_1 \wedge F_2) \mid (F_1 \vee F_2) \mid (F_1 \Rightarrow F_2) \mid (F_1 \Leftrightarrow F_2) \mid \exists x.F \mid \forall x.F$ with $n \geq 0$, a n-ary predicate symbol p, and a function symbol F. We denote formulas by $A, B, C, \dots$ (chapter 2 of [27]).

Terms are intended to denote objects. We call $\top, \bot$ and $p(t_1, \dots, t_n)$ *atoms* or *atomic formulas*. Atoms and negated atoms are also called *literals*. Everything else is called a *compound formula*. Any atom of the form $s = t$ is called an *equality*; by $s \neq t$ we denote the formula $\neg(s = t)$. Literals that are atoms are called *positive*, whereas literals that are negated atoms are called *negative*. We call a predicate symbol, a function symbol or a constant a *symbol*; variables are not symbols as well as equality is not (instead we consider the equality as part of the language) (chapter 3 of [16]).

Definition 3.1.3. Let $L_1, \ldots, L_n$ be literals for $n \geq 0$. Then the disjunction $L_1 \vee \ldots \vee L_n$ is called a *clause*. For $n = 0$ this is called the *empty clause*, denoted by $\square$ (sometimes also $\top$). The empty clause is always false. A clause that contains only one literal ($n = 1$) is called *unit clause*. If a unit clause contains the equality predicate = it is also called *equality literal* (chapter 3 of [16]). An alternative definition of a *clause* is, that it is a pair of finite multisets of equations Γ (also called the *antecedent*) and Δ (also called the *succedent*). We denote this by $\Gamma \rightarrow \Delta$. A clause is called a *Horn clause* if at least one equation is contained in Δ (chapter 7.2 of [27]).

Remark. A *multiset* over a set S is a function $F : S \rightarrow \mathbb{N}$. The union and the intersection of multisets are defined as usual (chapter 7.2 of [27]).

Definition 3.1.4. Let F be a formula and x be a variable. A formula is called *universal* (respectively *existential*) if it is of the form $(\forall x.\ F)$ (respectively $(\exists x.\ F)$). We call x the *quantified variable*, F the *scope of the quantifier* and every occurrence of x in F is *bound* (by the respective quantifier). An occurrence of a variable x in the formula F which is not bound by any quantifier is called *free* (also x occurs free in F). A formula without occurrences of free variables is called *closed* (sometimes we also call it a *sentence*). Otherwise it is called *open* (chapter 3 of [16] and chapter 3 of [13]).

In order to be able to drop excessive parentheses we apply the following binding rules:

$$\neg > \wedge > \vee > \Rightarrow > \Leftrightarrow > \{\exists, \forall\}$$

The arguments associate to the right for the same connective. If we do not write parentheses, the scope of a quantifier ranges till the end of the formula. By writing $E(s)$ we mean an expression E with an occurrence of a term s. If we want to denote the expression obtained from $E(s)$ by replacing all occurrences of s by t we then write $E(t)$.

To obtain the semantics (meaning) of first-order logic formulas, the function and predicate symbols need to be interpreted ("interpretation") and variables need to get values assigned ("valuation").

Definition 3.1.5. Given a non-empty domain of values D, an *interpretation* I of a formula F assigns to each constant, function symbol and predicate symbol in F its meaning in the following way:

- I assigns an element c_I from the domain D to every constant c
- I assigns a function $f_I : D^n \rightarrow D$ to every n-ary function symbol f
- I assigns a predicate $p_I : D^n \rightarrow \{\mathbb{T}, \mathbb{F}\}$ to every n-ary predicate symbol p

A *structure* S is a pair of a domain D and an interpretation I. We write this as $S = (D, I)$ (chapter 3 of [13] and chapter 3 of [27]).

There are also special structures, called *Herbrand structures*, whose domain is the set of ground terms (also *Herbrand universe or Herbrand domain*) and where all ground terms are interpreted in themselves for a particular first-order language. A *Herbrand interpretation* is a canonical interpretation, where the domain is the Herbrand universe for some appropriate language. There is also a *Herbrand model* of a set of formulas, that is a model of those formulas that is also a Herbrand interpretation (chapter 3 of [13]).

Definition 3.1.6. A *valuation* v (sometimes also called *assignment*) of a formula F maps every variable x to an object $v(x)$ in the domain D (chapter 2 of [13] and chapter 6 of [27]).

In this thesis we denote the *(truth) value* of a first-order formula F under a structure S and a variable assignment v by $[\![F]\!]_S^v$.

Let F be a first-order formula, S be a structure and v be a valuation. Then

- F is called *satisfiable*, if there exists some structure S and a valuation v such that $[\![F]\!]_S^v = \text{true}$. A formula is called *unsatisfiable*, if it is not satisfiable (chapter 2 of [13]).
- S is called a *model* of a formula F (denoted as $S \vDash F$), if for every valuation v holds $[\![F]\!]_S^v = \text{true}$ (chapter 3 of [13]).
- F is called *valid* (denoted as $\vDash F$), if every structure S is a model of F. In other words: for every structure S we have $M \vDash F$ (we also say that F is a *tautology*) (chapter 2 of [13]).

Definition 3.1.7. A *substitution* σ is a finite set of pairs (*replacements*) $v \to t$ between a variable v and a different term t: $\sigma = \{v_1 \to t_1, \dots, v_n \to t_n\}$ (for every $i \neq j : v_i \neq v_j$ and $n \geq 0$). The *application* of a substitution σ to an expression E is written by $E\sigma$. $E\sigma$ is obtained by simultaneously replacing all occurrences of v_i by the corresponding term t_i. $E\sigma$ is then called an *instance* of E (chapter 3 of [16] and chapter 2 of [27]).

Remark. We will call a substitution σ of p that maps into the Herbrand universe a *ground instance* of p (chapter 3 of [13]).

Theorem 3.1.1 (Herbrand's theorem)**.** *A quantifier-free formula p is first-order satisfiable if and only if the set of all its ground instances is (propositionally) satisfiable (page 156 in chapter 3 of [13]).*

3.2 Skolemization

All the strategies described in the following sections work on a set of clauses. *Skolemization* is a method to transform a first-order logic formula into such a set of clauses. In the so-called *prenex normal form (PNF)* all quantifiers are sorted out to the beginning of the formula, but this may still contain an arbitrarily complicated nesting of quantifiers. The technique of Thoralf Skolem (1928) called *Skolemization* [40] makes it possible to eliminate all existential quantifiers and leaving only universal ones.

It is possible to prove that the following two statements are in general mathematically equivalent (chapter 3.6 of [13]):

1. for all $x \in D$, there exists a $y \in D$ such that $P[x, y]$
2. there exists an $f : D \to D$ such that for all $x \in D$, $P[x, f(x)]$ holds

If we allowed existential quantification of functions (which is not legal in first-order logic) the above equivalence means that the following formula would be logically valid:

$$(\forall x.\exists y.P[x, y]) \Leftrightarrow (\exists f.\forall x.P[x, f(x)])$$

More generally, the following formula would be valid:

$$(\forall x_1, \ldots, x_n.\exists y.P[x_1, \ldots, x_n, y]) \Leftrightarrow (\exists f.\forall x_1, \ldots, x_n.P[x_1, \ldots, x_n, f(x_1, \ldots, x_n)])$$

With the help of these equivalences we are able to transform a formula into PNF by moving all existential quantifiers before all universal ones. Since these, however, are not first-order formulas, it is not possible to follow this process exactly, but we can get a similar effect if we do not require that the transformed formula is logically equivalent to the original one, but only that it preserves its satisfiability, i.e., that it is satisfiable is and only if the original one is. To decide whether or not a certain formula is satisfiable, we only need to check the satisfiability of the transformed version without existential quantification of functions (chapter 3.6 of [13]).

For the Skolem transformation we have to introduce new function symbols, denoted as *Skolem functions* (respectively *Skolem constants* in the case of arity zero), which are not allowed to be present in the original formula (chapter 3.6 of [13]).

The fact that the transformation preserves satisfiability rests on the following result.

Theorem 3.2.1. *Let p be a formula not involving the n-ary function symbol f and let M be any interpretation. Then there exists another interpretation M' that differs from M only in the interpretation of f, such that in all valuations v the truth value of $\exists y.p$ under M and v is equal to the truth value of $p\sigma$ (for $\sigma = \{y \to f(x_1, \ldots, x_n)\}$) under M' and v. Additionally, the truth value of $\exists y.p$ under M and v and $\exists y.p$ under M' and v is the same (chapter 3.6 of [13]).*

Proof. page 146 and 147 in chapter 3.6 of [13] □

According to this theorem we can establish the following: Consider an arbitrary interpretation M and a formula p with some subformula $\exists y.q$. Assume f does not occur in the formula p. We can now Skolemize the subformula with f and then get a new formula p' and a new model M'. This new model differs from M only in the interpretation of f such that for all valuations v the truth value of p under M and v is the same as the truth value of p' under M' and v. We can repeat this process and replace all existentially quantified subformulas (choosing only function symbols that are not present in the formulas so far) (chapter 3.6 of [13]).

Remark. This transformation does not depend on any kind of initial normal form — the process of Skolemization is nondeterministic, i.e., it is possible to apply Skolemization to any existentially quantified subformula. If the original subformula is satisfiable, then also its Skolemization is. This also means that many significantly different Skolemized formulas can be produced — some might be more complicated than others. It is advisable to work from the outside in, i.e., we should choose an existential quantifier that is not itself within the scope of another existential quantifier (chapter 8.3 of [11]).

3.3 Unification

Decision methods for propositional logic are applied together with a systematic enumeration of the ground instances. Prawitz, Prawitz and Voghera introduced in 1960 [21] the idea to perform propositional operations on formulas that are just as much instantiated as necessary to make progress with the propositional reasoning. J. A. Robinson extended this work by defining an effective syntactic procedure called *unification* [29]. This is for deciding on appropriate instantiations to match up terms correctly [13].

Definition 3.3.1. Given a set of pairs of terms $\mathcal{S} = \{(s_1, t_1), \ldots, (s_n, t_n)\}$. Let σ be a substitution. If $\sigma s_i = \sigma t_i$ (for each $i = 1, \ldots, n$), then σ is called an *unifier* of the set $\mathcal{S}$. In the case of a single pair $\{(s,t)\}$ we often call it an *unifier of s and t*. The set $\mathcal{S} = \{(s_1, t_1), \ldots, (s_n, t_n)\}$ is called *unifiable* if and only is there exits an unifier of it (chapter 3.9 of [13]).

In general, the unification problem, i.e. the problem of finding a unifier may be unsolvable; for example there is no unifier for x and $f(x)$. Conversely, if there exists an unifier, then there are always infinitely many, because if σ is an unifier of s and t, then also $\tau \circ \sigma$ is one for any other substitution τ. Luckily it is always feasible to find a *most general unifier* that keeps the instantiating terms as "simple" as possible.

Definition 3.3.2. Let σ_1 and σ_2 be two substitutions. We say that σ_2 is *more general* than σ_1 if for some substitution τ: $\sigma_1 = \sigma_2 \circ \tau$ (chapter 7.2 of [11]).

Definition 3.3.3. A unifier σ of a set $\mathcal{S}$ of expressions is a *most general unifier (MGU)* of $\mathcal{S}$ if and only if for each unifier θ of the set $\mathcal{S}$ there exists a substitution τ such that $\theta = \sigma \circ \tau$. In other words, a substitution σ is a most general unifier if it is a unifier and at least as general as any other unifier (chapter 7.2 of [11] and chapter 3.9 of [13]).

MGUs may not be unique. Consider for example the set $\{p(x), p(y)\}$, then there exist two different MGUs namely $\{x \to y\}$ and $\{y \to x\}$. It is possible to show, that two MGUs of a given set differ only up to a permutation of the variable names (for substitutions that affect only a finite number of variables). See also chapter 3.9 of [13].

Now we are turning to a general algorithm for deciding whether or not two terms are unifiable and, if they are, for computing a MGU. Basically, we will explain in the following a unification algorithm similar to that of Robinson [29]. It might be helpful to think instead of the conventional, linear term writing about terms as tree representation. An *augmented tree representation* of a term is a tree in which the label on each node has been extended by an integer representing which child this is (chapter 7.2 of [11]).

Definition 3.3.4. Let t_1 and t_2 be two terms. A *disagreement pair* for t_1 and t_2 is a pair (d_1, d_2), where d_1 is a subterm of t_1 and d_2 is a subterms of t_2, such that d_1 and d_2 have distinct labels at their roots, but the path from the root of t_1 down to the root of d_1 (analogously for d_2 ant t_2) are the same (chapter 7.2 of [11]).

Remark. If two terms differ, there exits at least one disagreement pair. If σ unifies those two terms, it has to unify every disagreement pair of these terms.

To find the *disagreement set* $\mathcal{D}$ of a non-empty set $\mathcal{S}$ of expressions we do the following [18]:

1. locate the first position (starting from the left) at which not all expressions in S have the exact same symbol
2. extract from each expression in S the subexpression that begins with the symbol at this position

We state the algorithm in a non-deterministic form (using pseudocode). The denotation "STOP" means that the algorithm terminates and gives some kind of message. The algorithm is taken from [18].

Algorithm 3.1 Unification Algorithm

Require: set $\mathcal{S}$ of expressions
Ensure: σ is a most general unifier for $\mathcal{S}$ (empty if not unifiable)

```
k := 0, S_k := S, σ_k := {}
if S_k is singleton then
    STOP; σ_k is a MGU of S
else
    find the disagreement set D_k of S_k
end if
if ∃v_k, t_k ∈ D_k : v_k is a variable which does not occur in t_k then
    continue
else
    STOP; S is not unifiable
end if
Let σ_{k+1} = σ_k ∘ {v_k → t_k} and S_{k+1} = S_k{v_k → t_k}
k = k + 1; Start again at "if S_k is singleton"
```

Example 3.3.1. Let us consider the following set of expressions:

$$\mathcal{S}_1 = \{p(g(x), x, f(g(y))), p(z, a, f(z))\}$$

Then the disagreement set is $\mathcal{D}_1 = \{g(x), z\}$ and a MGU is $\sigma_1 = \{z \to g(x)\}$. A second example is $\mathcal{S}_2 = \{p(g(a), a, f(g(y))), p(g(a), a, f(g(a)))\}$. Then the disagreement set is $\mathcal{D}_2 = \{y, a\}$ and a MGU is $\sigma_2 = \{z \to g(a), x \to a, y \to a\}$ [18].

This algorithm has some useful properties: it will always terminate; if there does not exists an unifier the algorithm will STOP; if there exists an unifier the algorithm will not terminate with STOP; and the final value of σ will be a MGU of $\mathcal{S}$.

To prove termination, let $S(\sigma)$ be the set of variables that occur in $\mathcal{S}$. Every run through the algorithm decreases the size of $S(\sigma)$ (if it does not terminate with STOP), since v_k is replaced by a term t_k that can not contain v_k. The set $\mathcal{S}$ has only finitely many variables, therefore the algorithm is guaranteed to terminate (chapter 7.2 of [11]). Proving the correctness requires more argumentation. The following lemma provides the key component:

Lemma 3.3.1. *Assume r and s are two terms that are unifiable and τ is an unifier of r and s. Assume also that (x,t) is a disagreement pair for r and s where x is a variable. Then (1) x does not occur in t and (2) $\tau = \tau \circ \{x \to t\}$ chapter 7.2 of [11].*

Proof. page 156, chapter 7.2 of [11]. □

Based on this lemma, one can prove the so-called *Unification Theorem* which states the correctness of the algorithm. The theorem as well as its proof can be found in chapter 7.2 of [11].

The unification algorithm as we stated it above finds a unifier for a set of more than two terms. We denote this kind of problem as *multiple unification*. More fundamental algorithms are only unifying exactly two terms — this is sometimes called *binary unification*. It is possible to reduce the problem of multiple resolution to a sequence on binary unification problems (chapter 7.2 of [11]).

Of course there exists many different unification algorithms (especially more efficient ones). A description of another algorithm can for example be found on page 166 − 171 in chapter 3.9 of [13]

3.4 Inference Systems and Proofs

Definition 3.4.1. An *inference rule* is a n-ary relation on formulas (where $n \geq 0$). The elements of this relation are called *inferences* which we write as follows:

$$\frac{F_1 \quad \dots \quad F_n}{F}$$

The formulas $F_1, \dots, F_n$ are called the *premises* of the inference, the formula F is called the *conclusion* of the inference. A set of inference rules is called an *inference system*, denoted as $\mathcal{I}$. An *axiom* of an inference system is any conclusion of an inference without premises ($n = 0$). This is written by simply F — without the line above (chapter 3 of [16]).

Remark. We might consider inference systems where the order of the premises in an inference is relevant. Therefore a distinguished premise (the *main premise*) gets reduced to the conclusion using the other premises (the *side premises*). Unless we explicitly specify differently, the last premise of an inference is the main premise and the other premise are the side premises (chapter 2 of [27]).

An example for an inference system for first-order logic with equality is the so-called superposition inference system. This concept gets explained in section 3.7.2.

Definition 3.4.2. A *derivation* in an inference system $\mathcal{I}$ is a tree built from inferences in $\mathcal{I}$. We call it a *derivation of F*, if the root of this derivation is F (chapter 3 of [16]).

Definition 3.4.3. We call a derivation of F a *proof of F* if it is finite and all leaves in the derivation are axioms. If a formula F has a proof, it is called *provable* in $\mathcal{I}$ (chapter 3 of [16]).

If a derivation of F is finite and each leaf is either an axiom or one of the formulas $F_1, \ldots, F_n$, then this derivation is called *from assumptions* $F_1, \ldots, F_n$. We say that a formula F is *derivable from* assumptions $F_1, \ldots, F_n$ if there exists a derivation of F from $F_1, \ldots, F_n$. A derivation of $\perp$ is called a *refutation*. A derivation from the empty set of assumptions is a proof (chapter 3 of [16]).

Definition 3.4.4. Let $\mathcal{I}$ be an inference system and $\mathcal{S}$ a set of formulas in $\mathcal{I}$. If the empty clause $\square$ is derivable from $\mathcal{S}$ in $\mathcal{I}$, then $\mathcal{S}$ is unsatisfiable. $\mathcal{I}$ is then called a *sound* inference system (chapter 3 of [16]).

Definition 3.4.5. Let $\mathcal{I}$ be an inference system and $\mathcal{S}$ a set of formulas in $\mathcal{I}$. If $\mathcal{S}$ is unsatisfiable, then the empty clause $\square$ is derivable from $\mathcal{S}$ in $\mathcal{I}$. $\mathcal{I}$ is then called *refutation complete* (sometimes also *refutationally complete*) (chapter 3 of [16]).

3.5 Ordering and Selection

For better understanding of the following sections we need to introduce the concepts of *simplification orderings* and *selection functions*.

Definition 3.5.1. A *(strict partial) ordering* $\succ$ is a transitive and irreflexive binary relation (chapter 7.2 of [27]).

Definition 3.5.2. An ordering $\succ$ on terms is denoted as a *simplification ordering* on terms if these statements are fulfilled (Chapter 3.2 of [16]):

- $\succ$ is *well-founded*, i.e., there is no infinite sequence of terms $s, t, \ldots$ such that $s \succ t \succ \ldots$
- $\succ$ is *monotonic*, i.e., if $s \succ t$, then $r[s] \succ r[t]$ for all terms r, s, t
- $\succ$ is *stable* under substitutions, i.e., if $s \succ t$, then also $s\sigma \succ t\sigma$ for all terms s, t and all substitutions σ
- $\succ$ has the *subterm property*, i.e., if s is a subterm of t and $s \neq t$, then $t \succ s$

This can be easily extended to a simplification ordering on atoms, literals or clauses by comparing their atoms. Additionally, this definitions provides some notion of an expression " being simpler" than another one (chapter 3 of [16]). Furthermore, a *rewrite ordering* is a monotonic ordering that is stable under substitutions and a *reduction ordering* is a well-founded rewrite ordering (chapter 7.2 of [27]).

Example 3.5.1. The Knuth-Bendix ordering is a simplification ordering. Let s and t be two ground terms. If s has fewer symbols than t, then $t > s$ (chapter 3 of [16]).

Definition 3.5.3. Let $>$ be an ordering on terms, $=$ be a congruence relation and s_1, s_2, t_1, t_2 be terms. $>$ is called *compatible* with $=$ if $s_1 = s_2 > t_2 = t_1$ implies $s_1 = t_1$ for all s_1, s_2, t_1, t_2 (chapter 7.2 of [27]).

Example 3.5.2. Let $>$ be an ordering and $=$ a congruence relation. We consider the *lexicographic (left to right) extension* of $>$ with respect to $=$ denoted by $>^{lex}$. On n-tuples of terms this is defined by $(s_1, \dots, s_n) >^{lex} (t_1, \dots, t_n)$ if $s_1 = t_1, \dots, s_{k-1} = t_{k-1}$ and $s_k > t_k$ for $k \in \{1, \dots n\}$ (chapter 7.2 of [27]).

Definition 3.5.4. A *selection function* is a mapping, which selects in every clause C a (possibly empty) subset of literals (or equations). In this thesis we will mark the selected literals by underlining them i.e. If we write $\underline{L} \vee C$ we mean that L is selected in $L \vee C$ (chapter 3 of [16]).

Example 3.5.3. The function that selects all maximal literals with respect to the simplification ordering that is used in the inference system is a selection function. This function is sometimes used in superposition theorem provers (chapter 3 of [16]).

Definition 3.5.5. A selection function is called *well-behaved*, if it selects either a negative literal or all maximal literals, according to the simplification ordering (chapter 3 of [16]).

It is possible to improve an inference system using simplification orderings and selection functions. Intuitively, we only compute inferences that involve a selected literal (if there is at least one selected literal in the clause) and if a clause has no selected literals, we compute only those inferences that involve a maximal literal. More details on that can be found in the following sections about "resolution", "paramodulation" and "superposition".

3.6 Resolution

The resolution calculus was in essence invented by Robinson [29] and is a refutationally complete method for theorem proving. It was already briefly discussed in chapter 2 "State of the Art". Subsequently, we explain the theoretical concepts that form the basis of resolution theorem provers.

One version of resolution combines the two inference rules "(binary) resolution" and "factoring" to a single rule (chapter 3 of [27]):

$$\frac{C \lor A \lor \ldots \lor A \qquad \neg A \lor D}{C \lor D} \text{ Binary resolution with factoring}$$

Remark. Combining resolution and factoring into one single inference rule has the advantage of being simpler to implement and it restricts the formation of factors to those that are necessary to "lift" a particular propositional resolution step (chapter 3 of [27]). Please note, that all these A are different atoms, but they are unifiable (why we chose to denote it like this)

How does the first-order resolution principle employ unification? Remember Herbrand's theorem: for an unsatisfiable set of clauses there is a propositionally unsatisfiable finite conjunction of propositional instances of them. One way to detect this propositional unsatisfiablity is to repeatedly apply the propositional resolution rule. Let $C[x_1, \ldots, x_n]$ and $D[y_1, \ldots, y_m]$ be two clauses to which we can apply propositional resolution: $C[x_1, \ldots, x_n] = \ldots \lor P(s_1, \ldots, s_m) \lor \ldots$ and $D[y_1, \ldots, y_n] = \ldots \lor \neg P(s'_1, \ldots, s'_m) \lor \ldots$ such that, if we apply the appropriate ground instantiation σ, then the set $S = \{(s_i, s'_i) \mid i = 1, \ldots, m\}$ gets unified and we are able to apply resolution. Unfortunately, when using an MGU instead of σ we are not guaranteed that performing resolution on the instantiated clauses also makes it possible to obtain the original result by further instantiations. The MGU might just be too general. This means we have to additionally consider unifying some subset of the literals in the same clause (chapter 3 of [13]).

Lemma 3.6.1 (Lifting lemma)**.** *Let A and B be first-order clauses with no variables in common and let A' and B' be two (not necessarily ground) instances of A and B, such that A' and B' have a propositional resolvent C'. Then there exist non-empty subsets $A_1 \subseteq A$ and $B_1 \subseteq B$ such that $S = A_1 \cup B_1^-$ is unifiable. Additionally, for any σ that is a MGU of S, C' is an instance of $((A - A_1) \cup (B - B_1))\sigma$ (chapter 3 of [13]).*

Proof. page 181 – 182, chapter 3.11 of [13] □

According to this results it is possible to define a (first-order) resolvent of two clauses A and B to be $((A_0 - A_1) \cup (B_0 - B_1))\sigma$ where A_0 and B_0 are A and B but renamed without variables in common and $A_1 \subseteq A_0$ and $B_1 \subseteq B_0$ (non-empty subset) with σ the selected MGU of $A_1 \cup B_1^-$ (chapter 3 of [13]).

Theorem 3.6.2 (compactness theorem)**.** *Let Γ be a set of propositional formulas. If each finite subset $\Delta \subseteq \Gamma$ is satisfiable, then also Γ is satisfiable (chapter 2.12 of [13]).*

Proof. See page 108 in chapter 2.12 of [13]. □

Theorem 3.6.3. *Let S be a set of first-order clauses. If S is unsatisfiable, then the empty clause $\square$ is derivable by resolution (chapter 3 of [13]).*

Proof. Let S be a set of first-order clauses. Then by Herbrand's theorem and compactness, there is a finite set of ground instances of clauses in S that is unsatisfiable. By the refutation completeness of the propositional resolution the empty clause is derivable by resolution. Let C' be an instance of C. Using induction on the structure, it is possible to apply the lifting lemma to show that for each subproof of a clause C' there exists a corresponding proof using first-order resolution of a clause C. To finally conclude the empty clause, the empty clause has to be derivable by first-order resolution, because the empty clause cannot be an instance of a non-empty clause (page 182, chapter 3.11 of [13]) $\square$

Since resolution is a sound and refutation complete inference rule, it provides a suitable basis for refutational theorem proving. The above rule might produce a (very) large search space, therefore we discuss selection functions and simplification orderings as control mechanisms. For example, it is possible to restrict resolution on the maximal atoms in the side premise only; thus it is sufficient to only resolve on negative literals in a non-deterministic way. Finally, by simultaneously resolving on more than one atom we achieve a larger inference step (chapter 2 of [27]).

In the following let $\succ$ be an admissible clause ordering and let S be a selection function. Let $\mathcal{D}$ denote the clause $\neg A_1 \vee \ldots \vee \neg A_n \vee D$. The symbol $\mathcal{O}_S^{\succ}$ defines the inference system for *ordered resolution with selection.* $\mathcal{O}_S^{\succ}$ looks like this (chapter 2 of [27]):

$$\frac{C_1 \vee \ldots \vee A_1 \qquad \ldots \qquad C_n \vee \ldots \vee A_n \qquad \neg A_1 \vee \ldots \vee \neg A_n \vee D}{C_1 \vee \ldots \vee C_n \vee D}$$

where (1) either the subclause $\neg A_1 \vee \ldots \vee \neg A_n$ is selected by S in $\mathcal{D}$ or $S(\mathcal{D})$ is empty, $n = 1$ and A_1 is maximal with respect to $\mathcal{D}$ (2) every atom A_i is strictly maximal with respect to C_i (3) non of the clauses $C_i \vee \ldots \vee A_i$ contains a selected atom.

Generally, in ordered inferences there are two possibilities: first, all selected atoms get resolved at once or second, if no atoms are selected, one resolves the maximal atom of the main premise. Ordered resolution is design in a way, that each resolvent is smaller than the corresponding main premise (chapter 2 of [27]).

To prove the refutation completeness of ordered resolution with selection we summarize some important facts beforehand.

Definition 3.6.1. Let $\succ$ be a total, admissible ordering on clauses, N be a set of clauses and $I_C = \bigcup_{C \succ D} \epsilon_D$ be a Herbrand interpretation. Furthermore, if C is a clause fulfilling the following conditions:

1. C is contained in N
2. C is of the form $C' \vee A$ for the maximal literal A in C
3. C is false in I_C
4. there is nothing selected in C

then $\epsilon_C = \{A\}$, else $\epsilon_C = \emptyset$. We call C a *productive clause* if $\epsilon_C = \{A\}$ (chapter 2 of [27]).

Theorem 3.6.4. *Let N be a set of clauses that does not contain the empty clause and C be the minimal counterexample in N for $I_N := \bigcup_{C \in N} \epsilon_C$. Then there exists an inference in $\mathcal{O}$ from C such that (chapter 2 of [27]):*

1. *its derived conclusion is a counterexample for I_N and it is smaller than C*

2. C is the main premise and the side premises are productive clauses.

Remark. We call a clause that is false in an interpretation I *counterexample* for I (chapter 2 of [27]).

This is also called reduction property for counterexamples by ordered resolution and it immediately implies that the inference system $\mathcal{O}_S^{\succ}$ is refutation complete (chapter 2 of [27]).

Theorem 3.6.5. *Let N be a set of clauses that is saturated with respect to $\mathcal{O}_S^{\succ}$ and it does not contain the empty clause. Then I_N is a model of N. Therefore the inference system $\mathcal{O}_S^{\succ}$ is refutation complete (chapter 2 of [27]).*

3.7 Reasoning about Equality

In the "State of the Art" chapter 2 we already heard a brief introduction about handling the equality predicate. We will focus on two proving techniques with additional rules for equality — *paramodulation* and *superposition*.

3.7.1 Paramodulation

Since 1965 Robinson's resolution procedure has been the main focus in the field of automated theorem proving. As already discussed, this method is based on a single inference rule, but several refinements have improved the procedure. Researchers sought for extended methods to incorporate different theories such as equality, set theory or number theory [12]. To handle the theory of equality G. Robinson and Wos have added a new inference rule called *paramodulation* [28]. If this gets used together with the standard resolution rule, it results in a theoretically complete method of handling the equality. The initial form of the paramodulation rule can already be found in Chapter 2 "State of the Art".

The strengths and weaknesses of this method have encouraged lots of theoretical and practical research. A large number of experiments with paramodulation has been done at the Argonne group for example by Wos, Overbeek and Henschen (chapter 7 of [27]). Let's consider the paramodulation inference rule for variable free formulas (chapter 1 of [2]):

$$\frac{\Gamma_1 \to \Delta_1, s = t \qquad \Gamma_2 \to \Delta_2, u[s] = v}{\Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2, u[t] = v} \text{ (Paramodulation)}$$

To restrict the number of inferences computed during the proof search *term orderings* are an important tool. In fact, paramodulation is based on the so-called *Leibniz' law* for replacing equals by equals. Assume $\succ$ is an ordering which is total on variable-free terms and formulas. The basic idea of *ordered paramodulation* is to only replace big terms by smaller ones according to $\succ$ — same idea as in *ordered rewriting* (chapter 7 of [27]).

Definition 3.7.1. A paramodulation inference is called *ordered* (with respect to $\succ$) if the following conditions are fulfilled:

1. $s \succ t$
2. $s = t$ is strictly maximal with respect to $\Gamma_1 \cup \Delta_1$
3. $u[s] = v$ is strictly maximal with respect to $\Gamma_2 \cup \Delta_2$

(chapter 1 of [2]).

Remark. Hsiang and Rusinowitch proved in 1991 the refutation completeness of ordered paramodulation with the use of *transfinite semantic trees* [14].

3.7.2 Superposition

Consider the paramodulation inference rule as well as the definition 3.7.1 about ordered paramodulation. We can extend this definition to receive further interesting properties:

Definition 3.7.2. An ordered paramodulation inference is called a *superposition inference* if the additional condition (additional to definition 3.7.1) is satisfied:

4. $u[s] \succ v$

If s does not occur in Γ_1 the superposition inference is called *strict*. A *weak superposition* inference denoted a paramodulation inference for which the conditions (1), (3) and (4) hold (but (2) is not necessary) (chapter 1 of [2]).

In 1991 weak superposition was proven to be refutation complete by Rusinowitch [31]. Unfortunately strict superposition is not complete, but exists several slight enrichments of the calculus that are also proven to be refutation complete (by Bachmair and Ganzinger in 1990 [1])

Definition 3.7.3. Let $\succ$ be a reduction ordering. A clause $C = \Gamma \rightarrow \Delta, s = t$ is called *reductive* for $s = t$ if $t \nsucceq s$ and $s = t$ is a strictly maximal equation in C (chapter 3 of [2]).

In this chapter we focus on superposition calculi with selection functions on negative literals and analyse their refutation completeness. We define the following inference rules with respect to the reduction ordering $\succ$. This first rule represents the reflexivity property of the equality predicate (chapter 3 of [2]):

$$\frac{\Gamma, u = v \rightarrow \Delta}{\Gamma\sigma \rightarrow \Delta\sigma} \text{ (Equality resolution)}$$

where σ is a MGU of u and v and $u\sigma = v\sigma$ is a maximal equation in $\Gamma\sigma, u\sigma = v\sigma \rightarrow \Delta\sigma$.

This inference rule encodes a variant of factoring which is restricted to the succedent of clauses (chapter 3 of [2]):

$$\frac{\Gamma \rightarrow \Delta, A, B}{\Gamma\sigma \rightarrow \Delta\sigma, A\sigma} \text{ (Ordered factoring)}$$

where σ is a MGU of A and B and $A\sigma$ is a maximal equation in $\Gamma\sigma \rightarrow \Delta\sigma, A\sigma, B\sigma$.

The following two rules represent now the restricted version of paramodulation (chapter 3 of [2]):

$$\frac{\Gamma_1 \rightarrow \Delta_1, s = t \qquad \Gamma_2, u[s'] = v \rightarrow \Delta_2}{\Gamma_1\sigma, \Gamma_2\sigma, u[t]\sigma = v\sigma \rightarrow \Delta_1\sigma, \Delta_2\sigma} \text{ (Superposition left)}$$

where σ is a MGU of s and s'; $\Gamma_1\sigma \rightarrow \Delta_1\sigma, s\sigma = t\sigma$ is a reductive clause for $s\sigma = t\sigma$; $v\sigma \not\succeq u\sigma$ and $u\sigma = v\sigma$ is a maximal equation in $u\sigma = v\sigma, \Gamma_2\sigma \rightarrow \Delta_2\sigma$; and s' is not a variable.

$$\frac{\Gamma_1 \rightarrow \Delta_1, s = t \qquad \Gamma_2 \rightarrow \Delta_2, u[s'] = v}{\Gamma_1\sigma, \Gamma_2\sigma \rightarrow \Delta_1\sigma, \Delta_2\sigma, u[t]\sigma = v\sigma} \text{ (Superposition right)}$$

where σ is a MGU of s and s'; $\Gamma_1\sigma \rightarrow \Delta_1\sigma, s\sigma = t\sigma$ is a reductive clause for $s\sigma = t\sigma$; $\Gamma_2\sigma \rightarrow u\sigma = v\sigma, \Delta_2$ is a reductive clause for $u\sigma = v\sigma$; and s' is not a variable.

To achieve refutation completeness the inference system gets combined with additional rules. We could for example add an inference rule which is basically a generalization of ordered factoring (chapter 3 of [2]):

$$\frac{\Gamma \rightarrow \Delta, s = t, s' = t'}{\Gamma\sigma, t\sigma = t'\sigma \rightarrow \Delta\sigma, s\sigma = s'\sigma} \text{ (Equality factoring)}$$

where σ is a MGU of s and s'; $t\sigma \not\succ s\sigma$ and $t'\sigma \not\succ s'\sigma$; and $s\sigma = t\sigma$ is a maximal equation in $\Gamma\sigma \rightarrow, s\sigma = t\sigma, s'\sigma = t'\sigma$.

Alternatively it is possible to use the following paramodulation rule (chapter 3 of [2]):

$$\frac{\Gamma_1 \rightarrow \Delta_1, s = t \qquad \Gamma_2 \rightarrow \Delta_2, u = v[s'], u' = v'}{\Gamma_1\sigma, \Gamma_2 \rightarrow \Delta_1\sigma, \Delta_2\sigma, u\sigma = v[t]\sigma, u\sigma = v'\sigma} \text{ (Merging Paramodulation)}$$

where $\sigma = \tau\rho$ is the composition of a MGU τ of s and s' and a MGU ρ of $u\tau$ and $u'\tau$; $\Gamma_1\sigma \rightarrow \Delta_1\sigma, s\sigma = t\sigma$ is a reductive clause for $s\sigma = t\sigma$; $\Gamma_2\sigma \rightarrow \Delta_2\sigma, u\sigma = v\sigma, u'\sigma = v'\sigma$ is a reductive clause for $u\sigma = v\sigma$; $u\tau \succ v\tau$ and $v'\sigma \not\succeq v\sigma$; and s' is not a variable.

Remark. If we apply merging paramodulation repeatedly to ground clauses (in conjunction with ordered factoring), the atoms in the succedent containing a maximal term get merged (chapter 3 of [2]).

Additionally, the following two restrictions apply: First, it is not allowed that the premises of an inference rule share any variables (renaming if necessary) and second, assume C and D are the premises of a paramodulation inference with a MGU σ, then $C\sigma \not\succeq^C D\sigma$. These inference rules have one important property. The conclusion of a ground inference is always simpler than the maximal premise with respect to the ordering $\succ^C$. Considering a paramodulation inference, then the maximal premise is always the second one (chapter 3 of [2]).

The inference rules above can be further controlled by using a selection function. To distinguish between several selected equations an ordering might be used. Assume S is a given selection function (chapter 3 of [2]):

$$\frac{\Gamma, u = v \rightarrow \Delta}{\Gamma\sigma \rightarrow \Delta\sigma} \text{ (Selective resolution)}$$

where σ is a MGU of u and v; and $u = v$ is a selected equation in $\Gamma, u = v \rightarrow \Delta$.

$$\frac{\Gamma_1 \rightarrow \Delta_1, s = t \qquad \Gamma_2 \rightarrow \Delta_2, u[s'] = v}{\Gamma_1\sigma, \Gamma_2\sigma, u[t]\sigma = v\sigma \rightarrow \Delta_1\sigma, \Delta_2\sigma} \text{ (Selective superposition)}$$

where σ is a MGU of s and s'; $C = \Gamma_1 \rightarrow \Delta_1, s = t$ does not contain any selected equations and $C\sigma$ is a reductive clause for $s\sigma = t\sigma$; $v\sigma \not\succcurlyeq u\sigma$ and $u = v$ is a selected equation in $u = v, \Gamma_2 \rightarrow \Delta_2$; and s' is not a variable.

An inference system would then for example consist of the two selective inference rules as well as equality resolution, equality factoring, superposition and merging paramodulation. For the non-selective rules the additional condition of no selected literals in the premise applies (chapter 3 of [2]).

3.7.3 Refutation Completeness

In this section we discuss the refutation completeness of paramodulation and superposition. The standard technique for proving refutation completeness of ordered paramodulation is the so-called *model generation method* based on *strict superposition* i.e. the paramodulation involves only maximal terms of maximal equations of clauses (according to the ordering $\succ$). As we already discussed before, (strict) superposition is a restricted form of paramodulation, therefore we only give one proof idea in this thesis.

In this section let $\succ$ be a total reduction ordering and $s \succcurlyeq t$ denotes $s \succ t \lor s = t$. We introduce the concept on ground Horn clauses with equality and generalize it later on. The inference system $\mathcal{I}$ consists of the following rules (chapter 7.3 of [27]):

$$\frac{\Gamma_1 \rightarrow l = r \qquad \Gamma_2 \rightarrow s = t}{\Gamma_1, \Gamma_2 \rightarrow s[r] = t} \text{ (Superposition right)}$$

if s is at some position p equivalent to l; $l \succ r$; $s \succ t$; $l \succ u$ for all u in Γ_1; and $s \succ v$ for all v in Γ_2.

$$\frac{\Gamma_1 \rightarrow l = r \qquad \Gamma_2, s = t \rightarrow \Delta}{\Gamma_1, \Gamma_2 s[r] = t \rightarrow \Delta} \text{ (Superposition left)}$$

if s is at some position p equivalent to l; $l \succ r$; $s \succ t$; $l \succ u$ for all u in Γ_1; and $s \succ v$ for all v in Γ_2 and Δ.

$$\frac{\Gamma, s = s \rightarrow \Delta}{\Gamma \rightarrow \Delta} \text{ (Equality resolution)}$$

if $s \succ v$ for all v in Γ and Δ. This inference rule combines the resolution rule with the reflexivity axiom.

Next we define a total ordering $\succ_C$ on ground clauses. Let C be a clause $s_1 = s'_1, \ldots, s_n = s'_n \rightarrow t_1 = t'_1, \ldots, t_m = t'_m$ and let $ms(C)$ be the multiset $\{\{s_1, s_1, s'_1, s'_1\}, \ldots, \{s_n, s_n, s'_n, s'_n\}, \{t_1, t'_1\}, \ldots,$
$\{t_m, t'_m\}\}$. Now let $\succ_C$ be the ordering that compares these expressions by the two-fold multiset extension of $\succ$ i.e. $C \succ_C D$ if $ms(C) \succ_{mul} ms(D)$ (chapter 7.3 of [27]).

Let S be a set of ground Horn clauses closed under the inference system $\mathcal{I}$. The key idea for proving completeness of $\mathcal{I}$ is the following: If $\square \notin S$, then S is satisfiable (this gives a constructive proof). In a first step, an equality Herbrand interpretation will be constructed, then we can show that this interpretation is a model of S. Informally speaking, we construct the interpretation by a congruence R^* created by a set of ground rewrite rules R. Every of these rules has been generated by some clause of S. This generation process is formally defined by induction on the ordering $\succ_C$. Every clause C in S either generates a rule or it does not. This depends on the set R_C of rules generated by clauses D in S where $C \succ_C D$ (chapter 7.3 of [27]). Formally this is defined as follows.

Definition 3.7.4 (Model generation)**.** Let S be a set of ground Horn clauses closed under the inference system $\mathcal{I}$ and let C be a clause in S. Then $Gen(C) = \{s \Rightarrow t\}$ and C *generate* the rule $s \Rightarrow t$ if and only if C has the form $\Gamma \rightarrow s = t$ and these three conditions are fulfilled:

1. $R^*_C \nvDash C$
2. $s \succ t$ and $s \succ u$ for all u in Γ
3. s is irreducible by R_C

where $R_C = \bigcup_{C \succ_C D} Gen(D)$, otherwise $Gen(C) = \emptyset$ and $R = \bigcup_{C D \in S} Gen(D)$ (i.e. the set of all rules generated by S) (chapter 7.3 of [27]).

Condition two and three are the key components for showing convergence of R. It is quite easy to show completeness from this result.

Lemma 3.7.1. *Let S be a set of ground clauses. Then the set of rules R generated for S is convergent. Additionally, if $R^*_C \vDash C$ then $R^* \vDash C$ for all ground clauses C (chapter 7.3 of [27]).*

Proof. page 387 – 388, chapter 7.3 of [27]. ☐

Theorem 3.7.2. *The inference system $\mathcal{I}$ is refutation complete for ground Horn clauses (chapter 7.3 of [27]).*

Proof. Let S be a set of ground Horn clauses, closed under the inference system $\mathcal{I}$ and $\square \notin S$. We show that S is satisfiable. For this we prove that R^* is a model for S. We perform induction on $\succ_C$ to obtain a contradiction from the existence of a minimal clause C in S such that $R^* \nvDash C$. A detailed demonstration of the proof can be found on page 388 – 389, chapter 7.3 of [27]. □

In this framework we can assume that equality as the only predicate, because it is possible to write every other predicate symbol $p(t_1, \ldots, t_n)$ as an equation $p(t_1, \ldots, t_n) = true$. Now we consider *true* as a new special symbol and p as a function symbol. It is quite easy to understand that this transformation preserves satisfiability. For details we recommend to read chapter 7.3.2 of [27].

What about clauses with variables? We take into consideration that a non-ground clause is a representation of the set of all the ground instances of the clause. To construct a refutation complete method for non-ground clauses we could systematically numerate all ground instances. Then we only perform inferences by $\mathcal{I}$ between those instances. Additionally, it is also allowed to perform inferences between non-ground clauses. We can now adjusted to match this idea. To see the adapted rules check page 392 of chapter 7.2 in [27].

In this last step we consider general clauses, i.e., a clause may have several equations in its succedent. Again we adapt the inference system $\mathcal{I}$. To keep the number of inferences restricted, the notion of *strictly maximal* terms gets introduced. Among several maximal equations we choose the one whose other side is maximal (chapter 7.3 of [27]). The full inference system for general clauses contains the following rules already defined before: equality resolution, equality factoring, superposition left and superposition right.

The following definition lists all the new aspects for the completeness proof of the inference system $\mathcal{I}$.

Definition 3.7.5. Let S be a set of ground clauses and let C be a clause in S. Then $Gen(C) = \{s \Rightarrow r\}$ and C *generate* the rule $s \Rightarrow t$ if and only if C has the form $\Gamma \rightarrow s = t$ and these three conditions are fulfilled:

1. $R_C^* \nvDash C$
2. $s \succ t$, $s \succ u$ for all u in Γ and $s = t$ is a maximal equation in Δ
3. s is irreducible by R_C
4. $R_C^* \nvDash r = v$ for every $s = v \in \Delta$

where $R_C = \bigcup_{C \succ_C D} Gen(D)$, otherwise $Gen(C) = \varnothing$ and $R = \bigcup_{C D \in S} Gen(D)$ (i.e. the set of all rules generated by S) (chapter 7.3 of [27]).

Similar as before, we first define a lemma to help us prove the completeness:

Lemma 3.7.3. *If a clause C of the form $\Gamma \rightarrow s = t, \Delta$, generates the rule $s \Rightarrow t$ then $R^* \vDash C$ and $R^* \nvDash \Delta$ (chapter 7.3 of [27]).*

Theorem 3.7.4. *The inference system $\mathcal{I}$ is refutation complete for general clauses (chapter 7.3 of [27]).*

Proof. Roughly speaking, the proof of theorem 3.7.2 gets extended by one additional case (due to the new conditions for generating rules). The other cases get adapted by the lemma 3.7.3 in order to show that the conclusion of the required inference is not satisfied by the model. A detailed demonstration of the proof and the new case can be found on page 396, chapter 7.3 of [27] □

3.8 Special Theories

In this chapter, we discuss methods to integrate special theories of datatypes such as integers or arrays in a first-order theorem prover. We focus on two different options — the *axiomatiation* of the theory and the integration of *SMT solvers* in the prover. To this effect we give a sample axiomatisation of the linear integer arithmetic and of functional arrays with extenionality. We explain how we integrated SMT solvers in our prover and additionally we outline how the first-order theorem prover SPASS+T [22] mastered this challenge.

3.8.1 Axiomatisation

We already heard in chapter 2 "State of the Art" that combining first-order logic theorem proving with special theories is pretty difficult. One of the first saturation-based approaches for reasoning with theories was SPASS+T [22], which implements different ideas, for example simplification of theory terms and the introduction of theory axioms. Vampire [45] adapted both of this methods. In this section we will focus on the so-called *axiomatisation*. This method consists of adding additional theory axioms to the input problem that define the theory (a theory axiom is a statement that is true in the relevant theory). In this thesis we focus on integers and arrays with extensionality. We also give an (incomplete) sample axiomatisation of these two theories [24].

One problem of adding theory axioms is, that when they combine with each other, they can lead to an explosion of the search space really quickly. Especially the associativity and commutativity axioms lead to great inefficiency. Vampire and also other solvers have explored some techniques for minimizing this problem [23] [39].

Linear Integer Arithmetic

The following theory axioms state several important properties of the set of integers $\mathbb{Z}$, but this axiomatisation is not complete; in essence, we mainly add the axioms that Vampire does [24] [23].

First we want to point out two special constants $0, 1 \in \mathbb{Z}$. 0 denotes the so-called additive identity element whereas 1 describes the multiplicative identity element. Both properties get described below.

Consider the binary operation $+$ on $\mathbb{Z}$ denoting addition and the unary operation $-$ denoting negation, which satisfy the following statements:

- $\forall a, b \in \mathbb{Z}\ (a + b = b + a)$ (commutativity)
- $\forall a, b, c \in \mathbb{Z}\ (a + (b + c) = (a + b) + c)$ (associativity)
- $\forall a \in \mathbb{Z}\ (a + 0 = a)$ (zero element, also referred to as additive identity)
- $\forall a \in \mathbb{Z}\ (a + (-a) = 0)$ (additive inverse)

Remark. We define the subtraction of integers by adding up negative values, i.e., $(a-b) = a + (-b)$. Therefore we do not define any specific axioms for subtraction.

Naturally, we also consider the binary function $\cdot$ denoting multiplication, which has the following properties:

- $\forall a, b \in \mathbb{Z}\ (a \cdot b = b \cdot a)$ (commutativity)
- $\forall a, b, c \in \mathbb{Z}\ (a \cdot (b \cdot c) = (a \cdot b) \cdot c)$ (associativity)
- $\forall a \in \mathbb{Z}\ (a \cdot 1 = a)$ (identity element)
- $\forall a \in \mathbb{Z}\ (a \cdot 0 = 0)$ (zero multiplication property)

Additionally we state some further properties:

- $\forall a \in \mathbb{Z}\ (-(-a) = a)$ (properties of negatives)
- $\forall a, b \in \mathbb{Z}\ (-(a + b) = ((-a) + (-b)))$ (properties of negatives)
- $\forall a, b, c \in \mathbb{Z}\ ((a \cdot b) + (a \cdot c) = a \cdot (b + c))$ (distributivity)
- $\forall a, b \in \mathbb{Z}\ (a < b \lor b < a \lor a = b)$ (trichotomy principle)
- $\forall a \in \mathbb{Z}\ \neg(a < a)$
- $\forall a, b, c \in \mathbb{Z}\ (\neg(a < b) \lor \neg(b < c) \lor \neg(a < c))$
- $\forall a, b, c \in \mathbb{Z}\ (\neg(a < b) \lor (a + c < b + c))$

Remark. All of these statements follow from basic properties of the integers. We add them because it shortens the proofs.

Functional Arrays With Extenionality

Since arrays are omnipresent in programs, every solving engine should be able to reason with arrays. In the following we present a sample axiomatisation of arrays with extentionality. The theory of arrays is parametrised by two sorts: the index sort ι and the value sort ν. In detail, the theory of arrays is a first-order theory that consists of the following three entities (again we follow the notion from Vampire [15]):

- a sort $array(\iota, \nu)$
- a function symbol $select\colon array(\iota, \nu) \times \iota \to \nu$ (this function symbol describes a binary operation to find an element in the array according to its index [15])
- a function symbol $store\colon array(\iota, \nu) \times \iota \times \nu \to array(\iota, \nu)$ (this function symbol describes a ternary operation to update an array with a given value at a given index [15])

The functions are characterized by three array axioms which are added by the prover to the input problem [15]:

- $(\forall a \in array(\iota,\nu))(\forall v \in \nu)(\forall i \in \iota)(\forall j \in \iota)(i = j \Rightarrow select(store(a,i,v),j) = v)$
  (read-over-write 1)
- $(\forall a \in array(\iota,\nu))(\forall v \in \nu)(\forall i \in \iota)(\forall j \in \iota)(i \neq j \Rightarrow select(store(a,i,v),j) = select(a,j))$
  (read-over-write 2)
- $(\forall a \in array(\iota,\nu))(\forall b \in array(\iota,\nu))(\forall i \in \iota)(select(a,i) = select(b,i) \Rightarrow a = b)$ (extentionality)

3.8.2 Satisfiability Modulo Theories

In the past few years the field of theory reasoning has been dominated by SMT solvers. In chapter 2 “State of the Art” we already heard that integrating a SMT solver into a first-order theorem prover is a possibility to incorporate special theories; we also gave a general introduction of SMT solving. Now we collect strategies how to use SMT solvers properly and efficient in our first-order prover.

Consider a given-clause algorithm (shortly described in chapter 2). We want to extend such a saturation algorithm by using a SAT solver. The solver should then check for inconsistencies within sub-problems of the input problem. The sub-problems get selected iteratively. To make this checking possible, the initial clause set as well as all subsequent clauses get propositionally abstracted. This abstraction is then given to the SAT solver. The described process bears resemblance to the utilisation of SAT solvers in SMT solvers, with the difference of getting applied to the full problem, not only to the ground parts. Since we want to preserve soundness we have to only break down clauses into variable-disjoint sub-clauses (chapter 3.3 of [24]).

Example 3.8.1. Let us discuss the example on page 7 in chapter 3.3 of [24]. We consider the clause $s(x) \vee \neg r(x,z) \vee \neg q(w)$. This clause is split into the two sub-clauses $A = (s(x) \vee \neg r(x,z))$ and $B = (\neg q(w))$, each is propositionally abstracted. In the sub-clause A $s(x)$ and $\neg r(x,z)$ share x and z whereas the sub-clause B is disjoint [24].

What happens when a set of sub-clauses with propositional names $a_1, \dots, a_n$ is shown to be inconsistent? Then the propositional clause $\neg a_1 \vee \dots \vee \neg a_n$ is *learned* (passed to the SAT solver) and we select a new sub-problem by running the SAT solver again (chapter 3.3 of [24]).

Since we want to achieve theory reasoning, the SAT solver gets replaced by an SMT solver. We have to be aware of the following difference: Previously, sub-clauses were translated to propositional symbols. Now only non-ground sub-clauses are translated in that way. Ground sub-clauses (which are necessarily unit ground clauses) are mapped to themselves. This ensures that the sub-problem that gets explored by the first-order saturation-based proof search is theory-consistent in the ground part (chapter 3.3 [24]).

Let us also consider the way SPASS+T [22] integrated an arbitrary SMT procedure for arithmetic symbols into their superposition-based theorem prover SPASS [42]. We briefly discuss how the system works: As usual, SPASS generates formulas using its deduction rules. The system also propagates all formulas that can be handled by the SMT solver to this solver (all ground formulas). Therefore, the ground formulas get processed by both — the SMT solver and the first-order prover. The problem is reckoned as solved as soon as one of the two systems found a contradiction. This can be viewed as a special case of hierarchical theorem proving (a combination of a base prover and an extension prover) (chapter 1 of [22]).

SPASS+T consists of three modules that interact with each other. First, the theorem prover SPASS, second, a SMT procedure and third, a module SMT-Control for connecting the other two. The two proof systems only wait for each other if they have finished their own task. The "communication" works via messages (three types of message) that SPASS sends to the SMT-Control (chapter 3 of [22]):

1. All ground formulas of the input problem and all ground formulas derived during the saturation process get sent,
2. if SPASS has been able to derive the empty clause ("find a proof"), and
3. if SPASS has finished saturating the input.

The SMT-Control has the task to collect all ground formulas that get sent and then sends repeatedly portions of this set of formulas to the SMT solver. This is done, until SPASS has found a proof, or the SMT procedure has detected unsatisfiability, or the SMT solver detects satisfiability (after SPASS finished saturating the input problem) (chapter 3 of [22]).

Unfortunately, SPASS might not produce this ground formulas (by the usual inference rules) that would be needed for the SMT solver to process the sub-problems and find a contradiction. This problem can be solved by adding an additional inference rule that computes the required ground instances. To increase the efficiency, SPASS+T integrated a special restriction: Only those instantiations are used where a term that is headed by a non-theory symbol occurs also in at least one other ground clause. If such a term is only contained in one clause, this clause is probably not very useful for the SMT procedure anyway (for linear arithmetic at least) (chapter 5 of [22]). This gives rise to the following rule

$$\frac{C[s] \qquad L[t] \vee D}{(L[t] \vee D)\sigma}\ \text{Theory instantiation}$$

where $L[t] \vee D$ is not ground, t is headed by a non-theory function symbol and occurs in a maximal literal $L[t]$ immediately below a theory symbol, all function or predicate symbols that occur above t in $L[t]$ are theory symbols or the equality, $C[s]$ is ground, σ is a MGU of s and t and $(L[t] \vee D)\sigma$ is ground (chapter 5 of [22]).

Depending on the implementation of redundancy elimination criteria, there might arise a problem. The formula $(L[t] \vee D)\sigma$ gets subsumed by the premise $L[t] \vee D$ and will therefore be deleted again. SPASS evades this by exporting the formula to the SMT solver before the redundancy check is performed. Additionally, it has to be possible, that $(L[t] \vee D)\sigma$ can be used as a first premise in the theory instantiation inference rule again. This can for example be done by splitting the inference rule into two separate rules and a new special predicate symbol (for example called “ground”) (chapter 5 of [22]).

Chapter 4

Design and Implementation of the Prover

In this chapter we describe how we designed our automated theorem prover for first-order logic and justify our decisions. Inspired by the E theorem prover [44] a saturation algorithm tries to derive the empty clause — and therefore detect the unsatisfiability of a problem — using an instance of the superposition calculus. Additionally, we give a detailed insight into our implementation.

4.1 Calculus

In general our prover is based on a variation of the superposition calculus, described in subsection 3.7.3, with literal selection. Additionally, we use explicitly defined inference rules for simplification. Like the theorem prover E [44] we use so-called pure superposition, which is a refutation-based saturation calculus only working with the equational representations of formulae (in clause normal form) [38]. We denote the calculus used in our prover by $\mathcal{SP}$.

Definition 4.1.1. Let A be a non-equational atom. Then the equation $A = \top$ is the equational representation of A, where $\top$ is a special function symbol.

The initial *proof state* gets derived from the problem formalization and throughout the algorithm is represented by a set of equational clauses. The so-called *proof derivation* is the process of systematically infering new clauses and adding them to the proof state. This proceeds until the empty clause has been found and therefore the unsatisfiability of the set has been shown. Additionally it is possible to simplify clauses or to remove redundant clauses from the proof state [38].

It is important to mention, that in this chosen calculus two clauses are not allowed to share any variables. All clauses have to have distinct variables. Of course also newly generated clauses need to have distinct variables to all existing variables.

4.1.1 Generating Inferences

The superposition calculus as we design it restricts generating inferences to maximal literals. It is also possible to restrict the generating inferences to *selected literals*. In this case a selection function arbitrarily selects certain literals in clauses which contain at least one negative literal, (but this is not realized in the proof design in this thesis). In the following we assume a literal selection function *sel* and a ground reduction ordering > (lifted to literals) [38]. For the term ordering > we decided to implement the lexicographic path ordering, shortened LPO. The LPO fulfils the needed properties of being a reduction ordering and it is a.lso used by many well-known state of the art provers including the E Theorem prover [44]. Choosing the "correct" ordering for each proof problem can be quite a good heuristic approach to fasten the proof search.

Definition 4.1.2. Let > be a binary relation on terms. If > is transitive, irreflexive and closed under instantiation and simple congruences (within a fixed set of function symbols), then > is a rewrite order. If > is additionally terminating, then > is called a reduction order [13].

Definition 4.1.3. [13] Let t and u be two terms. Then $t > u$ is defined as follows:

- $t > u$, if u is a proper subterm of t
- $f(t_1, \dots, t_n) > t$, if $t_i > t$ for some i
- $f(t_1, \dots, t_n) > f(u_1, \dots, u_n)$, if $t_i > u_i$ for some i and $t_j = u_j$ for all $j < i$
- $f(t_1, \dots, t_n) > g(u_1, \dots, u_m)$, if $f > g$ for some ordering of function/constant symbols

The last two rules require the additional condition$f(t_1, \dots, t_n) > u_i$ for every i.

In order to be able to give an exhaustive explanation of the generating inferences we have to define some important preconditions we need:

Definition 4.1.4. Let $C = l \vee R$ be a clause and σ a substitution. Let C^- denote the subset of negative literals of C and C^+ the subset of positive literals of C. If one of the following conditions is fulfilled

- $sel(C) = \varnothing$ and $\sigma(l)$ is >-maximal in $\sigma(C)$
- $sel(C) \neq \varnothing$ and $\sigma(l)$ is >-maximal in $\sigma(C) \cap C^-$
- $sel(C) \neq \varnothing$ and $\sigma(l)$ is >-maximal in $\sigma(C) \cap C^+$

we call $\sigma(l)$ *eligible for resolution* (passive inference partner). If $l \in C^+, sel(C) = \varnothing$ and $\sigma(l)$ is maximal in $\sigma(C)$, then $\sigma(l)$ is called *eligible for paramodulation* [38].

Remark. Although we decided not to include literal selection in our proof design to stay within the scope of this thesis, we still follow the notion of the preconditions eligibility for resolution and paramodulation as the E Theorem prover [38]. This approach ensures easy integration of a selection function at a later time.

The following rules are the four generating inference rules that are used in our prover [38] (the same inferences are used in the theorem prover E [44]):

$$\frac{s \neq t \vee R}{\sigma(R)} \text{ (Equality resolution (ER))}$$

where $\sigma = \text{mgu}(s, t)$ and $\sigma(s \neq t)$ is eligible for resolution.

$$\frac{s = t \vee S \qquad u \neq v \vee R}{\sigma(u[p \leftarrow t] \neq v \vee S \vee R)} \text{ (Superposition into negative literals (SN))}$$

where $\sigma = \text{mgu}(u|_p, s), \sigma(s) \nless \sigma(t), \sigma(u) \nless \sigma(v), \sigma(s \neq t)$ is eligible for paramodulation, $\sigma(u \neq v)$ is eligible for resolution and $u|_p \notin V$.

$$\frac{s = t \vee S \qquad u = v \vee R}{\sigma(u[p \leftarrow t] = v \vee S \vee R)} \text{ (Superposition into positive literals (SP))}$$

where $\sigma = \text{mgu}(u|_p, s), \sigma(s) \nless \sigma(t), \sigma(u) \nless \sigma(v), \sigma(s \neq t)$ is eligible for paramodulation, $\sigma(u \neq v)$ is eligible for resolution and $u|_p \notin V$.

$$\frac{s = t \vee u = v \vee R}{\sigma(t \neq v \vee u = v \vee R)} \text{ (Equality factoring (EF))}$$

where $\sigma = \text{mgu}(s, u), \sigma(s) \nless \sigma(t)$ and $\sigma(s \neq t)$ is eligible for paramodulation.

Is this calculus complete? Remember the inference system ε from [2] (also described in section 3.7.2 of this thesis). Our system $\mathcal{SP}$, containing the inference rules (ER), (SN), (SP) and (EF), subsumes ε with literal selection. This means that for every $\mathcal{SP}$-selection function there exists a version of ε that allows at most the same inferences as $\mathcal{SP}$ does. We already know that the inference system ε is complete, therefore it follows that also $\mathcal{SP}$ is complete. Additionally, it is possible to apply all general redundancy elimination schemes for the standard superposition calculus [38].

4.1.2 Redundancy Elimination

To achieve a reasonable runtime for our prover, we need to implement good simplifying operations (also called contracting operations) to remove or modify already existing clauses. Most modern provers do this even though it is theoretically not necessary — the generating inferences are sufficient [38] for completeness.

An important property which underlies all simplifying inference rules is called *compositeness* [2]. Composite clauses can be deleted from the proof state without any affects on the completeness of the calculus. An example of composite clauses are tautologies [38].

Definition 4.1.5. Let C be a clause and F a set of clauses. If all ground instances of C are implied by smaller ground instances from clauses in F, then C is called *composite* with respect to F [38].

Note that inferences where composite clauses are involved are redundant. We speak of *deleting inferences* if the rule deletes a composite clause. If a new and simpler clause is derived we speak of *modifying or simplifying inferences* [38].

To avoid any confusion, a short recall on the from of a simplification inference rule used in this thesis (also used in [38]):

$$\frac{\textit{precondition}}{\textit{conclusion}} \text{ if } \textit{condition}$$

The rule is applicable if the condition is fulfilled. This rule changes the proof state, by replacing all clauses in the precondition with the clauses in the conclusion. If the conclusion is empty, we deal with a deleting inference rule [38].

Two basic contraction rules to eliminate unnecessary literals from a clause are deletion of duplicated literals and deletion of resolved literals:

$$\frac{s = t \vee s = t \vee R}{s = t \vee R} \text{ (Deletion of duplicated literals (DD))}$$

$$\frac{s \neq s \vee R}{R} \text{ (Deletion of resolved literals (DR))}$$

Since superposition is also compatible with more sophisticated deletion rules for tautologies we also implement some of these. Especially for comprehensive proof problems this is quite important, because it is often the case that many clauses can be rewritten to equational tautologies. Considering the equational logic, the question, if a clause is tautological is undecidable. Two criteria to detect tautologies are [38]:

1. If a clause contains a trivial equation $s = s$
2. If a clause contains two identical atoms with opposite signs (equality and inequality)

The following two inference rules encode these criteria [38]:

$$\frac{s = s \vee R}{} \text{ (Syntactic tautology deletion 1 (TD1))}$$

$$\frac{s = t \vee s \neq t \vee R}{} \text{ (Syntactic tautology deletion 2 (TD2))}$$

If the equational theory induced by the set of negated negative literals implies one of the positive literals, then the clause is tautological. Encoded in a contraction rule [38]:

$$\frac{s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n \vee s = t \vee R}{} \text{ (Semantic tautology deletion (SD))}$$

if $\sigma(s_1 = t_1), \ldots, \sigma(s_n = t_n) \models \sigma(s = t)$, where the substitution σ maps all variables to distinct new constants.

These three criteria combined are sufficient for detecting tautological clauses. Note, that due to our data structure in the program we decided not to implement this particular rule (SD) in our prover, since the implementation would be linked to a very unhandy and slow algorithm, which subsumes the advantage of deleting the clause nearly completely.

Additionally, the following rule implements a special case of the semantic tautology deletion. The inference rule "destructive equality resolution" also represents a special case of equality resolution. This rules enables us to destructively solve a negative literal with one variable side, only by applying a substitution to the clause [38]:

$$\frac{x \neq s \vee R}{\sigma(R)} \text{ (Destructive equaltiy resoltuion (DER))}$$

if $x \in V$ and $\sigma = mgu(x, s)$.

Remark. All these simplifying inference rules introduced so far are called *intra-clause simplification* [38].

Another crucial group is the so-called *clause-clause simplification*, especially rewriting is quite an efficient way to derive many functions defined by sets of equations. Rewriting is a so-called modifying inference mainly coming from the field of the Knuth-Bendix completion [32]. We differ between rewriting positive literals and negative ones [38]:

$$\frac{s = t \qquad u = v \vee R}{s = t,\, u[p \leftarrow \sigma(t)] = v \vee R} \text{ (Rewriting of positive literals (RP))}$$

if $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$ for a substitution σ.

Remark. In the construction of the E Theorem prover [44] as well as the corresponding literature [38] the application of this rule gets more restricted by adding additional preconditions. In [38] the preconditions are the following way: If $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$ for a substitution σ and if $u = v$ is not eligible for resolution or $u \not> v$ or $p \neq \lambda$ or σ is not a variable renaming. This more sophisticated precondition only prevents unnecessary applications of the inference rule, but does not interfere with the completeness of the calculus. Due to this fact, as well as the lack of parentheses (making the statement ambiguous) and a missing declaration of λ we only included the necessary preconditions in our design for the prover (as stated in the rule above).

$$\frac{s = t \qquad u \neq v \vee R}{s = t,\, u[p \leftarrow \sigma(t)] \neq v \vee R} \text{ (Rewriting of negative literals (RN))}$$

if $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$ for a substitution σ.

While we are only able to apply rewriting to orientable clauses, it is also possible to use unorientable clauses for simplification. Let $s = t$ be a clause. This clause is called orientable, if $s > t$ for an ordering $>$ [38].

$$\frac{s \neq t \qquad \sigma(s = t) \vee R}{s \neq t,\, R} \text{ (Negative simplify-reflect (NS))}$$

for a substitution σ.

For symmetry reasons there is also a rule applying to positive literals (so-called "positive simplify-reflect" (PS)). In our program it is not possible to implement this rule reasonably efficient, due to the chosen data structure. An integration of this rule would be linked to a significant increase of the runtime in contrast to a mild advantage of simplifying some clauses for the purpose of our prover. The formal declaration of this simplifying rule used for the E Theorem prover looks like this [38]:

$$\frac{s = t \qquad u[p \leftarrow \sigma(s)] \neq u[p \leftarrow \sigma(t)] \vee R}{s = t,\ R} \text{ (Positive simplify-reflect (PS))}$$

for a substitution σ.

In comparison to rewriting, subsumption represents another deleting simplifying inference rule, coming from the field of resolution. This enables the deletion of clauses which are already contained in the proof state in a more general (in some sense smaller) form [38].

$$\frac{T \qquad R \vee S}{T} \text{ (Clause subsumption (CS))}$$

if $\sigma(T) = S$ for a substitution σ.

A second subsumption rule implemented in the E Theorem prover is the so-called "equality subsumption", useful in the case of equational logic. It is induced by positive unit clauses [38]:

$$\frac{s = t \qquad u[p \leftarrow \sigma(s)] = u[p \leftarrow \sigma(t)] \vee R}{s = t} \text{ (Equality subsumption (ES))}$$

for a substitution σ.

As already explained for the simplifying rule "positive simplify-reflect", also for this rule we decided not to implement it in our own prover due to the data structure in the program and the associated increase in the runtime compared to little advantage of the simplification.

4.2 Search Organization

With the calculus constructed so far, it is possible to arbitrarily apply inference rules. This can lead to an explosion of the search space quickly. Therefore we need to create a complete algorithm which efficiently applies the inferences in a special order. Just like the E Theorem prover [44], our saturation algorithm consists of a main loop that combines generating rules with simplifying inferences [38]. As most of the current saturation high performance theorem provers, our prover uses a variant of the *given-clause-algorithm* (see chapter 2.2).

In detail, we design our proof procedure around the DISCOUNT loop. In the construction of our algorithm the proof state is represented by two sets of clauses, U representing the unprocessed clauses and P collecting the already processed ones. It results, that initially U contains all clauses and P on the other hand is empty. With each loop traversal a new *given clause* is selected from the set U. This clause gets simplified with respect to the set P. In return, this clause is used to simplify the clauses in the set P. Note, that when using the DISCOUNT loop, only processed clauses participate in the simplifying process. Afterwards, the algorithm computes all possible generating inferences using the rules equality factoring, equality resolution and superposition. At least one clause has to be the given clause and possible other participants are arbitrary clauses from the set P. The results get simplified and added to the set U, if they are not trivial. The algorithm terminates, if the empty clause was found (proving unsatisfiability) or the set U is empty and therefore no more inferences are possible. In this case the set was satisfiable [38].

Please note, the goal is to prove the validity of a formula F. To achieve this certain steps are followed by the program:

1. Want to show the unsatisfiability of $\neg F$
2. Transform $\neg F$ to an equi-satisfiable set of clauses C
3. The algorithm tries to show the unsatisfiability of C. There are three possible outcomes:
    - "Success": F is valid
    - "Failure": all possible resolvents were generated, but not the empty clause; In this case F is invalid (a model can be generated)
    - "non-termination": in practice "timeout"; In this case nothing about the (in)validity of F is known

Remark. Our main goal is not to provide a high performance prover, that can compete with already existing provers such as the E Theorem prover [44], Vampire [45] or SPASS+T [22]. A design and implementation of such an prover would exceed the scope of a master thesis. The main focus is to provide an extension for RISCTP, which can be used to learn and study the development of a proof and provide more insight information than other existing provers.

The following pseudo code shows the concept of the algorithm for our prover (similar to the algorithm of the E Theorem prover described in [38]):

Algorithm 4.1 Discount algorithm for resolution prover

$U := \{\textit{initial set of input clauses}\}$
$P := \emptyset$
while $U \neq \emptyset$ **do**
    c :=select_best(U)
    $U := U \setminus \{c\}$
    c :=simplify(c, P, $=$, $\neq$)
    if not redundant(c, P, $=$, $\neq$) **then**
        if if c is the empty clause **then**
            success; clause set is unsatisfiable
        else
            for for each $p \in P$ **do**
                p :=simplify($p, (P \setminus \{p\}) \cup \{c\}, =, \neq$)
                if if c is the empty clause **then**
                    success; clause set is unsatisfiable
                end if
            end for
            $P := P \cup \{c\}$
            $T :=$ generate(c, P, $=$, $\neq$)
            for for each $p \in T$ **do**
                if if c is the empty clause **then**
                    success; clause set is unsatisfiable
                end if
                p :=cheap_simplify(p, P, $=$, $\neq$)
                if if c is the empty clause **then**
                    success; clause set is unsatisfiable
                end if
                if not trivial(p, P, $=$, $\neq$) **then**
                    $U := U \cup \{p\}$
                end if
            end for
        end if
    end if
end while
Failure: Initial U is satisfiable, P describes model

In this pseudo code and also in the algorithm we use some subfunctions [38]:

select_best(U)

This function selects the so-called *given clause*. This is the "best" clause from the set of unprocessed clauses U. The best clause gets defined by a heuristic evaluation function. The evaluation function basically calculates a "score" for every clause. This score is then used to select a clause. In our program, the select_best function always selects the first element of the set U, because U is sorted according to the evaluation and new clauses get inserted at the correct position [38]. We go into more detail about the heuristic in the next section.

simplify(c, P)

This function is applied to simplify clauses. It performs all possible simplifying inferences between the clause c (the clause that gets modified, also called *main premises*) and all possible other clauses from the set P of already processed clauses. In the first step, we use it to simplify the given clause with the set of already processed clauses, the second application is to simplify the set of processed clauses using the given clause. The following inference rules get applied in the stated order: RP, RN, NS, DD, DR and DER (see subsection 4.1) [38].

cheap_simplify(c, P)

This method is very similar to $simplify(c, P)$ and is used to simplify newly generated clauses. To do this, we only use the rules DD and DR, since they are implemented quite efficiently [38].

redundant(c, P)

This function returns true, if and only if the clause c is subsumed by any of the clauses in the set P (using the rule CS). Redundant clauses are no longer needed, they are deleted from the set U of unprocessed clauses and are not added to the set P of processed clauses [38].

trivial(c, P)

This subroutine is very similar to $redundant(c, P)$. It tests, if the clause c is trivial with respect to the clauses in the set P. It returns *true* if and only if it is possible to show redundancy of the clause c with respect to the set P. The check is carried out with the rules TD1 and TD2 [38].

generate(c, P)

This function uses the rules ER, EF,SN and SP (in this order) to compute all possible conclusions between the clause c and the clauses in the set P. At least one premise has to be c [38].

The design of this algorithm has several advantages. For example, at the start of each loop traversal, all generating inferences between clauses in P have been computed (general given-clause invariant). At the same time P is maximally simplified with respect to itself, also referred to as *interreduced*. Additionally, clauses in U are truly passive, therefore the only time they are used is when they get selected as the given clause. Compared to the Otter loop [19] the *select_best*() function is of great importance. The sets P and T contain a relatively small amount of clauses and since only those two sets are involved in costly steps (T denotes the set of all newly generated clauses in this traversal). The number of processed clauses within a certain time is quite high. On the other hand, there might be clauses in U that would be helpful for rewriting and therefore might lead to a (quicker/shorter) proof. This disadvantage can be diminished by the higher inference rate (more clauses get processed in less time) and due to the heuristic evaluation function smaller, more useful clauses can get selected first. Most clauses in U are large and specific, therefore not useful for rewriting [38].

4.2.1 The Heuristic of Clause Selection

It is important for fully automated theorem provers for first order logic to have good search control, especially for the ones based on the DISCOUNT loop as ours. It turns out that clause selection, literal selection and the selection of the term ordering are probably the three most important choices to make [38]. Regarding the term ordering we already explained and defined the lexicographic path ordering in the section before. As already pointed out, we decided not to implement literal selection, therefore we are left to reasonably define the clause selection process.

Choosing the order in which the clauses get processed might be the most important decision criterion. As in [38] we came to the conclusion, that there is no background information about favouring some strategies over others, respectively a theoretical argumentation. Therefore we offer different options of clause selection functions to choose from for experimental work. A very easy pseudo code for a simply selection function is given in [38] and looks like this:

Algorithm 4.2 Simple example version of the method select_best()

select_best(U)
 $e := \min_{>}\{\text{eval}(c) \mid c \in U\}$
 select c arbitrarily from the set $\{c \in U \mid \text{eval}(c) = e\}$
return c

Generally, the evaluation function is applied to every clause (initially to all in U and later to every newly generated one) to calculate its evaluation (also called score in this thesis). The select_best method chooses the clause with the lowest score. Between clauses with the same score the function chooses randomly. Those scores are only computed once and stored together with the clauses. The set U is sorted once according to the evaluations (priority queue) right at the beginning of the algorithm. New clauses get inserted at the correct position in U [38].

Remark. It is important for the completeness of the calculus to follow the notion of *fairness* when defining (and implementing) the evaluation function. In other words, every clause has to get a fair chance to get picked at some point in the algorithm. For example, if symbol counting is used as an evaluation function, it is possible, that infinitely many clauses of weight 5 get generated, so a clause of weight 10 never gets picked.

In our prover we implemented three different types of an evaluation function:

- clause weight
- first-in-first-out (FIFO)
- combination of clause weight and FIFO

The so-called *clause weight function* function is based on symbol counting and is very common. This evaluation returns the number of function symbols and variables and assigns a specified weight to the different symbol types. Therefore with this evaluation smaller clauses get preferred. Small clauses have several advantages. Since they encode more general concepts they are very useful for simplifying inferences and are in general applicable in more situations. Also, smaller clauses tend to lead to fewer descendants and therefore the extreme increase of the search space gets delayed [38]. Please note that using only symbol counting effects the completeness of the prover as described in the remark above.

The second option for an evaluation function is a so-called *FIFO weight function*. This function implements a first-in-first-out mentality and therefore simply returns the number of an increasing counter for every new clause. Generally, this heuristic always finds the shortest proofs according to the inference depth. This is due to the fact that the clauses get enumerated in order of increasing depth [38]. Also this type of evaluation function leads to an incomplete prover.

To provide a fair evaluation function (and therefore an option for a complete prover) we designed a combination of the clause weight function and the FIFO weight function. The basic idea behind this is, to set up two priority queues. One queue sorted by the clause weight evaluation and the other queue sorted by age (FIFO). In the process of picking the given clause (select_best()) we alternately pick clauses from those two queues.

4.3 SMT Solver Integration

To extend the designed prover we also integrated an external SMT solver. For this purpose the SMT solver Z3 [10] is used. The access is already provided in the existing implementation of the RISCTP software [35]. The SMT solver Z3 [10] supports the resolution prover in a quite direct way. At the beginning of the loop traversal the SMT solver is asked to try and prove the problem using the current proof state (processed clauses and given clause). The SMT solver gets called at every n-th loop traversal of the main loop ($n \in \mathbb{N}$ and $n > 0$). The algorithm terminates, if a proof was found. If the SMT solver could not find a proof, the SMT prover is again queried but now with the ground (variable-free) instances of the clauses. The algorithm proceeds normally, if also this attempt fails. In general, no information about the reason why the proof failed is further processed.

4.4 Implementation

Our prover is build as an extension of the RISCTP software [35] under the GNU general public license and implemented in Java using the eclipse environment. The user can work with the prover via the RISCTP interface and the RISCTP user syntax [36]. In this chapter we summaries our implementation and give a detailed insight on how the program is built. Throughout this chapter we write statements, variables and function names used in the implementation in this special `font` for clarification.

4.4.1 Class Structure

In this subchapter we want to give an overview about the overall structure of the program, i. e. the several different classes, that together build our prover. The goal at this point is to create an understanding of the concept and the relation between classes, before we dive deeper into the implementation in the following sections.

Main.java

This class represents the intersection to the RISCTP software and all its current features. It was already existent and prepared from Wolfgang Schreiner before this thesis was written. When starting a prove, this class calls the method `solve()` from the class Resolution.

Resolution.java

In this class the main algorithm for the prover, defined in the pseudo code in 4.1, is implemented and carried out together with the defined subroutines and functions to print a proof tree. The method `solve()` represents the main algorithm and executes out the proof. We will go through this algorithm in detail in the following sections.

PureEquation.java

A PureEquation defines the data type we use throughout our implementation to represent a clause. The input problem gets transformed to a set of PureEquations and also gets further processed in this data type. We explain this data type extensively in the next section. This class also implements a function to test maximality according to the lexicographic path ordering.

GeneratingInferences.java

Here we define the generating inference rules ER, EF, SP and SN as well as all preconditions. This class consists of a clause (PureEquation), the two function symbols for (in-)equality and an ordering for the function symbols, needed for the LPO. The methods in this class are used for generating new clauses under the subroutine `generate(c, P)`.

SimplifyingInferences.java

The setting of this class is designed analogously to GeneratingInferences.java, additionally tracking the underlying substitution. Here however, we define all the defined simplifying inference rules (DD, DR, TD1, TD2, SD, DR, RP, RN, NS, PS, CS and ES) alongside with all preconditions. The methods in this class are used when calling the subroutines `simplify(c, P)`, `cheap_simplify(c, P)`, `trivial(c, P)` and `redundant(c, P)`.

4.4.2 Representation of Clauses

In this chapter we introduce the data structure and clausal representation we use in our algorithm to represent clauses, terms and literals. As already stated, the definition of the data type used is defied in the class PureEquation. Before even entering our prover (and therefore the program written in the scope of this thesis), the input problem of first order logic clauses gets transformed by the RISCTP software to *negation normal form*. In our algorithm, this clauses get translated to PureEquations.

We only work with the equational representation of each clause, building the name of the class and data type. A PureEquation consist of the following parts:

- `name`: a string representation of the name of the clause
- `vars`: an array of variable symbols containing the quantified variables of the clause
- `lits`: an array of objects, representing the literals of the clause. Even tough the literals are of the data type java object, they are internally understood as *terms* (instances of the class Term.java) and treated as such
- `negs`: a boolean array, tracking the negation of each literal (negated literals, i. e. inequalities, get assigned true — meaning "yes, this literal is negated" —, equalities are shown as false)
- `select`: a list of the positions of the selected literals (integers); in our case this list is always empty, since we decided not to implement a selection function
- `score`: the integer score, an evaluation function assigns each to clause, needed for the `selectbest()` method and the ordering of the set of unprocessed clauses
- `age`: the age of the clause (first clause to be in the system has age 0, second clause has age 1, ...), needed for the `selectbest()`
- `generatedBy`: a string tracking the short cut of the inference rule that generated this clause, states "initial" for clauses coming from the input set
- `parents`: a list of PureEquations memorizing the parents of the clause, i. e. the clauses which generated it

Within the scope of a PureEquation we also have function and variable symbols (constants being function symbol with arity zero). The equality and inequality is represented by a predefined function symbol of arity two (already implemented within the existing program before my thesis). Similar for the representation of the $true$ and $false$. Those are constants and also already introduced in the program. We construct the empty clause by a PureEquation containing zero elements in the array `lits`.

The sets U and P of (un)processed clauses are represented by a list of PureEquations. In the case of the list U, the list is ordered in a specific way, explain in the next paragraph.

4.4.3 User Settings

The implementation of our resolution prover allows the user to set a few user preferences. This enables the user to experiment with different options and adds flexibility to adjust to certain situations. Those settings are done in class Resolution.java.

Simplification

Simplifying clauses is very crucial for a good saturation prover. We offer three different options for the application of simplification inferences (boolean variables set by true and false).

- `intraClauseSimpl`: With this option the user can choose if intra-clause simplification inference rules should be used or not. This includes the rules deletion of duplicated literals, deletion of resolved literals destructive equality resolution and syntactic tautology deletion 1 and 2. Note that this setting effects the subroutines `simplify()`, `cheapsimplify()` and `trivial()`.
- `clauseClauseSimpl`: Analogously, with this setting it is fixed if clause-clause simplification inference rules should be used or not, including the rules rewriting of positive and negative literals, negative simplify-reflect and clause subsumption. This effects the subroutines `simplify()` and `redundant`.
- `simplifyProcessed`: The last option is to decide, if the set `proc` of processed clauses should be simplified using the given clause. If not, this part gets skipped in the algorithm. This option was added, because for some examples the prover seemed to be faster if the processed clauses are not simplified.

Evaluation Function for the method selectbest()

We already explained before (chapter 4.2.1) that there are different options for the evaluation function, needed to select "the best" clause. Every evaluation function might lead to a very different runtime. For our prover we offer three different possibilities. For choosing the user has to either change the string object to the wanted option or comment in the correct line of code.

- `fifo`: Uses the first-in-first-out method. Every clause (PureEquation) gets an evaluation (age) assigned at the variable `score`. The first clause to appear in the algorithm gets age 0, the second clause age 2, . . . This is done with an global, increasing counter. Smaller age therefore means older clause. Using this method, the given clause chosen at each loop traversal is always the oldest clause from the set `unproc` of unprocessed clauses.
- `clauseweight`: This method uses symbol counting to calculate the score for each PureEquation (stored in the variable `score`). Using the two parameters `weightfuncsymb` and `weightvarsymb` the user is able to assign a weight to function and variable symbols. Smaller clauses have a smaller score. At each loop traversal the smallest clause (with the lowest score) gets chosen to be the given clause.
- `combined`: The last option combines the two strategies FIFO and clauseweight. Intentionally, the score of the clause is assigned according to the clause weight evaluation function (also the set `unproc` is sorted according to this score). Additionally we assign an age to every clause, using the variable `age` of the data type PureEquation. The given clause is then picked alternately according to these two criteria. To accommodate this, the user has to fix a frequency `freqUseOldClause` to set, how many small clauses are chosen before an old clause gets picked. For example: set

`freqUseOldClause` = 3; in the first two loop traversals the smallest clause is picked as the given clause and at the next, third loop traversal we chose the oldest clause in `unproc`, followed again by two small clauses etc.

Use of a SMT solver

To ensure also reasoning with special theories (integers, arrays) we offer the possibility two possibilities: 1) add theory axioms or 2) use a SMT solver. By setting the variable `smt` to true the user enables the use of a SMT solver. We currently offer a connection to the SMT solver Z3 [10]. Additionally, the user has to fix how often to apply the SMT solver (`freqUseSMT`). Setting `freqUseSMT` to 1 results in an application of the SMT solver at each traversal of the loop, setting it to 2 uses the solver in every second traversal of the main loop, etc.

4.4.4 Prerequisites

Before starting the main loop written down in the algorithm 4.1, we start with some preliminary arrangements. We are now focussing on the class Resolution.java and the therein included method `solve()`. The input to the program is a first-order logic formula (read from a text file). This formula gets negated and and transformed to a set of clauses. The prover then tries to prove the unsatisfiablity of this set. The method `solve()` then takes this set of clauses as the only input parameter and is of the type `ClauseProblem`. It basically contains the clauses in negation normal form (type `Clause`). Please note, that the manual input to the program (read from text file) is a first-order logic formula. Those clauses already store all the information we would need, but are very unhandy to use for our purpose. Therefore we transform them to our own data type PureEquation. According to the design of our prover it is required that every literal (in every clause) is an equation. We ensure this property within the transformation process. If a literal is already an equation (or an inequation) there is nothing to do. If a literal l is not an equation, we "force" it to become one by exchanging this literal by $l = true$ and $l \neq true$ for inequalities ($\neg l$) respectively.

After the transformation to PureEquations we ensure that all clauses have distinct variables. We do this in a quite simple way: There is a global variable `counter`, which is steadily increased with every variable identified. Running over all clauses and all variables in the clause and adding the `counter` to each variable. Should the variable already contain a counter (number), than we replace this current counter with our variable `counter`. We call this method `newInstance(PureEquation clause)`. This function is also called after a new clause was generated. The source code looks like this:

```java
/*****************************************************************************
 * Create a new variable name from a given name.
 * @param name the name of a variable.
 * @return a fresh variable name that does not appear in the proof yet.
 ****************************************************************************/
private String newVarName(String name)
{
  String name0;
  int i = name.indexOf('@');
  if (i == -1)
  {
    name0 = name + "@" + varCounter;
  } else
  {
    String name1 = name.substring(0, name.indexOf('@'));
    name0 = name1 + "@" + varCounter;
  }
  varCounter++;
  return name0;
}


/*****************************************************************************
 * Register a new instance of a rule with fresh names for the
 * quantified variables that do not appear in the proof yet.
 * @param rule the rule for which the instance is to be created.
 ****************************************************************************/
private PureEquation newInstance(PureEquation clause)
{
  VariableSymbol[] vars = clause.vars;
  int n = vars.length;
  VariableSymbol[] vars0 = new VariableSymbol[n];
  Subst subst = new Subst();
  for (int i = 0; i < n; i++)
  {
    VariableSymbol var = vars[i];
    Id id0 = Id.create(newVarName(var.id.toString()));
    VariableSymbol var0 = new VariableSymbol(id0, var.tsymbol);
    id0.setSymbol(var0);
    vars0[i] = var0;
    subst.put(var, var0);
  }
  Object[] lits = clause.lits;
  int litn = lits.length;
  Object[] lits0 = new Object[litn];
  for (int i = 0; i < litn; i++) lits0[i] = subst.apply(lits[i]);
  String name0 = clause.name;
  boolean[] negs0 = clause.negs;
  List<Integer> select0 = clause.select;
  int score0 = clause.score;
  String generatedby0 = clause.generatedby;
  List<PureEquation> parents0 = clause.parents;
  List<Integer> marked0 = clause.marked;
  PureEquation clause0 = new PureEquation(name0, vars0, lits0, negs0,
          select0, score0, generatedby0, parents0, marked0);
  return clause0;
}
```

Figure 4.1: Screenshot from the method "newInstances" to make all variables distinct

In the next step we fix an ordering for the function symbols in a list of function symbols, needed for the lexicographic path ordering (LPO). We order the function symbol according to their arity, meaning function symbol with greater arity are at the front of the list, constants are last (arity zero). Between symbols with the same arity we choose randomly.

Next we calculate an evaluation for every clause in the initial input set (equals the set of unprocessed clauses at the beginning of the algorithm). Our evaluation function calls either the "fifo" evaluation function the clauseweight one with the corresponding weights or the combination of these two. Those methods return a list with PureEquation with the `score` and `age` set accordingly.

```
// Returns a list of PureEquation;
// for every PureEquation the "score" and the "age" gets set
// This evaluation function represents a combination of the "clause weight" method
// and the fifo strategy. Clauses get an integer number as score (number of symbols
// - maybe weighted), smaller clauses get preferred additionally the age of the
// clause gets tracked by a continuous global counter, smaller age means older clause
public List<PureEquation> combinedClauseSelection(List<PureEquation> unprocessed,
        int weightfuncsymb, int weightvarsymb)
{
  List<PureEquation> combined = new ArrayList<>();
  for (PureEquation clause : unprocessed)
  {
    combined.add(new PureEquation(clause.name, clause.number, clause.vars, clause.lits,
            clause.negs, clause.select,
            PureEquation.symbolCounting(clause, weightfuncsymb, weightvarsymb), oldestage,
            clause.generatedby, clause.parents, clause.marked));
    oldestage++;
  }
  return combined;
}
```

Figure 4.2: Screenshot from the "combine" evaluation function, combining clauseweight and fifo

The last step before entering the main loop presented in algorithm 4.1 is the sorting of the set `unproc` of unprocessed clauses according to the score the evaluation function calculated for every clause (PureEquation). We take the variable `score` from the PureEquation. Clauses with a smaller score are ranked at the beginning the set `unproc`, so they get chosen faster. When the evaluation function `combine` is selected, `unproc` gets sorted according to the clause weight method (each time an old clause needs to be picked it is evaluated by comparing which clause this is and then pick it). For the sorting itself we use the merge sort algorithm.

If the user chose to use a SMT solver, the solver gets set up and started right before the main loop. To do this, we use the already implemented integration to SMT solvers from the RISCTP software [35] (used for the MESON prover), therefore the methods/functions itself have not been implemented within the scope of this thesis. For our thesis we use the connection to the SMT solver Z3 [10]. To initialize the solver we use the command `startSMT()` with the input ClauseProblem for the prover. With this, we create a `solver`, initialize it with the information form the input ClauseProblem and set the variable `rules` (type array) containing all clauses from the ClauseProblem but transformed to PureEquations.

4.4.5 Main Algorithm

The main loop of the algorithm was already presented in the form of pseudo code in algorithm 4.1. The implementation is pretty straight forward, so let us walk though the code in detail. In the very first step we try to solve the problem with a SMT solver, if the user set this option accordingly. The SMT solver gets called with the method `smtTheorem` and takes the initial ClauseProblem and a ProofProblem derived from it as input. We always send the whole problem to the SMT solver (tracked in the variable `rules`). The function returns true, the SMT solver was able to prove the unsatisfiability of the problem and the algorithm terminates. If the methods returns false we continue with the algorithm. We do not use any of the information of the SMT solver why the proof failed.

If no SMT solver shall be used the first step the given clause is selected from the set `unproc` of unprocessed clauses — denoted by `c`. The function `selectbest(unproc)` always selects the first element in the list `unproc`, since `unproc` is at any time in the algorithm sorted according to the evaluation function (fifo or clauseweight). Therefore we can also remove the head of the list to remove `c` from `unproc`. If we use the evaluation function `combine` the algorithm picks the oldest clause (at the frequence that is set) by comparing the `age` of the clauses. In this case also the oldest clause gets deleted from `unproc`, not the head. In the next step, the given clause `c` is simplified. The subroutine `simplify` simply calls all inference rules in the stated order "rewrite positive and negative literals", "negative simplify-reflect", "delete duplicated literals", "delete resolved literals" and "destructive equality resolution". If intra-clause and/or clause-clause simplification is excluded by the user settings, these rules get skipped.

Before we start digging into the implementation of clause simplification we take a closer look at the implementation of the lexiographic path ordering (LPO), since this is an important tool when testing requirements for the application of inference rules. We try to implement this ordering in a straight forward manner and therefore set up a method `lpo` for testing if $lit_1 > lit_2$, sticking to the definition given in this thesis. In the first step we track the length of both literals (if a literals is just a variable symbol we set the length to zero). The we try to prove, that lit_2 is a proper subterm of lit_1 by calling the relevant subroutine. If this was not successful, we try to show $f(t_1, \ldots, t_n) > t$, if $t_i > lit_2$ for some i by recursively calling the method `lpo` with all the arguments of lit_1 separately and return true if `lpo` returned true. The next step in the algorithm to check $f(t_1, \ldots, t_n) > f(u_1, \ldots, u_n)$, if $t_i > u_i$ for some i and $t_j = u_j$ for all $j < i$. So first we extract the first symbol of lit_1 and lit_2 and check if they are both the same. In a loop (to track all terms in the PureEquation with less literals) we additionally have to prove $f(t_1, \ldots, t_n) > u_i$ for every i so we call the method `lpo` recursively with lit_1 and u_i. Since all terms before position i have to be equal we again recursively call `lpo` with t_i and u_i if these two are not equal. If this criterion still was not enough to decide, which literal is greater, we test the last possible condition in the definition of the LPO. We try to show $f > g$ for some ordering of function/constant symbols to prove $f(t_1, \ldots, t_n) > g(u_1, \ldots, u_m)$. For the ordering the function symbols we chose the arity i. e. function symbols with higher arity are greater than others. If two symbols have the same arity, we chose randomly (but fix this random order throughout the whole algorithm).

```java
/*****************************************************************************
 * Simplifies a clause (pure equation) with respect to a set of clauses.
 * Applies the rules rewriting negative literals, delete duplicated literals,
 * delete resolved literals and destructive equality resolution
 * @param c is the pure equation to be simplified
 * @param processed set of pure equations used to simplify c
 * @param eq represents the equality function symbol
 * @param neq represents the inequality function symbol
 * @return simplified pure equation
 ****************************************************************************/
public PureEquation simplify(PureEquation c, List<PureEquation> processed,
        FunctionSymbol eq, FunctionSymbol neq)
{
  SimplifyingInferences simpl = new SimplifyingInferences(c, eq, neq,
          new Subst(), ordering);
  if (clauseClauseSimpl == true)
  {
    for (PureEquation clause : processed)
    {
      c = simpl.rewritingLiterals(clause, c);
      c = simpl.negativeSimplifyReflect(clause, c);
    }
  }
  if (intraClauseSimpl == true)
  {
    c = simpl.deleteDuplicatedLiterals(c);
    c = simpl.deleteResolvedLiterals(c);

    int pos = simpl.applicabilityDestrEqRes(c);
    if(pos != -1) c = simpl.destructiveEqualityResolution(c, pos);
  }
  return c;
}
```

Figure 4.3: Screenshot from the method `simplify`

After checking the arity (`Term.argnumber` declared as `len1` and `len2` we recursively call `lpo` for lit_1 and all positions in lit_2 and return true if this is true. If they have the same arity we track the position of both function symbols in the predefined random ordering (a list of function symbols, passed as input parameter). Smaller position means greater function symbol. If $f < g$ we again recursively call `lpo` for lit_1 and all positions in lit_2 and return true if this is true. In all other cases (non of the above described conditions were able to prove $lit_1 > lit_2$) the method returns false.

The method `simplify` holds a for-loop inside the block of clause-clause simplification inference rules running over all clauses that have already been processed. Inside the for-loop we call the implementations of the three inference rules “rewrite positive and negative literals”, “negative simplify-reflect”. Let us first take a detailed look at the rewriting rules. Since the rule is symmetric for positive and negative literals, we can handle both rules with the implementation of method. As input parameters we use two PureEquations, the first clause is the left clause in the theoretical inference rule and acts as an inference partner trying to simplify the second clause. Only the second parameter clause (given clause) gets changed (simplified). Two crucial and basic requirements for the application of this inference rule are that the first clause is only allowed to have exactly one literals and that this literal has to be an inequality. If this is not the case, we simply return the given clause without any changes. We then set up a list containing all literals from the given clause and track the left side s and the right side t of the equality literal from the clause passed first. With the help of a for-loop, we run through all literals in the given clause and try to find a substitution to match s to the left side u of the currently watched literal (the right hand side is denoted as v). To do this, we use the method `match` from the class Substitution. In comparison to the implementation to find a unifier the method `match` only tries to map variables from the literals in the given clause in a way to get the exact same term as in the inference partner. If a substitution is found, we still need to check the preconditions for the application of this inference rule. For checking $\sigma(s) > \sigma(t)$ we use the lexicographic path ordering (LPO), implemented as the method `lpo`. If all requirements are fulfilled, we create a new literal $\sigma(t) \neq v$, respectively $\sigma(t) = v$. Therefore we construct an array `args` of length two and set the first position to $\sigma(t)$, the second position to v (right side of the given clause). To generate a new literal we can the command `Term.term` from the class "Term" with the parameters (in)equality FunctionSymbol and the array `args`. We use the equality function symbol if the currently watched literal is an equality (rewriting of positive literals), analogously we use the inequality function symbol if the literal is an inequality (rewriting of negative literals). The corresponding literal gets overwritten in the list. Therefore we try to run through all positions in u and basically do the same procedure again for those positions. At the end we transform the list of literals to an array, construct a new PureEquation and return it. If we were not able to identify a substitution the inference rule allows to apply this rule to a position in u, not only to the whole literal. This is done by using the same procedure as described above, but the separately implemented method gets applied recursively to all subterms of u.

Starting very similar for the implementation of the inference rule "negative simplify-reflect". We have two PureEquations as input parameters. The first clause to be passed never gets changed, but is only there to help simplify the second clause (given clause) that is passed. Analogously to the inference rule described right above we need to test, if the first clause has exactly one literals and that that literal is an inequality. We simply return the given clause without any changes, if these two tests fail. In all other cases we create a list containing the literals of the given clause and a second list containing the polarities of the literals. In a for-loop over all literals of the given clause we try to find a substitution for this literals and the (first and only) literals from the inference partner. To achieve this, we again use the method "match" from the class "Substitution". Before attempting to find a substitution we make sure that the currently watched literals from the given clause is an equation (positive literal). If we found a substitution, we do nothing and continue with the next loop-traversal. If we were not able to find a substitution, we add this literals and its polarity to two new lists. The reason behind this is, that the literal that can be matched gets deleted from the given clause. Only adding literals that can't be matched brings us to the wanted result. Also if the currently watched literal was an inequality, the literal and its polarity also get added to the new lists. In the last step we transform the lists of literals and polarities to arrays, create a new PureEquation and return this simplified clause.

```java
/**************************************************************************
 * Rewriting of positive and negative literals: handling positive and
 * negative rewriting (symmetric rules) also checks the preconditions:
 * u|p = sigma(s) and sigma(s) > sigma(t)
 * @param clause1, the clause s = t (left hand side)
 * @param givenclause, the given clause u = v V R respectively u != v V R
 *        (right hand side)
 * @return PureEquation u[p <-- sigma(t)] = v V R
 *************************************************************************/
public PureEquation rewritingLiterals(PureEquation clause1, PureEquation givenclause)
{
  if (clause1.lits.length != 1) return givenclause;
  if (clause1.negs[0] == true) return givenclause;
  List<Object> lits = new ArrayList<>();
  for (Object l : givenclause.lits) lits.add(l);
  Object s = Term.argument(clause1.lits[0], 0);
  Object t = Term.argument(clause1.lits[0], 1);
  int i = 0;
  for (Object lit : lits)
  {
    Subst sigma = new Subst();
    Object u =  Term.argument(lit, 0);
    // trying to find a substitution such that u = sigma(s)
    boolean unify = Subst.match(sigma, s, u);
    if (unify == true)
    {
      // check if sigma(s) > sigma(t)
      if (PureEquation.lpo(sigma.apply(s), sigma.apply(t), ordering) == true)
      { // creating the new literal sigma(t) = v respectively sigma(t) != v
        Object[] args = new Object[2];
        args[0] = sigma.apply(t);
        args[1] = Term.argument(lit, 1);
        if (givenclause.negs[i] == true) lits.set(i, Term.term(neq, args));
        else lits.set(i, Term.term(eq, args));
      }
      i++;
      continue;
    } //end if unify
    i++;
  } //end for
  int litn = lits.size();
  Object[] lits0 = new Object[litn];
  for (int j = 0; j < litn; j++) lits0[j] = lits.get(j);
  return new PureEquation(givenclause.name, givenclause.number, givenclause.vars, lits0,
          givenclause.negs,
      givenclause.select, givenclause.score, givenclause.age, givenclause.generatedby,
      givenclause.parents, givenclause.marked);
}
```

Figure 4.4: Implementation of the inference rule rewriting of positive and negative

```java
/**********************************************************************
 * Negative simplify-reflect:
 * @param clause1, the clause s != t (left hand side)
 * @param givenclause, the given clause sigma(s = t) V R (right hand side)
 * @return PureEquation R
 **********************************************************************/
public PureEquation negativeSimplifyReflect(PureEquation clause1,
        PureEquation givenclause)
{
  if (clause1.lits.length != 1) return givenclause;
  if (clause1.negs[0] == false) return givenclause;
  List<Object> lits = new ArrayList<>();
  for (Object l : givenclause.lits) lits.add(l);
  List<Boolean> negs = new ArrayList<Boolean>();
  for (boolean neg : givenclause.negs) negs.add(neg);

  Object st = clause1.lits[0];
  List<Object> newlits = new ArrayList<>();
  List<Boolean> newnegs = new ArrayList<>();

  int i = 0;
  for (Object lit : lits)
  {
    Subst sigma = new Subst();
    if (givenclause.negs[i] == false)
    { // trying to find a substitution for the literal in clause 1 (s != t)
      // and a literal l in the given clause such that sigma(l) = s!=t
      boolean unify = Subst.match(sigma, st, lit);
      // only literals that cannot be matched with a substitution are added
      // to the return PureEquation
      if (unify == true)
      {
        i++;
        continue;
      }else
      {
        newlits.add(lit);
        newnegs.add(givenclause.negs[i]);
      }
    } else
    {
      newlits.add(lit);
      newnegs.add(givenclause.negs[i]);
    }
    i++;
  } //end for
  int litn = newlits.size();
  Object[] lits0 = new Object[litn];
  for (int j = 0; j < litn; j++) lits0[j] = newlits.get(j);
  boolean[] negs0 = new boolean[litn];
  for (int j = 0; j < litn; j++) negs0[j] = newnegs.get(j);

  return new PureEquation(givenclause.name, givenclause.number, givenclause.vars,
          lits0, negs0, givenclause.select,
          givenclause.score, givenclause.age, givenclause.generatedby,
          givenclause.parents, givenclause.marked);
}
```

Figure 4.5: Implementation of the inference rule rewriting of positive and negative

The second block is dedicated to intra-clause simplification. One rule is for "deleting duplicated literals". The implementation is quite straight forward. The literals in a clause are represented by an array, therefore we can simply add the literals one by one to a new array and thereby check, if this literals already exists in this array. For "deleting resolved literals" (left and right side of the equation are syntactically equal and the literal is negated) we also use straight forward testing. We loop through all literals in the clause and check if it is a negated literal, followed by checking if the left and right side is the same. If this condition is not fulfilled, the literal is not a resolved literal and is added to a new array for constructing the new simplified clause (PureEquation). Last we apply the inference rule "destructive equality resolution". With an auxiliary function we make sure the simplifying rule is applicable. If so, we return the position of the literal to be deleted, otherwise -1. We run through all literals $x \neq s$ in the given clause `c` and check if x is of the type `VariableSymbol`. Then we try to find a most general unifier for x and s. For getting one side of the equality $s = t$ translated to $= (s, t)$ we can use the method `Term.argument(lit, 0)` to get s, respectively `Term.argument(lit, 1)` to get t. If the preconditions are not fulfilled (return value -1) we are done with simplifying and move on. Otherwise we apply the inference rule. This rule starts by creating a new list of literals by adding all literals from the given clause except the one at the position return from the auxiliary function ($x \neq s$). We find the most general unifier for x and s and apply this unifier to all the other literals. Lastly the negations of the literals get initialised. A new PureEquation is constructed and returned:

```java
/*************************************************************************
 * Destructive equality resolution:
 * (This rule is only applied if the method applicabilityDestrEqRes returns
 * an integer greater 0)
 * @param clause, the clause to apply the rule to
 * @param pos, the position of the literal x != s
 * @return returns PureEquation without the literal  x != s, but applied the
 *         most general unifier of x and s to the other literals in clause
 ************************************************************************/
public PureEquation destructiveEqualityResolution(PureEquation clause, int pos)
{
  List<Object> lits = new ArrayList<Object>();
  Object x = clause.lits[pos];
  for (Object c : clause.lits) if (c != x) lits.add(c);

  Subst sigma = new Subst();
  boolean unify = Subst.unify(sigma, Term.argument(lits.get(pos), 0),
          Term.argument(lits.get(pos), 1));

  int litn = lits.size();
  Object[] lits0 = new Object[litn];
  for (int i = 0; i < litn; i++) lits0[i] = sigma.apply(lits.get(i));

  // initialize the literals and their polarities
  boolean[] negs = new boolean[litn];
  for (int i = 0; i < litn; i++)
  {
    if (lits0[i] instanceof Not) negs[i] = true;
    else negs[i] = false;
  }

  PureEquation rule = new PureEquation(clause.name, clause.vars, lits0, negs, clause.select,
          clause.score, clause.generatedby, clause.parents, clause.marked);
  return rule;
}
```

Figure 4.6: The implementation of destructive equality resolution

This simplifying process is followed by a check, if the given clause `c` is redundant. Redundant clauses are tossed completely at this point. This check is carried out with the inference rule `clause subsumption`. In the subroutine `redundant` we check for every clause in the set processed T, if it subsumes the given clause $R \vee S$, by applying the inference rule accordingly. The given clause always represents the right clause in the rule state in the chapter before, the left clauses are always clauses form the set of processed clauses. In the first step we construct two lists respectively with the literals of the two input clauses. If the given clause holds less literals than the clause from the set processed, we return false and move on to the next clause from "processed". If the size of the two clauses is exactly the same, we try to find a substitution σ such that $\sigma(T) = S$. Note that sigma is not a unifier, it is a variable matching. We use the method "match" from the class "Substitution", which tries to match the variable patterns from T to be equal to S. If this is possible, we return true, the clause can be subsumed and therefore is redundant. In all other cases we calculate all subsets of the given clause (of the same size as the clause T from the set of processed clauses) and try for every subset again to find a substitution (variable matching) such that $\sigma(T) = S$ as before, using the method `match`. If such a substitution can be found, the given clause can be subsumed and is redundant.

```
/****************************************************************************
 * Clause subsumption:
 * @param clause1, the clause (from the set processed) which might subsume
 *        the givenclause
 * @param givenclause, the clause (in our case the given clause) which might
 *        get subsumed
 * @return returns true, if clause1 subsumes givenclause
 ***************************************************************************/
public boolean clauseSubsumption(PureEquation clause1, PureEquation givenclause)
{
  Subst sigma = new Subst();

  List<Object> lits1 = new ArrayList<>();
  for (Object l : clause1.lits) lits1.add(l);
  List<Object> lits2 = new ArrayList<>();
  for (Object l : givenclause.lits) lits2.add(l);

  int size1 = lits1.size();
  int size2 = lits2.size();

  if (size2 < size1) return false;
  if (size1 == size2)
  {
    boolean matchable = Subst.match(sigma, lits1, lits2);
    if (matchable == false) return false;
    else return true;
  } else
  {
    List<List<Object>> allsubsets = subsets(givenclause, size1);
    for (List<Object> subset : allsubsets)
    {
      boolean matchable = Subst.match(sigma, lits1, subset);
      if (matchable == false) return false;
      else return true;
    }
  }
  return false;
}
```

Figure 4.7: The implementation of the inference rule clause subsumption

The first thing after is a check if `c` is the empty clause. If so we the algorithm terminates. The input set is unsatisfiable. This leads us to the conclusion, that the original input formula is valid (since we checked the negation for unsatisfiability). If not we add the given clause `c` to the set `processed` of processed clauses and enter the process of generating new clauses. Adding the given clause before generating new clauses might lead to a faster prove, since the clause might resolve with itself. A new empty list `t` is generated to collect the new clauses. Now we use the given clause `c` to simplify the set `processed`. We only do this if the user selected this option. To do this we first have to temporarily add `c` to the set of processed clauses and delete the clause we want to simplify. This set is called `help` in our code. Then the same procedure are described above is executed. After the simplification we check again if the clause has zero literals and therefore represents the empty clause. The algorithm would print a kind of proof tree and terminate at this point (unsatisfiable input set). The clause taken from the processed set gets replaced in the set `processed` by its simplified version.

Now we enter the subroutine `generate()` to generate new clauses. This method applies the "generating" inference rules equality resolution, equality factoring and superposition into positive and negative literals in this order. The newly generated clauses are first stored in separated list. This newly generated clauses have to have distinct variables to all other clauses, we ensure this by calling the method `newInstance` again. We use this right at the beginning of the algorithm and explain the implementation in the section 4.4.4 "Prerequisites". We combine all these newly generated clauses to one list and return it.

```java
/*****************************************************************************
 * Generate uses the inference rules equality resolution, equality factoring
 * as well as positive and negative superposition to compute all possible
 * new clauses between the givenclause and the already processed clauses
 * @param c is the givenclause
 * @param processed set of pure equations already processed
 * @param eq represents the equality function symbol
 * @param neq represents the inequality function symbol
 * @return a list of all newly generated pure equations
 ****************************************************************************/
public List<PureEquation> generate(PureEquation c, List<PureEquation> processed,
        FunctionSymbol eq, FunctionSymbol neq)
{
  GeneratingInferences gen = new GeneratingInferences(c, eq, neq, ordering);
  List<PureEquation> t = new ArrayList<>();
  List<PureEquation> resolution = new ArrayList<>();
  List<PureEquation> factoring = new ArrayList<>();
  List<PureEquation> superposition = new ArrayList<>();

  resolution.addAll(gen.computeAllEqResolvents(c));
  for (PureEquation clause : resolution) t.add(newInstance(clause));

  factoring.addAll(gen.equalityFactoring(c));
  for (PureEquation clause : factoring) t.add(newInstance(clause));

  superposition.addAll(gen.superposition(c, processed));
  for (PureEquation clause : superposition) t.add(newInstance(clause));

  return t;
}
```

Figure 4.8: Screenshot of the method generate

Before we look into the implementation of the actual generating inference rules, we should discuss a few methods needed and implemented to check preconditions for the application of the inference rules. We start of with a method to check for "eligibility for resolution". Since we did not implement a selection function only the first property $sel(C) = \emptyset$ and $\sigma(l)$ is $>$-maximal in $\sigma(C)$ is needed to show that $\sigma(l)$ is eligible for resolution. First we build a new list with the literals of the clause C and a list containing all selected literals (in our case always empty). We then check, is the list of selected literals is really empty and then call another method to check the maximality (for a given substitution σ). If both criteria return true, we also return true to indicate the eligibility for resolution. We also implemented a method for checking if $\sigma(l)$ is "eligible for paramodulation". We already check beforehand in our implementation, if l is a positive literals ($l \in C^+$) and that there are no literals selected. This leaves to prove, that $\sigma(l)$ is maximal in $\sigma(C)$ (using the same method as for "eligibility for resolution").

The method for checking the maximality takes a clause C (PureEquation), a substitution σ and a literal l as input. This function return true, if l is greater than every literal in the clause C. In the first step we apply the substitution σ to the literals l. If the input clause has only one literal, we return true, since this literals is then the greatest literals of this clause. For clauses with more literals, we create a list with all literals from the clause and apply the substitution σ to it. In a loop over all literals c in C (σ applied) we check for all literals unequal to l if l is greater than c according to te LPO (implementation already explained at the beginning). If so, the method `maximality` returns true.

For the implementation of the generating inference rule "equality resolution" we split the algorithm into two subproblems. The first method is `computeAllEqResolvents` is used to compute all possible equality resolvents in the clause $s \neq t \vee R$ (the given clause) and also checks if there exists a unifier for s and t and the eligibility for resolution for $s \neq t$. The second method `computeEqResolvent` the is called to really compute the equality resolvent (deleting the resolvent literal $s \neq t$). This method is implemented pretty straight forward. As input parameters this function takes a clause (the given clause), a substitution and a position (of the literal to be resolved). In a first step we create a new list containing all literals accept the one at the input position. Then we create an array with these literals and apply the input substitution to them. The last step is to initialize the polarities of the literals (`negs`). A new `PureEquation` is constructed and returned. Therefore the parents variable of the PureEquation is filled with the input clause and the variable `generatedb"` is set to "ER". This method will be called by computeAllEqResolvents if all preconditions are fulfilled. First we set up an empty list `t` of PureEquations to collect all newly generated clauses and create a list with all the literals of the input clause (the given clause in our case). We loop over all literals. For inequalities we try to find a unifier for the left and right side of the inequality. To do this we use the method `unify` from the class Substitution, which was written by Wolfgang Schreiner. If we were able to identify a unifier we use the method described above to test for eligibility fo resolution. The last step is to call `computeEqResolution` and add the result to `t`. After all literals were checked we return `t` (and therefore all newly generated clauses).

```
/****************************************************************************
 * Computes all possible equality resolvent of a clause (the givenclause).
 * Every literal gets checks if it fulfills the preconditions.
 * @param givenclause - the PureEquation
 * @return list of all newly generated equality resolvents
 ****************************************************************************/
public List<PureEquation> computeAllEqResolvents(PureEquation givenclause)
{
  List<PureEquation> t =  new ArrayList<>();
  List<Object> lits =  new ArrayList<>();
  for (Object c : givenclause.lits) lits.add(c);

  for(int i = 0; i < lits.size(); i++)
  {
    if (givenclause.negs[i] == true)
    {
      Subst sigma = new Subst();
      boolean unify = Subst.unify(sigma, Term.argument(lits.get(i), 0),
              Term.argument(lits.get(i), 1));
      if (unify == true)
      {
        Object sigmal = sigma.apply(lits.get(i));
        boolean test = eligibleForResolution(givenclause, sigma, sigmal);
        if (test == true) t.add(computeEqResolvent(givenclause, sigma, i));
      }
    }
  }
  return t;
}
```

Figure 4.9: Implementation of the generating inference rule equality resolution for a clause

```
/****************************************************************************
 * Computes the equality resolvent of a clause given a unifier and a position.
 * The preconditions for applying this rule (as well as the unifier and the position)
 * have to be checked/computed in advance.
 * @param clause - the PureEquation s != t v R
 * @param subst - the unifier sigma = mgu(s, t)
 * @param pos - position of the literal s != t used to compute the equality resolvent
 * @return the newly generated PureEquation generated by equality resolution sigma(R)
 ****************************************************************************/
public PureEquation computeEqResolvent(PureEquation clause, Subst subst, int pos)
{
  List<Object> lits = new ArrayList<Object>();
  for (Object c : clause.lits) if (c != clause.lits[pos]) lits.add(c);

  int litn = lits.size();
  Object[] lits0 = new Object[litn];
  for (int i = 0; i < litn; i++) lits0[i] = (Object[]) subst.apply(lits.get(i));

  boolean[] negs = new boolean[litn];
  for (int i = 0; i < litn; i++)
  {
    if (lits0[i] instanceof Not) negs[i] = true;
    else negs[i] = false;
  }

  List<PureEquation> parents = new ArrayList<>();
  parents.add(clause);

  PureEquation rule = new PureEquation(clause.name, clause.vars, lits0, negs,
          clause.selectionFunction(lits0, negs), clause.score, "ER", parents, clause.marked);
  return rule;
}
```

Figure 4.10: Implementation of the application of the inference rule equality resolution

Similar to the implementation of the inference rule “equality resolution” we have one method `equalityFactoring` that organizes and tests all preconditions and a second method that really computes the new clause (application of the inference rule), called `computeEqualityFactor`. The implementation of the method `equalityFactoring` also start with creating a new empty list of PureEquations `t` to collect all newly generated clauses and set up a list with all literals of the input clause (in our case the given clause). One precondition for applying this inference rule is the criterion of eligibility for paramodulation. This allows the application only, if no literals are selected. Therefore we first check, if the variable `selected` (type list) from the input PureEquation is empty. Then we start two nested for loops to check all possible pairs of literals to find $s = t$ and $u = v$ such that s and u have a most general unifier σ, $\sigma(s) \geq \sigma(t)$ and $\sigma(s = t)$ is eligible for paramodulation. For every literal we have to check that is represents an equality (not an inequality). Then we try to find a unifier for the left side of the equality of each of the two literals we look at at the moment (nested for-loop). If there exists one, we apply the unifier to s and t and use the lexicographic path ordering (as described above) to check if $\sigma(s) \geq \sigma(t)$. Additionally we try to prove that $\sigma(s = t)$ is eligible for paramodulation using the corresponding method already described at the beginning of this section. If both test turn our to be true, we call the method "computeEqualityFactor" to actually compute the new clause. As input parameters we use the original input clause, the unifier σ, and the positions p_1, p_2 of the two literals (in the list of literals). In the first step this method sets up a list with all the literals of the clause and an array with the polarities of each literal. Next we must create a new literal (in our program represented by terms) to generate $t \neq v$. To do this we extract respectively the left hand sides of the two literals at p_1 and p_2 with the command "Term.argument" and store them in an array of size 2. We then construct the new term with the command "Term.term" and the inequality function symbol and the array and overwrite the literals at p_1 with this newly constructed literal. In the following steps we update the polarity of the literal at p_1 in the corresponding array. We set the parents variable of the PureEquation and return a new PureEquation with the updated literals and polarities (`negs`). Back to the original method `computeEqualityFactor`. The return `PureEquation` gets added to the list of newly generated clauses and the nested for-loop starts the next iteration. After all literals have been tested with each other we return the list of newly generated clauses (PureEquation) by equality factoring. This list might stay empty, if no pair of literals fulfils the preconditions.

```
/*************************************************************************
 * Computes the equality factor of a clause given a unifier and a position.
 * The preconditions for applying this rule (+ the unifier and the positions)
 * have to be checked/computed in advance.
 * @param clause - the PureEquation s = t V u = v V R
 * @param subst - the unifier sigma = mgu(s, u)
 * @param pos1 - position of the literal s = t used to compute the equality factor
 * @param pos2 - position of the literal u = v used to compute the equality factor
 * @return the newly generated PureEquation generated by equality factor
 *         sigma(t != u V u = v V R)
 *************************************************************************/
public PureEquation computeEqualityFactor(PureEquation clause, Subst sigma,
        int pos1, int pos2)
{
  List<Object> lits = new ArrayList<>();
  for (Object l : clause.lits) lits.add(l);
  int litn = lits.size();

  boolean[] negs = new boolean[litn];
  for (int i = 0; i < litn; i++) negs[i] = clause.negs[i];

  Object[] args = new Object[2];
  args[0] = Term.argument(lits.get(pos1), 1);
  args[1] = Term.argument(lits.get(pos2), 1);

  lits.set(pos1, Term.term(neq, args));
  negs[pos1] = true;

  Object[] lits0 = new Object[litn];
  for (int i = 0; i < litn; i++) lits0[i] = sigma.apply(lits.get(i));

  List<PureEquation> parents = new ArrayList<>();
  parents.add(clause);

  PureEquation returnclause = new PureEquation(clause.name,
          PureEquation.clauseCounter, clause.vars, lits0, negs, clause.select,
          clause.score, clause.age, "EF", parents, clause.marked);
  PureEquation.clauseCounter = PureEquation.clauseCounter + 1;
  return returnclause;
}
```

Figure 4.11: Implementation of the generating inference rule equality factoring for a clause

```
/*********************************************************************
 * Computes all possible equality factory of a clause (the givenclause).
 * Every literal gets checks if it fulfills the preconditions.
 * @param givenclause - the PureEquation
 * @return list of all newly generated equality factors
 *********************************************************************/
public List<PureEquation> equalityFactoring(PureEquation givenclause)
{
  List<PureEquation> t =  new ArrayList<>();
  List<Object> lits = new ArrayList<Object>();
  for (Object c : givenclause.lits) lits.add(c);
  boolean[] negs = givenclause.negs;
  List<Integer> selected = givenclause.select;
  if (selected.isEmpty() == true)
  {
    for (int i = 0; i < lits.size()-1; i ++)
    {
      if(negs[i] == false)
      {
        for (int j = i+1; j < lits.size(); j ++)
        {
          if (negs[j] == false)
          {
            Subst sigma = new Subst();
            boolean unify = Subst.unify(sigma, Term.argument(lits.get(i), 0),
                    Term.argument(lits.get(j), 0));
            if (unify == true)
            {
              boolean test1 = false;
              Object lit = lits.get(i);
              Object s0 = sigma.apply(Term.argument(lit, 0));
              Object t0 = sigma.apply(Term.argument(lit, 1));
              if (PureEquation.lpo(t0, s0, ordering) == false) test1 = true;
              boolean test2 = eligibleForParamodulation(givenclause, sigma,
                      lits.get(i));
              if(test1 == true && test2 == true)
                  t.add(computeEqualityFactor(givenclause, sigma, i, j));
            }
          }
        }
      }
    }
  }
  return t;
}
```

Figure 4.12: Implementation of the application of the inference rule equality factoring

The last generating inference rules to be applied are “superposition” into positive and negative literals. Since these two rules are symmetric, they can be handled at once with one implemented method. In comparison to the other two rules superposition requires two clauses and generates one new clause from these. To ensure this, we implemented a function `superposition` with the set of processed clauses `processed` and the given clause as input.

```java
/*****************************************************************************
 * Computes all possible new clauses between the givenclause and the processed clauses
 * using superposition
 * Computes superposition (if possible) using the given clause as as the left hand
 * clause and then using the given clause as the right hand clause
 * runs trough all processed clauses
 * @param givenclause
 * @return list of all newly generated clauses (with superposition)
 ****************************************************************************/
public List<PureEquation> superposition(PureEquation givenclause,
        List<PureEquation> processed)
{
  List<PureEquation> t =  new ArrayList<>();
  List<PureEquation> givenclauselist =  new ArrayList<>();
  givenclauselist.add(givenclause);
  t.addAll(superpositionIntoPositiveLiterals(givenclause, processed));

  List<Object> lits = new ArrayList<Object>();
  for (Object c : givenclause.lits) lits.add(c);
  int i = 0;
  int j = 0;

  for (PureEquation clause : processed)
  {
    i = 0;
    t.addAll(superpositionIntoPositiveLiterals(clause, givenclauselist));
    for (Object lit : lits)
    {
        Function<Object,Object> overwrite = (Object t0)->t0;
        t.addAll(superpositionIntoPositiveLiteralsInnerTerms(givenclause, clause,
                lit, givenclause.lits[i], i, overwrite));
        i++;
    }
  }

  for (PureEquation clause : processed)
  {
    j = 0;
    List<Object> lits2 = new ArrayList<Object>();
    for (Object c : clause.lits) lits2.add(c);
    for (Object lit : lits2)
    {
        Function<Object,Object> overwrite = (Object t0)->t0;
        t.addAll(superpositionIntoPositiveLiteralsInnerTerms(clause, givenclause,
                lit, clause.lits[j], j, overwrite));
        j++;
    }
  }

  return t;
}
```

Figure 4.13: Screenshot of the method superposition to make sure all possible superpositions into positive and negative literals are computed

Analogously, we first set up an empty list of PureEquations `t` to collect all newly generated clauses. The goal is, to (try to) apply the superposition inference rule to all combinations of clauses. To achieve this, we implemented a method `superpositionIntoPositiveLiterals` which takes a clause and a list of clause and test the preconditions for every combination (similar to the other generating inference rules). In this implementation, the clause acts as *main premise* (i. e. the right clause in the theoretical inference rule stated in the previous section) and the clauses in processed as inference partner (left clause in the theoretical inference rule). To really check out all combination we have to make sure also the other way around is tested (given clause as inference partner and the clauses from "processed" as main premise). To match the input data types we have to create a list only containing the given clause, so we can call `superpositionIntoPositiveLiterals` again. We start with creating an empty list of PureEquations to collect all newly generated clauses (this list will later be returned) and a list containing all literals of the given clause. With a for loop we test the preconditions for every literal with all other possible inference partners from `processed`. We track the left and right side of the (in)equation of the currently watched literals with the command `Term.argument` (denoted as u and v) and the track if it is an equality or inequality, to know which rules we might apply. The second for-loop now runs through the set `processed`. With a third, nested for-loop we now run through all literals in this clause (from "processed"). Since the "left clause" always has to be an equality, this is the first test we perform. The term u is not allowed to be a variable symbol. This is tested by the command `instance of VariableSymbol`. In the next step we try to find a most general unifier between u and the left hand side of currently watched literal $s = t$ (from a clause in "processed"). If we found one, we use the implementation of the lexicographic path ordering to check $\sigma(u) \geq \sigma(v)$ ad well as $\sigma(s) \geq \sigma(t)$. We are left to check, if $\sigma(s = t)$ is eligible for paramodulation (we use the already described method for this) and if the corresponding literals from the given clause is eligible for resolution. If both of criteria are fulfilled, we call a third method called `computeSuperpositionPositiveLiteral`, which then really computes the new clause according to the inference rule. The inference rule is also applicable, if the preconditions are not only satisfied by the left side of a literal of the given clause (called u), but also if the preconditions are fulfilled at a position in u. If we were not able to prove the required criteria for the u we access a for-loop to check the preconditions in the same way as described above for all positions in u (and for every literal in the given clause, therefore this code is inside the first for-loop). This is done with a recursive call of the method called `computeSuperpositionPositiveLiteralInnerTerms`.

The method `computeSuperpositionPositiveLiteral` takes several input parameters: a PureEquation from the set `processed` and the position `pos1` of the literal fulfilling the preconditions, the given clause and the respective position `pos2` of the literal (in the list of literals), the unifier σ and a string variable indicating if we need to apply superposition into positive or negative literals. Additionally we also pass an integer p, denoting the position inside u (second case that is described in the paragraph above). If we can fulfil the precondition already with u, we pass -1. First we set up a list of literals in the given clause and a list of literals from the other clause, but without the literal at `pos1`. Next we initialise an array containing the polarities of both, the given clause and the inference partner (again without the polarity of the literal at `pos1`). This is later needed when creating the new clause, since the new PureEquation is a combination of both clauses. After these initialisations we construct the literal $u[p \leftarrow t] = v$, respectively $u[p \leftarrow t] \neq v$. Therefore extract the right side of the literals (equation) of the inference partner at `pos1`, denoted as t, by the command `Term.argument` and create an array of size two called `args` to generate the left and right side of the new literal. The next step is to set the first position in `args` to `t` (if the literals u was able to fulfil all preconditions), if a position in u was eligible to apply superposition we have to change the term at this position in u and use the updated literal u to set the first position in `args`. The second position in `args` gets set to v — the right side of the literals in the given clause at position `pos2`. Now we can construct the new literal by using the command `Term.term` with the (in)equality function symbol and the array `args`. This new term (literal) overwrites the literals in the given clause at `pos2`. We use the inequality symbol for superposition into negative literals and the equality symbol when applying superposition into positive literals. Additionally we set the variable `generatedby` of the PureEquation to `superposition into positive literals` or respectively to "superposition into negative literals". In the last steps we apply the unifer σ to all literals and set the `parents` variable and `vars` (now containing all variables of both input clauses) needed to generate a `PureEquation`. A new PureEquation is generated and returned.

```
/*******************************************************************************
 * Computes the new clause according to the superposition inference rule of two
 * clauses given two positions, a unifier and the note if it is superposition into
 * positive or of negative literals.
 * The preconditions for applying this rule (as well as the unifier and the positions)
 * have to be checked/computed in advance.
 * @param clause1 - PureEquation s = t V S
 * @param pos1 - position of the literal that superposition can be applied to
 * @param givenclause - the given clause u = v V R respectively u != v V R
 * @param pos2 - position of the literal that superposition can be applied to
 * @param sigma - the unifier sigma = mgu(u|p, s)
 * @param p - position of the term in u superposition can be applied to
 * @param posneg - string, indicating to use positive or negative superposition
 * @param newlit - newly constructed literal if the rule gets applied to an inner term of u
 * @return the newly generated PureEquation generated by superposition sigma(u[p <-- t] != v V S V R)
 ******************************************************************************/
public PureEquation computeSuperpositionPositiveLiteral(PureEquation clause1, int pos1, PureEquation givenclause,
                                                        int pos2, Subst sigma, int p, String posneg, Object newlit)
{
  List<Object> lits = new ArrayList<Object>();
  for (Object c : givenclause.lits) lits.add(c);
  Object remove = clause1.lits[pos1];
  for (Object c : clause1.lits) if (c != remove) lits.add(c);
  int n = givenclause.negs.length;
  boolean[] negs = new boolean[n + clause1.negs.length - 1];
  for (int i = 0; i < n; i++) negs[i] = givenclause.negs[i];
  int j = 0;
  for (int i = 0; i < clause1.negs.length; i++)
  {
    if (i != pos1)
    {
      negs[n+j] = clause1.negs[i];
      j++;
    }
  }
  Object t = Term.argument(remove, 1);
  int litn = lits.size();
  Object[] args = new Object[2];
  if (p == -1) args[0] = t;
  else
  {
    args[0] = Term.argument(newlit, 0);
  }
  String generatedby = "";
  args[1] = Term.argument(lits.get(pos2), 1);
  if (posneg == "positiv")
  {
    lits.set(pos2, Term.term(eq, args));
    generatedby = "SP";
  }
  else if (posneg == "negativ")
  {
    lits.set(pos2, Term.term(neq, args));
    generatedby = "SN";
  }
  Object[] lits0 = new Object[litn];
  for (int i = 0; i < litn; i++) lits0[i] = sigma.apply(lits.get(i));

  int len1 = givenclause.vars.length;
  int len2 = clause1.vars.length;
  VariableSymbol[] vars0 = new VariableSymbol[len1 + len2];

  for (int i = 0; i < len1; i++) vars0[i] = givenclause.vars[i];
  for (int i = 0; i < len2; i++) vars0[i+len1] = clause1.vars[i];

  List<PureEquation> parents = new ArrayList<>();
  parents.add(clause1);
  parents.add(givenclause);

  PureEquation returnclause = new PureEquation(givenclause.name, PureEquation.clauseCounter, vars0, lits0, negs,
        givenclause.select, givenclause.score, givenclause.age, generatedby, parents, givenclause.marked);
  PureEquation.clauseCounter = PureEquation.clauseCounter + 1;
  return returnclause;
}
```

Figure 4.14: Implementation of the generating inference rule superposition into positive and negative literals for a clause

```
..
/*****************************************************************************
 * Computes all possible superpositions of the given clause and the processed clauses
 * Every clause (and literal) gets checks if it fulfills the preconditions.
 * @param givenclause - the given clause u = v V R respectively u != v V R
 * @param processed - list of processed PureEquations (possible inference partners)
 * @return list of all newly generated clauses (with superposition)
 ****************************************************************************/
//given clause is the right-hand clause in the inference rule
public List<PureEquation> superpositionIntoPositiveLiterals(PureEquation givenclause,
        List<PureEquation> processed)
{
  List<PureEquation> t =  new ArrayList<>();
  String posneg;
  List<Object> lits = new ArrayList<Object>();
  for (Object c : givenclause.lits) lits.add(c);
  int j = 0;
  int i = 0;
  for (Object l1 : lits)
  {
    Object u = Term.argument(l1, 0);
    Object v = Term.argument(l1, 1);
    if (givenclause.negs[i] == false) posneg = "positiv";
    else posneg = "negativ";
    for (PureEquation clause : processed)
    { //with every clause in processed we try to generate a new one
      j = 0;
      for (Object l2 : clause.lits)
      {
        if (clause.negs[j] == false)
        { //in left clause has to be an equality
          if ((u instanceof VariableSymbol) == false)
          { // u at position p is not allowed to be a variable
            Subst sigma = new Subst();
            boolean unify = Subst.unify(sigma, u, Term.argument(l2, 0));
            if (unify == true)
            {
              if (PureEquation.lpo(Term.argument(l2, 1), Term.argument(l2, 0), ordering)
                      == false)
              {
                if (PureEquation.lpo(v, u, ordering) == false)
                {
                  if (eligibleForParamodulation(clause, sigma, l2) == true)
                  {
                    if (eligibleForResolution(givenclause, sigma, l1) == true)
                    {
                      t.add(computeSuperpositionPositiveLiteral(clause, j, givenclause, i,
                              sigma, -1, posneg, ""));
                    }
                  }
                }
              }
            }
            j++;
            continue;
          }
        }
        j++;
      }
    }
    i++;
  }
  return t;
}
```

Figure 4.15: Partial Implementation of the application of the inference rule superposition into positive and negative literals

After this whole process of generating new clauses the given clause `c` gets added to `processed`. Now we loop over all newly generated clause (for loop through the elements in `t`). We again use intra-clause simplification inference rules to simplify the newly generated clauses using the subroutine `cheapSimplify`. If the user chose to use intra-clause simplification that the two rules "delete duplicated literals" and "delete resolved literals" are applied. The implementation of these rules is already explained within the scope of the subroutine `simplify`. Afterwards we again check, if the newly generated and now simplified clause is the empty clause, by checking, if the array `lits` does not contain any elements. If so the program terminates. The input set is unsatisfiable, meaning that the original input first-order logic formula is valid (since we tried to prove the unsatisfiablity of the set of clauses derived from the negation of the formula).

Before adding the newly generated clauses to the set `unproc` we check, if those clauses are trivial. We only add non trivial clauses. We use the two inference rules syntactic tautology deletion 1 and 2, called by the subroutine `trivial`. At least one of these two rules has to return true. The user has again the chance to skip this check by setting the relevant variable (`intraClauseSimpl` = false). Let us observe the implementation of the two syntactic tautology deletion rules closer. We simply check for every literal in the clause, if the left and the right side of the equation are syntactically the same. Since every literal is a term in our program (so is every side of a literal) we can use the method `equal` from the class Term:

```
/***************************************************************************
 * Syntactic tautology deletion 1:
 * @param clause, the clause to checked
 * @return returns true, if the clause contains an equation where the left
 *         and right side are syntactically the same
 **************************************************************************/
public boolean syntacticTautologyDeletion1(PureEquation clause)
{
  List<Object> lits = new ArrayList<>();
  for (Object c : clause.lits) lits.add(c);
  boolean[] negs = clause.negs;
  for( int i = 0; i < lits.size(); i++)
  {
    if(Term.equal(Term.argument(lits.get(i), 0), Term.argument(lits.get(i), 1))
            && (negs[i] == false)) return true;
  }
  return false;
}
```

Figure 4.16: The implementation of syntactic tautology deletion (rule 1)

In a nested for loop we check all combinations of two literals (equations) which have different negations. Then the left-hand sides of the two literals are checks for syntactic equality, followed by the same check for the right side:

```
/***************************************************************************
 * Syntactic tautology deletion 2:
 * @param clause, the clause to checked
 * @return returns true, if the clause contains two syntactically equal
 *         literals with opposite negations
 ***************************************************************************/
public boolean syntacticTautologyDeletion2(PureEquation clause)
{
  List<Object> lits = new ArrayList<>();
  for (Object c : clause.lits) lits.add(c);
  boolean[] negs = clause.negs;
  int size = lits.size();

  for(int i = 0; i < size; i++)
  {
    for(int j = i + 1; j < size; j++)
    {
      if (negs[i] != negs[j])
      {
        if(Term.equal(Term.argument(lits.get(i), 0), Term.argument(lits.get(j), 0)))
        {
          if(Term.equal(Term.argument(lits.get(i), 1), Term.argument(lits.get(j), 1)))
          {
            return true;
          }
        }
      }
    }
  }
  return false;
}
```

Figure 4.17: The implementation of syntactic tautology deletion (rule 2)

Now that we know the clause is not trivial we have to add it to the set `unproc`. Since `unproc` is a sorted list according to the evaluation function we use the method *insertAtRightPosition* to insert the new clause `p` at the correct position in `unproc` and keep this list sorted. If the list `unproc` is empty at this point we simply add the new clause `p`. In all other cases we call the method `evaluationFunction` with the newly generated clause (which then calls either the fifo, the clauseweight evaluation function or a combination of these two). The result is, that we get a score for the clause, which we can use to compare it with the other clauses is `unproc`. If the set `unproc` currently contains only one element, we compare the `score` of this element and the newly generated clause and insert `p` accordingly at the beginning or the end. For longer lists `unproc` we take the "middle" element from `unproc` (clause at the position size divided by two rounded up) and compare its `score` with the one from `p`. If the `score` (evaluation) from `p` is smaller, we start comparing at the beginning of `unproc` and insert `p` at the correct position. If the `score` from `p` is greater, we start comparing at the list element size divided by two rounded up (second half of the list only). Again, `p` gets inserted at the correct position.

The last step in this loop traversal is to add all newly generated clauses that got added to the set `unproc` of unprocessed clauses also get added to the list `rule` to "inform" also the SMT solver about the new clauses. Next time the SMT solver gets called those new clauses are also considered.

The process of the described while loop starts again, if the list `unproc` is not empty. If the list is empty, the algorithm ends.

4.4.6 Output and Interpretation

As an output of the prover, we try to picture the proof and how the algorithm got there. We start by printing the initial set of clauses (showing the renaming of the variables) and enumerate them. The numbers are used to indicate which clauses were used to simplify which clause or to produce a new one. We also give a notification, which category of simplification was used (intra-clause or clause-clause). Similar for the generating inferences, we print a tag at the beginning of the output-line to inform which rule was used to generate this clause (ER for equality resolution, EF for equality factoring, SP for superposition into positive literals and SN for superposition into negative literals).

If the prover found the empty clause, it is stated so and we print the runtime. The output also contains the number of clauses processed until the proof was found and the inference depth. The last line always prints "SUCCESS" for indicating that a proof for unsatisfiablity was found. This needs to be interpreted in the following way: The initial input to the program is a first-order logic formula. This formula gets negated and translated to a set of clauses. The resolution prover takes this set of clauses and tried to prove the unsatisfiablility. If our prover comes to the result "SUCCESS" this means a proof for the unsatisfiability for the set of clauses was found. Since the set of clauses was derived from the negation of the input first-order formula, it follows that the input formula is valid. The second option is, that the prover ends with the last line printing "FAILURE". This means that no proof of unsatisfiability was found. In general this means that we do not know anything about the original first-order logic input formula. In a special case, the prover is indeed able to construct a model (set `processed` of processed clauses) and therefore really prove the satisfiability of the set of clauses (derived from the negation of the input formula). In this case the first-order logic formula would be invalid. This happens if only finitely many resolution steps are possible. In all other cases our prover ends with "FAILURE" it is due to timing out (this is marked in the output).

In this case we do not know anything about the original first-order logic formula. In the next chapter we show some examples with more detailed explanation to the output.

Chapter 5

Evaluation of the Prover

In this chapter we present some test runs performed with our prover. We give an overview over the runtime with different option and give a brief comparison to the MESON prover. Additionally we walk through a selection of example to give a detailed insight on how the prover solves problems.

5.1 Example Problems

In this first section we walk through a couple of examples to show the detailed work the prover performs and explain the program output. We show examples with different results.

5.1.1 barber.txt

With this first example we want to show the output of the program while simultaneously showing how the proof would be build by hand. This way we want to make the reader aware on how to read the output and be able to better understand how the algorithm works. Let us take a close look at the example *barber.txt* which can be downloaded from the RISCAL website (together with the RISCAL software or from the repository) [34]. It is taken from the textbook of Harrison "Handbook of Practical Logic and Theorem Proving" page 180 [13]. This example illustrates a variant of Russell's paradox stating, that there does not exists a barber, that shaves exactly those people, who do not shave themselves [13]. As a logic formula, this encodes to

$$\neg(\exists b{:}T.(\forall x{:}T.(shaves(b,x) \Leftrightarrow (\neg shaves(x,x)))))$$

So for our program we take the negation, i. e. $\exists b{:}T.(\forall x{:}T.(shaves(b,x) \Leftrightarrow (\neg shaves(x,x))))$ and try to show the unsatisfiability for the negation. This transforms to the initial set of clauses (proof problem) which is given as input to our algorithm:

1. $\forall x{:}T.\top \Rightarrow (shaves(b_o,x) \vee shaves(x,x))$
2. $\forall x{:}T.(shaves(x,x) \wedge shaves(b_o,x)) \Rightarrow \bot$

In the program output, this is printed as follows:

```
RISC Theorem Proving Interface 1.8.4 (February 25, 2025)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
-----------------------------------------------------------------
Reading file C:\Users\Viktoria\Documents\Studium\Master\RISCTP\problems\barber.txt...
=== axioms and theorems:
theorem T ⇔ ¬(∃b:T. (∀x:T. (shaves(b,x) ⇔ (¬shaves(x,x)))));
===
=== clause form of problem
type T;
pred shaves(x:T,y:T);
const b§:T;

----------------------------------------
1:[T.0] ∀x:T. (shaves(b§,x) ∧ shaves(x,x)) ⇒ ⊥
2:[T.1] ∀x:T. ⊤ ⇒ (shaves(b§,x) ∨ shaves(x,x))
```

Figure 5.1: Screenshot from the program output to show how the input problem gets transformed for the example barber.txt

For using our prover we fix some setting before we start: We use the method of symbol counting to select the given clause, we use no SMT solver for this test run, but we add integer axioms. For the simplification we do everything that is implemented, using all simplifying inference rules (intra-clause and clause-clause) for the subroutines `simplify`, `cheap_simplify`, `redundant`, `trivial` and we additionally simplify the set `proc`. We transform these to negation normal form (already done before entering our algorithm), transform each literal to an equality, respectively to an inequality, and rename the variable (add an @ symbol and a counter to each variable) such that all variables in all clauses are different (two equal variables in one clause of course have to be renamed to the same new variable):

1. $= (shaves(b0, x@0), true) \lor = (shaves(x@0, x@0), true)$
2. $\neq (shaves(x@1, x@1), true) \lor \neq (shaves(b0, x@1), true)$

Those two clauses build the set `unproc` at the beginning getting added in this order, since they have the same number of symbols we arrange the first clause first. The set `proc` is empty at the beginning. Clause (1) is picked as the given clause in the first loop traversal. This clause cannot be simplified and it is also not redundant. Using equality factoring we are able to produce a new clause:

3. $\neq (true, true) \lor = (shaves(b0, b0), true)$

Applying the inference rule "deletion of resolved literals", the prover is able to simplify this clause to $= (shaves(b0, b0), true)$. Since this clause (3) is not trivial it gets added to `unproc` at the first place in the list (smaller than clause (2)), clause (1) gets added to `proc`. In the second iteration the algorithm picks clause (3) as the given clause. The given clause could not be simplified and is also not redundant. No simplification of the set `proc` is possible. With the inference rule superposition into positive literals" the prover manages to generate four new clauses:

4. $= (true, true) \lor = (shaves(b0, b0), true)$ generated by (1) and (3)

5. $= (true, true) \vee = (shaves(b0, b0), true)$ generated by (1) and (3)
6. $= (true, true) \vee = (shaves(b0, b0), true)$ generated by (3) and (1)
7. $= (shaves(b0, b0), true) \vee = (true, true)$ generated by (3) and (1)

We do get for clauses since we apply to rule to every possible constellation of clauses (and literals). In detail, the application of the inference rule

$$\frac{\text{clause (1)} \qquad \text{clause (3)}}{\text{clause (4)}}\ \text{(SP)}$$

produces two clauses, since this rule can be applied two time (to each literals in clause (1)). Additionally, (SP) also gets called with clause (1) and (2) respectively in the other position creating the other two clauses. These clauses cannot be simplified with the implemented rules, but according to "syntactic tautology deletion 1" all on them are trivial. The literal $= (true, true)$ is a tautology (always true) leading in a conjunction to the result, that the whole clause simplifies to true. Therefore not added to the list `unproc` (containing only clause (2) now). Clause (3) is added to the set `proc` (containing now the clauses (1) and (3)). Everything that is explained in the text above is printed by the algorithm in the following way:

```
The inital set of clauses at the start of the prover:
1: =(shaves(b0,x@0), true) ∨ =(shaves(x@0,x@0), true)
2: ≠(shaves(x@1,x@1), true) ∨ ≠(shaves(b0,x@1), true)
Iteration: 1
EF:1 generated 3: ≠(true, true) ∨ =(shaves(b0,b0), true)
Intra-Clause (cheap): 3 simplified itself to =(shaves(b0,b0), true)
Iteration: 2
SP: 1 and 3 generated 4: =(true, true) ∨ =(shaves(b0,b0), true)
SP: 1 and 3 generated 5: =(true, true) ∨ =(shaves(b0,b0), true)
SP: 3 and 1 generated 6: =(true, true) ∨ =(shaves(b0,b0), true)
SP: 3 and 1 generated 7: =(shaves(b0,b0), true) ∨ =(true, true)
4 is trivial
5 is trivial
6 is trivial
7 is trivial
```

Figure 5.2: Screenshot from the program to show the output for the first and second loop traversal for the example barber.txt

In the third loop traversal clause (2) gets selected as given clause. It is not possible to simplify (2) and it is also not redundant. Again it is not possible with the implemented rules to further simplify `proc`. Now the inference rule "superposition into negative literals" is used to generate five new clauses:

8. $\neq (true, true) \vee \neq (shaves(b0, b0), true) \vee = (shaves(b0, b0), true)$ generated by (1) and (2)
9. $\neq (true, true) \vee \neq (shaves(b0, x@14), true) \vee (shaves(b0, x@14), true)$ generated by (1) and (2)
10. $\neq (true, true) \vee \neq (shaves(b0, b0), true)$ generated by (3) and (2)
11. $\neq (shaves(b0, b0), true) \vee \neq (true, true) \vee = (shaves(b0, b0), true)$ generated by (3) and (2)

12. $\neq(shaves(b0, b0), true) \vee \neq(true, true)$ generated by (1) and (2)

The algorithm uses the inference rule "deletion of resolved literals" in the subroutine `cheap_simplify` to delete the literal $\neq(true, true)$ from the newly generated clauses (8), (9), (10), (11) and (12). Additionally, clause (8), (9) and (11) are trivial according to the rule "syntactic tautology deletion 1", since those clauses contain the exact same literals with the opposite negation. Clause (10) and (12) are added to `unproc` in this order

12. $\neq(shaves(b0, b0), true)$

10. $\neq(shaves(b0, b0), true)$

and clause (2) is added to the set `proc`:

1. $=(shaves(b0, x@0), true) \vee =(shaves(x@0, x@0), true)$

3. $=(shaves(b0, b0), true)$

2. $\neq(shaves(x@1, x@1), true) \vee \neq(shaves(b0, x@1), true)$

Clause (12) gets selected to be the given clause. Using clause-clause simplification with clause (3) as an inference partner, the prover can simplify (12) to $\neq(true), true)$ applying the rule "rewriting of negative literals". For clause (3) and (12) there exits a variable matching such that the both left-hand sides of the clauses ($shaves(b0, b0)$ in bot clauses already) are equal. Therefore we can replace $shaves(b0, b0)$ in clause (12) by the right hand side of clause (3) ($true$), resulting in $\neq(true, true)$. Applying "deletion of resolved literals" finally generates the empty clause and the algorithm terminates with "SUCCESS". Success means, that a proof for unsatisfiablility could be found for the negation of the original logical statement, i. e. $\exists b{:}T.(\forall x{:}T.(shaves(b, x) \Leftrightarrow (\neg shaves(x, x))))$. It therefore follows, that the logical formula $\neg(\exists b{:}T.(\forall x{:}T.(shaves(b, x) \Leftrightarrow (\neg shaves(x, x)))))$ is satisfiable.

```
Iteration: 3
SN: 1 and 2 generated 8: ≠(true, true) ∨ ≠(shaves(b0,b0), true) ∨ =(shaves(b0,b0), true)
SN: 1 and 2 generated 9: ≠(true, true) ∨ ≠(shaves(b0,x@14), true) ∨ =(shaves(b0,x@14), true)
SN: 3 and 2 generated 10: ≠(true, true) ∨ ≠(shaves(b0,b0), true)
SN: 1 and 2 generated 11: ≠(shaves(b0,b0), true) ∨ ≠(true, true) ∨ =(shaves(b0,b0), true)
SN: 3 and 2 generated 12: ≠(shaves(b0,b0), true) ∨ ≠(true, true)
Intra-Clause (cheap): 8 simplified itself to ≠(shaves(b0,b0), true) ∨ =(shaves(b0,b0), true)
8 is trivial
Intra-Clause (cheap): 9 simplified itself to ≠(shaves(b0,x@14), true) ∨ =(shaves(b0,x@14), true)
9 is trivial
Intra-Clause (cheap): 10 simplified itself to ≠(shaves(b0,b0), true)
Intra-Clause (cheap): 11 simplified itself to ≠(shaves(b0,b0), true) ∨ =(shaves(b0,b0), true)
11 is trivial
Iteration: 4
Clause-Clause: 3 simplified 12 to ≠(true, true)
Intra-Clause: 12 simplified itself to {}
Found the empty clause 1 in 68 ms
Number of clauses processed until a proof for unsatisfiability was found: 31
Inference depth: 4
SUCCESS termination (130 ms).
```

Figure 5.3: Screenshot from the program to show the final output for the example barber.txt

5.1.2 pelletier20.txt

For this next example we want to show one using the method `combine` to select the given clause. We use the old clause at a frequency of 3. The settings for simplification remain unchanged, so we still use as much simplification os possible. Also we add integer axioms, but do not use a SMT solver. Let us take a closer look at the example *pelletier20.txt*, which can be found in the textbook of Harrison "Handbook of Practical Logic and Theorem Proving" page 253 [13]. It can also be downloaded from the RISCAL website (together with the RISCAL software or from the repository) [34]. The example represents the logical formula

$$\forall x{:}T, y{:}T.\exists z{:}T.\forall w{:}T.p(x)\wedge q(y) \Rightarrow r(z)\wedge u(w)) \Rightarrow ((\exists x{:}T, y{:}T.p(x)\wedge q(y)) \Rightarrow (\exists z{:}T.r(z))$$

As explained before, the prover is working with the negation of this formula and derives a set of clauses from this negation:

```
RISC Theorem Proving Interface 1.8.4 (February 25, 2025)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
---------------------------------------------------------------
Reading file C:\Users\Viktoria\Documents\Studium\Master\RISCTP\problems\pelletier20.txt...
=== axioms and theorems:
theorem T ⇔ (∀x:T,y:T. (∃z:T. (∀w:T. ((p(x) ∧ q(y)) ⇒ (r(z) ∧ u(w)))))) ⇒ ((∃x:T,y:T. (p(x) ∧ q(y))) ⇒ (∃z:T. r(z)));
===
=== clause form of problem
type T;
pred p(x:T);
pred q(x:T);
pred r(x:T);
pred u(x:T);
fun z§(x_0:T,x_1:T):T;
const x§:T;
const y§:T;

---------------------------------------
1:[T.0] ∀x0:T,y0:T. (p(x0) ∧ q(y0)) ⇒ r(z§(x0,y0))
2:[T.1] ∀x0:T,y0:T,w0:T. (p(x0) ∧ q(y0)) ⇒ u(w0)
3:[T.2] ⊤ ⇒ p(x§)
4:[T.3] ⊤ ⇒ q(y§)
5:[T.4] ∀z1:T. r(z1) ⇒ ⊥
```

Figure 5.4: Screenshot from the program output to show how the input problem gets transformed for the example pelletier20.txt

The algorithm is able to construct a prove for unsatisfiability within seven iterations and processing 61 clauses. Let us take a look at the program output and analyse it:

Right at the beginning we can see, that the initial set of clauses is sorted according to the clause wight function (symbol counting with weight 1 for function and variable symbols). In the first iteration, therefore clause (5) is picked as the given clause, but it is not possible to do any simplifications or generate new clauses. For the second loop traversal the algorithm again selects the given clause according to the clause weight method (small clause) and picks clause (3). Two trivial clauses are generated in this iteration. Now in iteration three `selectbest()` picks the oldest clause as the given clause — clause (1). Using superposition into negative literals we are able to generate two new clauses (8) and (9) and simplify them using intra-clause simplification. In the iterations four and five we again select the smallest clauses as given clauses and are able to generate a new, non trivial clause (12) in iteration 5 using the clauses (3) and (9). For the sixth iteration the algorithm again has to pick the oldest clause (fairness), currently clause (2), but

```
The inital set of clauses at the start of the prover:
5: ≠(r(z1@5), true)
3: =(p(x), true)
4: =(q(y), true)
2: ≠(p(x0@2), true) ∨ ≠(q(y0@3), true) ∨ =(u(w0@4), true)
1: ≠(p(x0@0), true) ∨ ≠(q(y0@1), true) ∨ =(r(z(x0@0,y0@1)), true)
Iteration: 1
Iteration: 2
SP: 3 and 3 generated 6: =(true, true)
SP: 3 and 3 generated 7: =(true, true)
6 is trivial
7 is trivial
Iteration: 3
SN: 3 and 1 generated 8: ≠(true, true) ∨ ≠(q(y0@7), true) ∨ =(r(z(x,y0@7)), true)
SN: 1 and 5 generated 9: ≠(true, true) ∨ ≠(p(x0@9), true) ∨ ≠(q(y0@10), true)
Intra-Clause (cheap): 8 simplified itself to ≠(q(y0@7), true) ∨ =(r(z(x,y0@7)), true)
Intra-Clause (cheap): 9 simplified itself to ≠(p(x0@9), true) ∨ ≠(q(y0@10), true)
Iteration: 4
SP: 4 and 4 generated 10: =(true, true)
SP: 4 and 4 generated 11: =(true, true)
10 is trivial
11 is trivial
Iteration: 5
SN: 3 and 9 generated 12: ≠(true, true) ∨ ≠(q(y0@13), true)
Intra-Clause (cheap): 12 simplified itself to ≠(q(y0@13), true)
Iteration: 6
2 got subsumed
Iteration: 7
SN: 4 and 12 generated 13: ≠(true, true)
Intra-Clause (cheap): 13 simplified itself to {}
Found the empty clause 3 in 51 ms
Number of clauses processed until a proof for unsatisfiability was found: 61
Inference depth: 7
SUCCESS termination (129 ms).
```

Figure 5.5: Screenshot from the program output for the example pelletier20.txt

this clause gets subsumed. In iteration 7 the algorithm can finally generate the empty clause ba superposition into negative literals (and intra-clause simplification) from the clause (4) and (12). The last line "SUCCESS" means, that the set of clauses shown in the first screenshot has been proven to be unsatisfiable. Since this set was derived from the negation of the original first-order logic input formula it follows, that the input formula is valid.

5.1.3 eq.txt

In this example, we present two proof problems where equality reasoning plays an important role. The first problem represents a valid theorem for which the prover is indeed able to show the unsatisfiability of its negation. The second problem represents an (intentionally) invalid theorem for which the prover shows the satisfiability of the negation by the construction of a satisfying model. This is represented in the example *eq.txt* which can also be downloaded from the RISCAL website (together with the RISCAL software or from the repository) [34]. The example is not from a textbook, but was constructed by Wolfgang Schreiner. This example illustrates two theorems:

$$\text{Axiom 1: } \forall x{:}T.p(x) \Rightarrow f(f(x)) = g(x)$$

$$\text{Theorem 1: } \forall c{:}T.p(c) \Rightarrow f(f(f(c))) = f(g(c))$$

As it can be easily seen that this theorem is valid. We declare this as proof problem 1.

$$\text{Axiom 2:} \,\forall x{:}T.i(x) = h(x)$$

$$\text{Theorem 2:} \,\forall c{:}T.j(c) = h(c)$$

Theorem 2 here represents an invalid formula. The formula was made invalid on purpose. We call this proof problem 2 in the following.

As for all examples, the negation of theorems gets transformed to a set of clauses (and the axioms get added). The prover tries to prove the unsatisfiablility of this set of clauses. For this example we derive two different proof goals. The prover tries to construct a proof for each theorem separately and starts again automatically after each proof attempt. As for the user settings, we use the method `combined` to select the given clause and this time we only use intra-clause simplification (no simplifying of the set of processed clauses and so clause-clause inference rules). The algorithm start, by processing the input as depicted in Fig. 5.6.:

```
RISC Theorem Proving Interface 1.8.4 (February 25, 2025)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
-----------------------------------------------------------------
Reading file C:\Users\Viktoria\Documents\Studium\Master\RISCTP\problems\eq.txt...
=== axioms and theorems:
axiom A ⇔ ∀x:T. (p(x) ⇒ '=§1'(f(f(x)),g(x)));
theorem T ⇔ ∀c:T. (p(c) ⇒ '=§1'(f(f(f(c))),f(g(c))));
axiom B ⇔ ∀x:T. '=§1'(i(x),h(x));
theorem U ⇔ ∀c:T. '=§1'(j(c),h(c));
===
=== clause form of problem
type T;
pred '=§1'(x:T,y:T);
pred p(x:T);
fun f(x:T):T;
fun g(x:T):T;
const c§:T;
fun h(x:T):T;
fun i(x:T):T;
fun j(x:T):T;

1:[A] ∀x:T. p(x) ⇒ '=§1'(f(f(x)),g(x))
2:[B] ∀x:T. ⊤ ⇒ '=§1'(i(x),h(x))
```

Figure 5.6: Screenshot from the program output to show how the input gets transformed for the example eq.txt

For the first proof goal the proof is quite easy. From Axiom 1 follows $p(c) \Rightarrow f(f(c)) = g(c)$. Replacing the inner subterm $f(f(c))$ in Theorem 1 by $g(c)$ using the axiom we get $f(g(c)) = f(g(c))$. This simplifies to "true" and the formula is valid. The prover takes a few more steps to generate the proof, but is also able to show the unsatisfiability:

```
---------------------------------------
1:[T.0] ⊤ ⇒ p(c§)
2:[T.1] '=§1'(f(f(f(c§))),f(g(c§))) ⇒ ⊥
The inital set of clauses at the start of the prover:
2: =(p(c), true)
4: =(i(x@1), h(x@1))
3: ≠(f(f(f(c))), f(g(c)))
1: ≠(p(x@0), true) ∨ =(f(f(x@0)), g(x@0))
Iteration: 1
SP: 2 and 2 generated 5: =(true, true)
SP: 2 and 2 generated 6: =(true, true)
SP: 2 and 2 generated 7: =(true, true)
SP: 2 and 2 generated 8: =(true, true)
5 is trivial
6 is trivial
7 is trivial
8 is trivial
Iteration: 2
SP: 4 and 4 generated 9: =(h(x@3), h(x@3))
SP: 4 and 4 generated 10: =(h(x@5), h(x@5))
SP: 4 and 4 generated 11: =(h(x@7), h(x@7))
SP: 4 and 4 generated 12: =(h(x@9), h(x@9))
9 is trivial
10 is trivial
11 is trivial
12 is trivial
Iteration: 3
SN: 2 and 1 generated 13: ≠(true, true) ∨ =(f(f(c)), g(c))
SN: 2 and 1 generated 14: ≠(true, true) ∨ =(f(f(c)), g(c))
Intra-Clause (cheap): 13 simplified itself to =(f(f(c)), g(c))
Intra-Clause (cheap): 14 simplified itself to =(f(f(c)), g(c))
Iteration: 4
SP: 1 and 13 generated 15: =(g(c), g(c)) ∨ ≠(p(c), true)
SP: 13 and 13 generated 16: =(g(c), g(c))
SP: 1 and 13 generated 17: =(g(c), g(c)) ∨ ≠(p(c), true)
SP: 13 and 13 generated 18: =(g(c), g(c))
SP: 13 and 13 generated 19: =(g(c), g(c))
SP: 13 and 13 generated 20: =(g(c), g(c))
15 is trivial
16 is trivial
17 is trivial
18 is trivial
19 is trivial
20 is trivial
```

Figure 5.7: Screenshot from the program output for the first part (iteration 1 to 4) of the example eq.txt

```
Iteration: 5
SP: 1 and 14 generated 21: =(g(c), g(c)) ∨ ≠(p(c), true)
SP: 13 and 14 generated 22: =(g(c), g(c))
SP: 14 and 14 generated 23: =(g(c), g(c))
SP: 1 and 14 generated 24: =(g(c), g(c)) ∨ ≠(p(c), true)
SP: 14 and 13 generated 25: =(g(c), g(c))
SP: 13 and 14 generated 26: =(g(c), g(c))
SP: 14 and 14 generated 27: =(g(c), g(c))
SP: 14 and 14 generated 28: =(g(c), g(c))
SP: 14 and 13 generated 29: =(g(c), g(c))
SP: 14 and 14 generated 30: =(g(c), g(c))
21 is trivial
22 is trivial
23 is trivial
24 is trivial
25 is trivial
26 is trivial
27 is trivial
28 is trivial
29 is trivial
30 is trivial
Iteration: 6
SN: 1 and 3 generated 31: ≠(g(f(c)), f(g(c))) ∨ ≠(p(f(c)), true)
SN: 1 and 3 generated 32: ≠(g(f(c)), f(g(c))) ∨ ≠(p(f(c)), true)
SN: 1 and 3 generated 33: ≠(f(g(c)), f(g(c))) ∨ ≠(p(c), true)
SN: 13 and 3 generated 34: ≠(f(g(c)), f(g(c)))
SN: 14 and 3 generated 35: ≠(f(g(c)), f(g(c)))
Intra-Clause (cheap): 33 simplified itself to ≠(p(c), true)
Intra-Clause (cheap): 34 simplified itself to {}
Found the empty clause in 126 ms
Number of clauses processed until a proof for unsatisfiability was found: 61
Inference depth: 6
```

Figure 5.8: Screenshot from the program output for the first part (iteration 5 and 6) of the example eq.txt

Let us extract the parts and clauses that are actually needed and leading to the successful proof. Using the clauses (2) $= (p(c), true)$ and (1) $\neq (p(x@o), true) \vee = (f(f(x@o)), g(x@o))$ and applying the inference rules "superposition into negative literals" and "deletion of resolved literals" we can generate the new clause (13) $= (f(f(c)), g(c))$. Again, by applying "superposition into negative literals" to the clauses (13) and (3) $\neq (f(f(f(c))), f(g(c)))$ the algorithm can create the new clause (34) $\neq (f(g(c)), f(g(c))$. Clause (34) then simplifies to the empty clause and the algorithm terminates.

After the algorithm successfully found a proof for the proof goal 1 it automatically starts with proof goal 2. Again the program derives a set of clauses from the negation of Theorem 2. This time the prover is not able to show the unsatisfiability of this set. The program output looks like this:

```
=== clause form of problem
type T;
pred '=§1'(x:T,y:T);
pred p(x:T);
fun f(x:T):T;
fun g(x:T):T;
fun h(x:T):T;
fun i(x:T):T;
fun j(x:T):T;
const c§:T;

1:[A] ∀x:T. p(x) ⇒ '=§1'(f(f(x)),g(x))
2:[B] ∀x:T. ⊤ ⇒ '=§1'(i(x),h(x))
----------------------------------------
1:[U] '=§1'(j(c§),h(c§)) ⇒ ⊥
The inital set of clauses at the start of the prover:
37: =(i(x@1), h(x@1))
38: ≠(j(c), h(c))
36: ≠(p(x@0), true) ∨ =(f(f(x@0)), g(x@0))
Iteration: 1
SP: 37 and 37 generated 39: =(h(x@3), h(x@3))
SP: 37 and 37 generated 40: =(h(x@5), h(x@5))
SP: 37 and 37 generated 41: =(h(x@7), h(x@7))
SP: 37 and 37 generated 42: =(h(x@9), h(x@9))
39 is trivial
40 is trivial
41 is trivial
42 is trivial
Iteration: 2
Iteration: 3
Input set is satisfiable
FAILURE termination (238 ms).
```

Figure 5.9: Screenshot from the program output for the second part of the example eq.txt

The program is starting the proof with the set of clauses (unprocessed clauses):

36. $\neq (p(x@o), true) \vee = (f(f(x@o)), g(x@o))$

37. $= (i(x@1), h(x@1))$

38. $\neq (j(c), h(c))$

The first given clause picked by the algorithm is $= (i(x@1), h(x@1))$. By applying the rule "superposition into positive literals" to the clause and itself the prover can generate four new clause $= (h(x), h(x))$ (where x is a different variable for every clause). There clauses are all trivial and therefore not added to the list of unprocessed clauses. In the next two iterations the clauses 38, respectively 36 are selected as the given clause, but it is not possible to generate any new clauses. Then the set of unprocessed clauses gets empty and the algorithm terminates with "FAILURE" after only three iterations. Since in this example it is only possible to perform finitely many steps, this means that the set of clause is satisfiable. The set of processed clauses represents a model. In this case we can conclude, that the input formulae (Theorem 2) is invalid.

Remark. Be careful when concluding, that the input formula is invalid when the prover shows "FAILURE" as a result. First-order logic is in general not decidable. In most cases we therefore do not know anything about the input formula (if we are not able to show unsatisfiability). It is possible (and most likely), that the prover ran out of time and therefore could not fine a prove for unsatisfiability. Also the user setting (using plain symbol counting to select the given clause) might lead to this situation, since the algorithm is no longer complete. Only in exceptional cases it is possible to show, that the set of clauses is satisfiable (input formula invalid). This is the case if only finitely many resolution steps were possible. The set of processed clauses `processed` represents a model.

5.1.4 lists.txt

This example, called lists.txt, is used to show the functionality of the SMT solver in our prover. For the simplifying we again use the settings in a way to achieve maximal possible simplification (clause-clause and intra-clause simplification and simplification of the set of processed clauses). We do not add integer axioms and use the SMT solver at the beginning of each main-loop traversal. The example encodes, how to compute the sum of a list containing integers. As logical statement this can be represented by:

$$\mathrm{IntList} = \mathrm{empty} \mid \mathrm{cons}(head : \mathrm{Int}, tail : \mathrm{IntList})$$

$$\forall l : \mathrm{IntList}.\ \mathrm{sum}(l) = \mathrm{match}\ l\ \mathrm{with}\ \begin{cases} \mathrm{empty} \to 0 \\ \mathrm{cons}(h : \mathrm{Int}, t : \mathrm{IntList}) \to h + \mathrm{sum}(t) \end{cases}$$

$$\mathrm{let}\ l = \mathrm{cons}(1, \mathrm{cons}(2, \mathrm{empty}))\ \mathrm{in}\ \mathrm{sum}(l) = 3$$

As for all examples, the negation of this logical formula (input) gets transformed to a set of clauses. The prover tries to prove the unsatisfiability of this set of clauses. The derived set of clauses looks like this in the program output:

```
<terminated> Main_fol [Java Application] C:\Program Files\java\jdk-17\bin\javaw.exe  (16.11.2025, 14:52:19 – 14:52:21) [pid: 7252]
RISC Theorem Proving Interface 1.8.4 (February 25, 2025)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
-----------------------------------------------------------------
Reading file C:\Users\Viktoria\Documents\Studium\Master\RISCTP\problems\lists.txt...
=== axioms and theorems:
axiom def§20 ⇔ ∀value:Int. ('Nat::type'(value) ⇔ (0 ≤ value));
theorem typecheck(Nat)§0 ⇔ ∃value:Int. 'Nat::type'(value);
axiom height§type ⇔ ∀x:IntList. 'Nat::type'(height(x));
axiom sumax ⇔ ∀l:IntList. (('is::empty'(l){*} ⇒ '=§0'(sum(l),0)) ∧ ((¬'is::empty'(l){*}) ⇒ '=§0'(sum(l),head(l)+sum(tail(l)))));
theorem T ⇔ '=§0'(sum(cons(1,cons(1+1,empty))),(1+1)+1);
===
=== clause form of problem
type Int;
pred '=§0'(x:Int,y:Int);
pred ≤(x1:Int,x2:Int);
type Nat = Int;
const 0:Int;
pred 'Nat::type'(value:Int);
fun +(x1:Int,x2:Int):Int;
datatype IntList = empty | cons(head:Int,tail:IntList);
fun height(x:IntList):Nat;
const empty:IntList;
pred 'is::empty'(§x:IntList);
fun cons(head:Int,tail:IntList):IntList;
fun head(x:IntList):Int;
fun tail(x:IntList):IntList;
fun sum(l:IntList):Int;
const 1:Int;

1:[def§20.0] ∀value:Int. 0 ≤ value ⇒ 'Nat::type'(value)
2:[def§20.1] ∀value:Int. 'Nat::type'(value) ⇒ 0 ≤ value
3:[height§type] ∀x:IntList. ⊤ ⇒ 'Nat::type'(height(x))
4:[sumax.0] ∀l:IntList. 'is::empty'(l){*} ⇒ '=§0'(sum(l),0)
5:[sumax.1] ∀l:IntList. ⊤ ⇒ ('is::empty'(l){*} ∨ '=§0'(sum(l),head(l)+sum(tail(l))))
```

Figure 5.10: Screenshot from the program output to show how the input problem gets transformed for the example lists.txt

As the output shows, for this example it is necessary to prove two different theorems. The first problem the prover tries to show is a type checking exercise.

$$\exists value\colon \mathrm{Int}.nattype(value)$$

Using the SMT solver, this can be proven in the first iteration:

```
-----------------------------------------
1:[typecheck(Nat)§0] ∀value:Int. 'Nat::type'(value) ⇒ ⊥
The inital set of clauses at the start of the prover:
3: ≠('Nat::type'(value@2), true)
4: =('Nat::type'(height(x@3)), true)
1: =('Nat::type'(value@0), true) ∨ ≠(≤(0,value@0), true)
2: =(≤(0,value@1), true) ∨ ≠('Nat::type'(value@1), true)
5: ≠('is::empty'(l@4), true) ∨ =('=0'(sum(l@4),0), true)
6: =('is::empty'(l@5), true) ∨ =('=0'(sum(l@5),+(head(l@5),sum(tail(l@5)))), true)
We apply the SMT solver to the clauses...
Theorem was proved from the clauses.
We apply the SMT solver to ground instances of the clauses...
Theorem was NOT proved from the instances.
true
The SMT solver was able to show unsatisfiability.
```

Figure 5.11: Screenshot from the program output showing the first part of the prove for the example lists.txt

The algorithm automatically starts again for the second theorem. As we already learned, we negate the input logic formula and try to prove the unsatisfiability of this negation. For this example we therefore try to show

$$\neg(\mathrm{sum}(\mathrm{cons}(1, \mathrm{cons}(1+1, \mathrm{empty}()))) \neq (1+1)+1)$$

This prove for unsatisfiability is also done immediately by the SMT solver in the first iteration:

```
=== clause form of problem
type Int;
pred '=§0'(x:Int,y:Int);
pred ≤(x1:Int,x2:Int);
type Nat = Int;
const 0:Int;
pred 'Nat::type'(value:Int);
fun +(x1:Int,x2:Int):Int;
datatype IntList = empty | cons(head:Int,tail:IntList);
fun height(x:IntList):Nat;
const empty:IntList;
pred 'is::empty'(§x:IntList);
fun cons(head:Int,tail:IntList):IntList;
fun head(x:IntList):Int;
fun tail(x:IntList):IntList;
fun sum(l:IntList):Int;
const 1:Int;

1:[def§20.0] ∀value:Int. 0 ≤ value ⇒ 'Nat::type'(value)
2:[def§20.1] ∀value:Int. 'Nat::type'(value) ⇒ 0 ≤ value
3:[height§type] ∀x:IntList. ⊤ ⇒ 'Nat::type'(height(x))
4:[sumax.0] ∀l:IntList. 'is::empty'(l){*} ⇒ '=§0'(sum(l),0)
5:[sumax.1] ∀l:IntList. ⊤ ⇒ ('is::empty'(l){*} ∨ '=§0'(sum(l),head(l)+sum(tail(l))))
----------------------------------------
1:[T] '=§0'(sum(cons(1,cons(1+1,empty))),(1+1)+1) ⇒ ⊥
The inital set of clauses at the start of the prover:
15: =('Nat::type'(height(x@2)), true)
13: =('Nat::type'(value@0), true) ∨ ≠(≤(0,value@0), true)
14: =(≤(0,value@1), true) ∨ ≠('Nat::type'(value@1), true)
16: ≠('is::empty'(l@3), true) ∨ =('=0'(sum(l@3),0), true)
17: =('is::empty'(l@4), true) ∨ =('=0'(sum(l@4),+(head(l@4),sum(tail(l@4)))), true)
18: ≠('=0'(sum(cons(1,cons(+(1,1),empty))),+(+(1,1),1)), true)
We apply the SMT solver to the clauses...
Theorem was proved from the clauses.
We apply the SMT solver to ground instances of the clauses...
Theorem was proved from the instances.
true
The SMT solver was able to show unsatisfiability.
SUCCESS termination (325 ms).
```

Figure 5.12: Screenshot from the program output showing the second part of the prove for the example lists.txt

Since the negation of both theorems could be proven to be unsatisfiable, we can conclude, that the original first-order logic input formula is valid.

5.2 Benchmarks

In this section we want to give some timings for a couple of examples and try different options we implemented in our prover. We test different simplification settings and switching the use of an SMT solver on and off. We use the pre-implemented examples which can be downloaded from the home page of the RISC Algorithm Language (RISCAL) [34] or automatically come with the download of the RISCAL software. Additionally, we briefly compare the runtime of our prover to the MESON prover available in RISCTP [35].

We start by showing and trying to briefly analyse the runtime for different settings for our implemented prover. We implemented a couple of option for the user to chose from. While testing we tried to cover many combinations and list the most interesting combinations in the following tables. First we need to decide which method to use for the selection of the given clause (combined, clause weight or first-in-first-out). Additionally, we set different parameters for the simplification. We decided on providing the runtime for three different simplification options:

1. simplify everything: Here we use all simplifying inference rules (intra-clause and clause-clause) that we implemented simplifying the given clause, using the given clause to simplify the set of processed clauses, checking for redundancy and triviality and simplifying newly generated clauses.
2. no simplification of processed clauses: Same as before, but without the simplification of the set of processed clauses.
3. intra-clause simplification: We do not simplify the set of processed clauses and for all the other subroutines (`simplify`, `cheap_simplify`, `redundant`, `trivial`) we only use intra-clause simplification inference rules.

Last we offer the option to add integer axioms to allow the prover to solve problems with integers, but also to use a SMT solver for outsourcing these problems.

5.2.1 Comparing Different User Settings for the Resolution Prover

In Table 5.1, we list the runtime for a set of examples comparing different user settings. The table only contains examples, where the prover was able to show the validity of the input first-order logic formula. This means, that the algorithm was able to construct a proof to show the unsatisfiability of the set of first-order clauses, derived from the negation of the input formula. We compare the following setting:

- We test three different option for simplifying clauses (simplify everything, no simplification of processed clauses, only intra-clause simplification). These option are already explained in the introduction to this chapter “Benchmarks”.
- For each simplification option we test the prover with three different function to select the given clause (Comb, CW, fifo). The shortcut “Comb” defines the combination of the first-in-first-out method and the clause weight function, “CW” denotes the clause wight method and “fifo” represents the first-in-first-out selection of the given clause.
- For every setting we do not use an external SMT solver, but we add integer respectively array axioms to the input problem

Some rows contain the entry “x”. This indicated, that the prover terminated (without finding a prove) due to a set time limit.

Runtime in ms for examples with proof of validity									
Example	Integer axioms, no SMT								
	Simplify everything			No simplification of processed clauses			Only intra-clause simplification		
	Comb	Cw	fifo	Comb	Cw	fifo	Comb	Cw	fifo
arith1.txt	8465	x	x	25065	x	x	x	x	x
arith4.txt	14124	x	x	7834	x	x	x	x	x
barber.txt	67	63	50	68	162	104	141	69	67
case.txt	65	46	63	56	115	85	58	56	76
davisPutnam.txt	50	35	41	50	43	64	55	58	68
drinker.txt	137	139	77	88	65	121	67	x	75
ewd1062.txt	1366	x	x	702	x	x	x	x	x
expand.txt	56	47	52	59	101	121	45	57	52
fol.txt	190	166	589	159	220	756	297	x	x
fol0.txt	44	29	35	32	35	43	34	38	29
fol3.txt	56	46	45	59	58	49	52	44	36
fol4.txt	82	80	106	87	75	80	79	84	93
group2.txt	4086	x	x	3158	x	x	x	x	x
leq.txt	127	x	x	115	x	x	x	x	x
los.txt	22786	x	x	21898	x	x	x	x	x
nat.txt	132	55	61	49	75	76	63	68	66
nat2.txt	149	2087	187	116	2094	295	259	1115	295
numbers.txt	234	138	x	138	95	x	x	x	x
numbers2.txt	0	0	0	0	1	1	0	0	0
pelletier20.txt	50	64	64	41	53	58	51	60	55
pelletier24.txt	67	58	x	66	59	67	72	70	81
pelletier32.txt	91	81	x	79	76	95	93	96	188
pelletier45.txt	266	210	x	306	145	4288	x	x	x
perm.txt	142	148	x	160	123	231	124	2441	241
sat.txt	x	x	x	1644	x	x	x	x	x
sort.txt	76496	x	x	72879	x	x	x	x	x
steamroller.txt	3638	4208	x	3131	16281	x	x	x	x

Table 5.1: Runtime comparison for different settings of the resolution prover in ms

As the table shows for the most examples the clause weight selection function with as much simplification as possible brings the best results. On the other hand, the fifo selection function is not much slower for most of the examples. Only for nat2.txt, pelletier45.txt and steamroller.txt the runtime differed perceptible. But it is important to point out, that both fifo and clause weight are not able to find a proof for all examples were it is possible to show unsatisfiability (for example for arith1.txt). This is due to the fact that these to functions for selecting the given clause are not fair, meaning, that an old clause might never get picked (but would be needed to find the proof). Therefore using the `combine` method can be rated as the best method and is therefore also used for further comparisons. For the second option — not simplifying the set of processed clauses — the algorithm seems to vary runtime-wise being a bit a bit faster for some examples, but also being slower for some examples. For the clause selection method `combine` this setting is actually faster and also able to find proofs for more examples. An interesting observation might be that the difference between the fifo function and symbol counting does not correlate between option one and two. When only using intra-clause simplification, the algorithm gets drastically slower. We also tried running the prover without any simplification, but noticed quickly that this does not work fine. Even for very simple example it suddenly took several minutes to complete a proof. Therefore we conclude that this setting is not practical nor useful for the purpose of finding a proof and to not give any runtime for this scenario.

5.2.2 Runtime Comparison to the MESON Prover

Table 5.2 contains a selection of examples, showing a comparison to the MESON prover. We show a comparison of the runtimes between the MESON prover and the resolution prover (our prover). Again the table only contains examples, where the MESON prover was able to show the validity of the input first-order logic formula. Naturally, it is possible to see which examples worked with the MESON prover and also worked with the resolution prover. If the MESON prover found a proof and the resolution prover did as well, it is marked with the symbol "+" in the column "Success". If the resolution prover could not generate a proof, but the MESON prover could, we mark this with the symbol "-". We used the following setting to try to ensure a high level of comparability:

- For the MESON prover we used the default settings.
- For our own prover we used the two different settings, which had the best results (Table 5.1).
  1. As selection function for the given clause we chose the method `combined` (fifo and clause weight combined) and "simplify everything". Additionally we add integer, respectively array, axioms.
  2. Again we use the combined method for selecting the given clause, but this time we do not simplify the set of processed clauses. Also for this case we add special theory axioms.
- We did not use an external SMT solver for both provers.

As before, the symbol "x" indicates, that the prover terminated without a prove due to exceeding the set time limit.

Comparison for valid formulae of the MESON prover and the resolution prover in ms				
Example	Success	MESON	Resolution prover	
		default settings	Combined	Combined without simplifying processed clauses
arith1.txt	+	412	8465	25065
arith3.txt	-	393	x	x
arith4.txt	+	660	14124	7834
barber.txt	+	111	67	68
case.txt	+	117	65	56
davisPutnam.txt	+	120	50	50
drinker.txt	+	133	137	88
ewd1062.txt	+	16647	1366	702
expand.txt	+	109	56	59
fol.txt	+	133	190	159
fol0.txt	+	105	44	32
fol3.txt	+	129	56	59
fol4.txt	+	130	82	87
group2.txt	+	467	4086	3158
leq.txt	+	60243	127	115
los.txt	+	543	22786	21898
nat.txt	+	117	132	49
nat2.txt	+	131	149	116
numbers.txt	+	254	234	138
numbers2.txt	+	69	0	0
pelletier20.txt	+	122	50	41
pelletier24.txt	+	116	67	66
pelletier32.txt	+	117	91	79
pelletier45.txt	+	160	266	306
perm.txt	+	137	142	160
sat.txt	+	242388	x	1644
sort.txt	+	298	76496	72879
split.txt	-	1681986	x	x
steamroller.txt	+	225	3638	3131

Table 5.2: Runtime comparison of the MESON prover and the resolution prover in ms

Using the MESON prover we were able to create a proof for satisfiability for a total of 30 examples. With the resolution prover, we were able to find a proof for 28 of those examples. It is interesting to see that the resolution prover is slightly faster than the MESON prover for most of the small examples. But the MESON prover is much stronger for bigger examples containing more clauses. For the examples we were not able to find a proof with the resolution prover, we strongly believe that this is an issue of time.

5.2.3 Comparison of the Resolution Prover and the MESON Prover Using an External SMT Solver

In Table 5.3 we again show a comparison of the MESON prover and the resolution prover, but this time using the support of an external SMT solver for both provers. As in the other tables, we only show examples, where the MESON prover was able to show the validity of the input first-order logic formula. This enable to compare, if the resolution prover is able to find a proof for an example, when the MESON prover does. If both provers find a proof we declare this with the symbol "+" in the column "Success". In this case we can also compare the runtime. If the resolution prover can not generate a proof, but the MESON prover can find one, we print the symbol "-". The entry "x" indicates, that the algorithm terminated without a prove due to a set time limit. For the comparison we use the following settings:

- For both provers we use the external SMT solver Z3 [10].
- As selection function for the given clause we chose the method `combined` (fifo and clause weight combined) and "simplify everything". We do not add any axioms for special theories (as integers or arrays). The SMT solver Z3 gets applied in every iteration of the main loop to the whole proof problem (current knowledge base).
- For the MESON prover we again use the default settings, but use two different settings for the use of the SMT solver:
  1. "min": The SMT solver is only applied in the MESON prover fails to find proof for the proof situation. It is only applied to a part of the proof problem — clauses containing universally quantified variables [36].
  2. "med": In this case the SMT solver gets applied first in every proof situation. Only if the SMT solver fails to generate a proof, the MESON prover is applied [36].

Runtime comparison in ms for valid formulae of the MESON prover and the resolution prover, both using a SMT solver				
Example	Success	MESON		Resolution prover
		Min (-mesmt 1)	Med (-mesmt 2)	comb
arith1.txt	+	695	832	85
arith3.txt	+	339	294	350
arith4.txt	+	286	324	219
barber.txt	+	190	220	234
cancellation.txt	-	10187	10886	x
case.txt	+	214	231	207
ccvalid.txt	+	203	230	242
davisPutnam.txt	+	216	248	219
div2.txt	+	663	771	763
drinker.txt	+	249	282	209
eqarray.txt	-	6159	6997	x
eqcase.txt	+	1125	1276	505
equality.txt	+	637	503	515
ewd1230.txt	+	425	488	491
ewd1266a.txt	+	361	365	394
expand.txt	+	183	223	15468
fol.txt	+	278	416	770
fol0.txt	+	150	205	203
fol3.txt	+	289	214	171
fol4.txt	+	246	250	242
group2.txt	-	541	594	x
leq.txt	+	408	409	485
lists.txt	+	331	434	307
lists2.txt	+	311	420	325
los.txt	-	927	985	x
nat.txt	+	270	261	210
nat2.txt	+	272	345	38810
numbers.txt	+	540	537	619
numbers2.txt	+	135	135	214
pelletier20.txt	+	220	244	240
pelletier24.txt	+	195	229	193
pelletier32.txt	+	247	306	229
pelletier45.txt	+	651	774	833
perm.txt	+	266	395	37091
rewrite.txt	+	248	330	75348
sat.txt	+	3480	4050	1002
sort.txt	-	1373	1458	x
split.txt	+	3144	3413	1338
steamroller.txt	-	606	652	x
step.txt	+	365	442	429
swap.txt	-	1810	1893	x
types.txt	+	510	615	655
wishnu.txt	+	743	1024	85

Table 5.3: Runtime comparison of the MESON prover and the resolution prover, both using a SMT solver, in ms

We notice that, the majority of examples that were successfully proven with the MESON prover also worked with our prover (7 examples out of 43 did not). For the runtime, we can not derive a clear general conclusion which prover is faster, the runtime varies a lot for both. As a positive statement, using a SMT solver helped finding a proof for more examples with both provers (13 more for the meson prover, 8 for our resolution prover).

Chapter 6

Future Work

In this chapter we point out some possibilities to further improve our algorithm, how to supplement current features or add new properties. All this possible future work would lead to an increase in the performance level, better management of the search space or adds additional functionality to the program. In the following we give examples on how to further improve the theorem prover and the integration of new tools. Obviously, these are not the only possibilities and there would be much more.

Additional Simplifying Inference Rules

As one of the most obvious improvement for our prover, it is always possible to add more simplifying inference rules. Some rules are already pointed out in this thesis like "positive simplify-reflect" or "equality subsumption". Another idea is to additionally implement some kind of AC redundancy elimination (associativity and commutativity). These tow properties are often encountered when dealing with binary operators. AC redundancy elimination is very tricky to implement and in practice quite hard to use efficiently, therefore no general purpose prover currently in the market implements this theory. The E-Theorem prover integrate an alternative approach based on the AC theory. They include explicit AC axioms while performing a normal proof search. The AC theory is only used for clause simplification and to delete redundant clauses.

Experimenting with the Term Ordering

We only implemented the lexicographic path ordering, but with the option for the user to set the weights for the variable and function symbols. Not only can one experiment with the current ordering LPO and different weights, but it is possible to integrate new orderings. One quite popular option would be the Knuth Bendix ordering. To implement this ordering a weight function has to be defined, that assigns weights to the function symbols. All variable symbols get a fixed weight. The function symbols have to have a precedence. An advantage of this ordering is, that is does not allow rewriting, that would increase the number of symbols in a clause. Function symbols with a high arity get eliminated over time [38].

Integrating a Literal Selection Function

As describes earlier in this thesis we decided not to integrate a selection function. However, the E-Theorem prover shows, that a good selection function can have a large impact on the efficiency of the prove search [38]. There are two separate questions to answer:

1. Do we want to select literals in this clause?
2. Which literals do we want to select in a clause?

One interesting and useful literal selection function always selects an arbitrary negative literal out of all literals with the largest difference in the number of symbols comparing both sides of the equation. Another popular option is to select the first literal of the form $s \neq t$. If the clause does not contain such a literal, but there exits at least one negative ground literal, the smallest negative ground literal is selected. In all other cases this function selects the literals as the first described option. For the E-Theorem Prover this is one of the strongest literals selection functions [38].

Improve Clause Selection

Selecting the givenclause can be very crucial regarding the runtime of the algorithm and shorten the proof significantly. We chose to implement two different, well-known options, but there are many more and special techniques of combining them. One additional way would be the *refined weight* clause selection. This method is an advanced version of symbol counting (respectively the clause weight method). Maximals terms and literals (or alternatively selected literals) are assigned a higher weight. One advantage is that only maximal literals are admissible for generating new clauses. Preferring smaller clauses delays the explosion of the search space. Additionally, this method chooses orientable clauses over unorientable ones (for unit clauses), which leads to a stronger application of the rewrite inference rules earlier [38].

In the paper [38] it is explained, that experimental tests with the E-Theorem Prover [44] show, that combining different methods for selecting the given clause deliver the best results. For example one could set up two priority queues, one sorted accoring to age (fifo) and one sorted accoring to the number symbols (clauseweight). When selecting the givenclause the clauses are picked alternately from those two queues by a fixed ratio. Naturally, every a clause has to be added to every queue and also deleted from every queue. In general, there could be an unlimited number of priority queues combined with each other. Experiments done with the E-Theorem Prover [44] (not by us, but in [38]) show that three or four priority queues deliver the best results. There would be two different versions of the refined weight method and the other queues sorted by some kind of fifo method (sorted by age) [38].

Chapter 7

Conclusion

In this thesis we developed a saturation based automated theorem prover for first-order logic with equality as an extension of the RISCTP software. Since RISCTP already has a built-in goal-oriented prover, the idea was to create a knowledge based system. We started with a comprehensive literature research to identify possible strategies and extracted the most reliable and most promising procedures for the design of our own prover. The results are shown in a throughout theoretical representation of different first-order logic prove strategies. The design of the prover is therefore based on well-known, modern techniques for first-order logic theorem proving. For the equality reasoning we apply the same procedure as the theorem prover E does and “force” every literal to become an (in)equality. In general, we use the superposition calculus (for generating new clauses) including the rules equality resolution, equality factoring and superposition into positive and negative literals. For easier search space organisation we designed the prover around a given-clause-algorithm with a DISCOUNT loop. According to this algorithm, in every traversal of the main loop a clause has to be picked (given clause). We provide three different options: first-in-first-out, clause-weight and a combined version (notion of fairness). Since the search space increased rapidly, we added several, explicitly defined, simplifying inference rules to speed up the proof search. These include deletion of duplicated literals, deletion of resolved literals, two rules for syntactic tautology deletion, destructive equality resolution, clause subsumption, rewriting positive and negative literals and negative simplify-reflect. To enable the reasoning with special theories (integers and arrays) is is possible to add axioms to the prove problem, encoding these theories. Additionally, an external SMT solver is integrated (currently only Z3 can be used).

The implementation itself is done with the programming language JAVA and the prover gets added to the RISCTP software as an additional option to the MESON prover in immediate future (https://www.risc.jku.at/research/formal/software/RISCTP/). The evaluation of the implemented prover was done by testing it with various examples, using the same ones as when evaluating the MESON prover. This also ensures, that the two provers are comparable. With those examples we performed test with different user settings and tracked the run time (also in comparison to the MESON prover). Overall, our resolution prover is for larger examples noticeable slower than the MESON prover, whereas for small examples the resolution prover is a bit faster. For most examples the runtime is quite similar. The use of a SMT solver speeds up larger examples, but at the same time might slow down smaller examples. We also noticed that the runtime varies (a lot) for different user settings and at the same time we are not able to identify a clear “best” setting for all examples. For the majority of the examples using all implemented simplification strategies paired with the combined method of fifo and clause weight for selecting the given clause performs best.

Although the resolution prover still has quite a lot of potential for improvement, the thesis illustrates, that the implemented prover is a good, nice alternative to the currently only option “MESON prover” for understanding proofs and how they are created.

Bibliography

[1] Leo Bachmair and Harald Ganzinger. 1990. On restrictions of ordered paramodulation with simplification. In *10th International Conference on Automated Deduction*. Mark E. Stickel, (Ed.) Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 427–441. DOI: 10.1007/3-540-52885-7_105.

[2] Leo Bachmair and Harald Ganzinger. 1991. Rewrite-Based Equational Theorem Proving With Selection and Simplification. Department of Computer Science , SUNY at Stony Brook, NY, USA and Max-Planck-Institut für Informatik, Im Stadtwald, Saarbrücken, Germany, (September 1991). DOI: 10.1093/logcom/4.3.217. https://pure.mpg.de/rest/items/item_1834970_5/component/file_1857487/content.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tine. [n. d.] SMT-LIB the satisfiability modulo theories library. The SMT-LIB Initiative. (). https://smtlib.cs.uiowa.edu/index.shtml.

[4] [n. d.] Boolector. GitHub. (). https://boolector.github.io/.

[5] Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation. Decision Procedures with Applications to Verification.* (1st ed.). Springer Berlin, Heidelberg. DOI: 10.1007/978-3-540-74113-8.

[6] D. Brand. 1975. Proving Theorems with the Modification Method. *SIAM Journal on Computing*, 4, 4, 412–430. DOI: 10.1137/0204036.

[7] [n. d.] CVC5. GitHub. (). https://cvc5.github.io/.

[8] Martin Davis and Hilary Putnam. 1960. A Computing Procedure For Quantification Theory. In *Journal of the ACM.* Association for Computing Machinery, New York, NY, United States, pp. 201–215. DOI: 10.1145/321033.321034.

[9] Nikolaj De Moura Leonardo; Bjørner. 2011. Satisfiability modulo theories : introduction and applications. *Communications of the ACM*, 54, 69, 9. ISSN: 0001-0782. DOI: 10.1145/1995376.1995394.

[10] Leonardo de Moura and Nikolaj Børner. 2008. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin, Heidelberg, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[11] Melvin Fitting. 1997. *First-order logic and automated theorem proving. Texts in Computer Science*. Springer, New York, NY, USA. DOI: 10.1007/978-1-4612-2360-3.

[12] Laurent Fribourg. 1985. A superposition oriented theorem prover. *Theoretical Computer Science*, 35, 129–164. DOI: 10.1016/0304-3975(85)90011-8.

[13] John Harrison. 2009. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, Cambridge, UK. DOI: 10.1017/CBO9780511576430.

[14] Jieh Hsiang and Michael Rusinowitch. 1991. Proving refutational completeness of theorem-proving strategies: the transfinite semantic tree method. *Journal of the ACM*, 38, 558–586, 3. ISSN: 0004-5411. DOI: 10.1145/116825.116833.

[15] Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. 2016. The vampire and the FOOL. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. ACM. DOI: 10.1145/2854065.2854071.

[16] Laura Kovacs and Andrei Voronkov. 2013. First-Order Theorem Proving and VAMPIRE. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, pp. 1–35. DOI: 10.1007/978-3-642-39799-8_1.

[17] Daniel Kroening and Ofer Strichman. 2016. *Decision Procedures. An Algorithmic Point of View*. Springer Berlin, Heidlberg, Heidelberg. DOI: 10.1007/978-3-662-50497-0.

[18] Temur Kutsia. 2020. First Order Predicate Logic 4. Resolution. https://www3.risc.jku.at/education/courses/ws2019/complog/FOL-4-Resolution_23-Jun-2020.

[19] William McCune. 2003. OTTER 3.3 Reference Manual. Technical Report. Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439. DOI: 10.48550/arXiv.cs/0310056.

[20] Hans Jürgen Ohlbach, Dov M. Gabbay, and David A. Plaisted. 1994. Killer transformations. Research Report MPI-I-94-226. Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, (June 1994).

[21] Dag Prawitz, Haåkon Prawitz, and Neri Voghera. 1960. A Mechanical Proof Procedure and its Realization in an Electronic Computer. *Journal of the ACM*, 7, 102–128, 2. DOI: 10.1145/321021.321023.

[22] Virgile Prevosto and Uwe Waldmann. 2006. Spass+T. *Sutcliffe, Geoff; Schmidt, Renate; Schulz, Stephan: ESCoR: FLoC'06 Workshop on Empirically Successful Computerized Reasoning*, 192, 19–33.

[23] Giles Reger and Martin Suda. 2017. Set of Support for Theory Reasoning. In IWIL@LPAR 2017 Workshop and LPAR-21 Short Presentations, pp. 124–134. DOI: 20.500.12708/57149.

[24] Giles Reger, Martin Suda, and Andrei Voronkov. 2017. Instantiation and Pretending to be an SMT Solver with Vampire. In 15th International Workshop on Satisfiability Modulo Theories, SMT 2017, pp. 63–75. DOI: 20.500.12708/86623.

[25] Franz-Xaver Reichel. 2020. *The Integration of SMT Solvers into the RISCAL Model Checker*. Master Thesis. Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria.

[26] Alexandre Riazanov and Andrei Voronkov. 2003. Limited resource strategyy in resolution theorem proving. Journal of Symbolic Computation. Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK, pp. 101–115. DOI: 10.1016/S0747-7171(03)00040-3.

[27] Alan Robinson and Andrei Voronkov. 2001. *Handbook of Automated Reasoning.* Volume 1. Elsevier, Amsterdam and Co-publisher The MIT Press, Cambridge, Massachusetts, printed in The Netherlands. DOI: 10.5555/581809.

[28] George Robinson and Lawrence Wos. 1969. Paramodulation and Theorem-proving in First-Order Theories with Equality. In Machine Intell. 4, pp. 135–150. DOI: 10.1142/9789812813411_0008.

[29] John Alan Robinson. 1965. A machine-oriented logic based on the resolution principle. Journal of the ACM. Argonne National Laboratory and Rice University, pp. 23–41. DOI: 10.1145/321250.321253.

[30] John Alan Robinson. 1974. Automatic deduction with hyper-resolution. In volume 39. Association for Symbolic Logic, pp. 189–190. DOI: 10.2307/2272384.

[31] Michael Rusinowitch. 1991. Theorem-proving with Resolution and Superposition: Extension of Knuth Bendix Procedure to complete Set of inference Rules. 11.

[32] Alex Sakharov. 2023. Knuth-Bendix Completion Algorithm. Wolfram MathWorld. (2023). https://mathworld.wolfram.com/Knuth-BendixCompletionAlgorithm.html.

[33] Wolfgang Schreiner. 2023. *Concrete Abstractions. Formalizing and Analyzing Discrete Theories and Algorithms with the RISCAL Model Checker.* Springer Cham, Linz, Austria. DOI: 10.1007/978-3-031-24934-1.

[34] Wolfgang Schreiner. 2019. The RISC Algorithm Language (RISCAL). Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria. (2019). https://www.risc.jku.at/research/formal/software/RISCAL.

[35] Wolfgang Schreiner. 2022. The RISCTP Theorem Proving Interface. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria. (2022). https://www3.risc.jku.at/research/formal/software/RISCTP/.

[36] Wolfgang Schreiner. 2024. The RISCTP Theorem Proving Interface — Tutorial and Reference Manual (Version 1.8.*). Technical Report. the latest version is available at [35]. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, (July 2024).

[37] Wolfgang Schreiner, Alexander Brunhuemer, and Christoph Fürst. 2018. Teaching the Formalization of Mathematical Theories and Algorithms via the Automatic Checking of Finite Models. In *Post-Proceedings ThEdu'17, Theorem proving components for Educational software, Gothenburg, Sweden, August 6, 2017* (*EPTCS*, volume 267). Pedro Quaresma and Walther Neuper, (Eds.), pp. 120–139. DOI: 10.4204/EPTCS.267.8.

[38] Stefan Schulz. 2002. E - a brainiac theorem prover. In volume 15. AI Communication, pp. 111–126. DOI: 10.5555/1218615.1218621.

[39] Stephan Schulz. 2013. System Description: E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning.* Ken McMillan, Aart Middeldorp, and Andrei Voronkov, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 735–743. ISBN: 978-3-642-45221-5.

[40] Thoralf Skolem. 1928. Über die mathematische Logik. 10. pp. 125–142 (german), English translation On Mathematical Logic by Stefan Bauer-Mengelberg and Dagfinn Føllesdal with an introduction by Burton Dreben and Jean van Heijenoort in: Heijenoort (1971), pp. 508–524 (english).

[41] [n. d.] SMT-COMP The International Satisfiability Modulo Theories (SMT) Competition 2022. (). https://smt-comp.github.io/2022/.

[42] 2022. SPASS Workbench. Max Planck Institut for Informatics, Saarbrücken, Germany. (2022). https://www.mpi-inf.mpg.de/departments/automation-of-logic/software/spass-workbench/.

[43] Ágoston Sütő. 2023. *Verification of non-deterministic systems using LTL model checking in RISCAL*. Master Thesis. Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria.

[44] [n. d.] The E Theorem Prover. DHBW Stuttgart. (). https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html.

[45] [n. d.] Vampire. (). https://vprover.github.io/team.html.

[46] Wolfgang Schreiner with contributions by Franz-Xaver Reichel. 2022. The RISC Algorithm Language (RISCAL) — Tutorial and Reference Manual (Version 4.2). Technical Report. Available at [34]. RISC, Johannes Kepler University, Linz, Austria.

[47] Lawrence Wos, George Robinson, Daniel Carson, and Leon Shalla. 1967. The Concept of Demodulation in Theorem Proving. Argonne National Laboratory, Argonne, Illinois, The University of Wisconsin, Madison, Wisconsin, and Stanford Linear Accelerator Center, Stanford, California, pp. 698–709. DOI: 10.1145/321420.321429.