TU WIEN Informatics

Vampires can Lean into Trust:

Checking Proofs for Soundness

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Artificial Intelligence

eingereicht von

Dipl.-Ing. Jonas Bodingbauer, BSc BSc
Matrikelnummer 11802486

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof.in Dr.in techn. Laura Kovács, MSc

Wien, 11. Mai 2026

Jonas Bodingbauer

Laura Kovács

TU WIEN Informatics

Vampires can Lean into Trust:

Checking Proofs for Soundness

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Artificial Intelligence

by

Dipl.-Ing. Jonas Bodingbauer, BSc BSc
Registration Number 11802486

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof.in Dr.in techn. Laura Kovács, MSc

Vienna, May 11, 2026

Jonas Bodingbauer | Laura Kovács

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Jonas Bodingbauer, BSc BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang "Übersicht verwendeter Hilfsmittel" habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 11. Mai 2026

Jonas Bodingbauer

Acknowledgements

I want to thank my Supervisor Laura Kovács for her support and guidance throughout the thesis. She brought this fascinating topic to my attention and provided me with feedback and help when I needed it. Furthermore, I want to thank my colleagues at the APRe group for the friendly working atmosphere and discussions. In particular, I want to thank Axel, who I could discuss many ideas with and who provided several insights on Lean. Furthermore, I want to thank Alex and Axel for proofreading (parts of) this thesis. I also want to thank the members of the Vampire team (largely coinciding with the APRe group) for feedback on the implementation and help regarding the proof-replay method. Without preexisting work on proof output generation in Vampire, this thesis would not have been possible in the given time frame.

Furthermore, I want to thank my family who supported me during all my studies and the somewhat foolish idea of pursuing two degrees in a more or less parallel fashion. Without their help, I would not be where I am today, and I am very grateful for that.

Last but not least, I want to thank Lena for her wholehearted support for the path I decided to take. For listening to me when I ranted about technical issues and for being a stable foundation on which I can always rely on.

Funding Acknowledgement:
This research was funded in whole or in part by the ERC Consolidator Grant ARTIST 101002685, the ERC Proof of Concept Grant LEARN 101213411, and by the SBA Research COMET Center SBA-K1 NGC managed by the FFG.

Kurzfassung

Moderne automatische Theorembeweiser (ATPs) wie Vampire sind große Programme, die symbolisches Schließen für ein breites Anwendungsspektrum ermöglichen, darunter Softwareanalyse, die Formalisierung von Mathematik und Cybersicherheit. Obwohl der zugrunde liegende Kalkül von Vampire als korrekt bewiesen ist, kann dessen Implementierung Fehler enthalten, die zu inkorrekten Beweisen führen können. Diese Arbeit präsentiert eine Methode, um das Vertrauen in solche Beweise zu erhöhen, indem ihre Korrektheit vollständig überprüft wird. Dazu werden die von Vampire erzeugten Beweise in eine Form übersetzt, die mit dem interaktiven Theorembeweiser Lean geprüft werden kann. Die vorgestellte Methode unterstützt die CNF- (clause normal form) und FOF- (first-order form) Fragmente der Prädikatenlogik erster Stufe mit Gleichheit.

Um eine vollständige Beweisprüfung einschließlich der Vorverarbeitung (Preprocessing) zu ermöglichen, werden in dieser Arbeit zwei Änderungen an Vampire bezüglich der Skolemisierung und der Behandlung purer Prädikate eingeführt und separat evaluiert. Darüber hinaus wird eine Methode des "proof replay" in Vampire vorgestellt, die während der Beweissuche aus Performanzgründen verworfene Beweisdetails rekonstruiert, welche jedoch für eine effiziente Rekonstruktion der Beweise in Lean relevant sind. Die Möglichkeit, einen Beweis vollständig durch den vertrauenswürdigen Kernel von Lean zu überprüfen, stellt eine erhebliche Steigerung des Vertrauens in die von Vampire generierten Beweise dar. Dadurch werden solche Beweise insbesondere für sicherheitskritische Anwendungen, aber auch für die weiteren Einsatzgebiete von Vampire noch wertvoller.

Die vorgestellte Methode wird auf einem großen Satz von Benchmark-Problemen (TPTP 9.2.1) evaluiert und zeigt, dass 99% der von Vampire (unter einem Instruktionslimit von 100 Giga-Instruktionen) gefundenen Beweise mit einem Zeitlimit von 1000 s erfolgreich in Lean geprüft werden konnten. Außerdem wurden zwei weitere, weniger spezialisierte Ansätze zur Beweisprüfung unter Verwendung eines externen beweisproduzierenden ATPs (Duper) evaluiert, die den Performanzvorteil der vorgestellten Methode demonstrieren.

Abstract

Modern automated theorem provers (ATPs) such as Vampire are large programs that provide symbolic reasoning capabilities for a wide range of applications, including software analysis, the formalization of mathematics, and cybersecurity. While the underlying calculus of Vampire is known to be sound, its implementation may contain bugs that can lead to unsound proofs. This thesis presents a method to increase trust in these proofs by verifying their soundness end-to-end, translating them into a form that can be checked using the interactive theorem prover (ITP) Lean. The presented approach applies to the CNF (clause normal form) and FOF (first-order form) fragments of first-order logic with equality. To achieve full proof checking including preprocessing, this work introduces two modifications concerning skolemization and the handling of pure predicates in Vampire and evaluates them separately. Furthermore, it introduces a "proof-replay" method in Vampire that reconstructs proof details discarded during proof search for performance reasons, but relevant for efficient proof reconstruction in Lean. The ability to check a proof end-to-end by a trusted kernel in Lean is a significant increase in trust in the proofs generated by Vampire, which makes such proofs even more valuable for safety-critical applications as well as in the other application domains of Vampire.

The presented method is evaluated on a large benchmark set (TPTP 9.2.1), showing that 99% of proofs found by Vampire (within an instruction limit of 100 giga-instructions) could be successfully checked in Lean with a time limit of 1000 s. Furthermore, two other, less specialized, approaches to proof checking using an external proof-producing ATP (Duper) were evaluated and compared against the developed method, demonstrating the performance benefit of the proposed approach.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
- 1.1 Related Work . . . 2
- 1.2 Research Questions . . . 4
- 1.3 Structure of this Thesis . . . 4

2 Background **5**
- 2.1 A Brief Introduction to First-Order Logic . . . 5
- 2.2 Superposition Calculus; or, How Vampire Finds Proofs . . . 10
  - 2.2.1 Saturation Loop . . . 10
  - 2.2.2 Inference Rules . . . 11
  - 2.2.3 Further Inferences and Theory Reasoning . . . 12
- 2.3 Dependent Type Theory; or, How Lean Checks Proofs . . . 13
  - 2.3.1 Special Types . . . 14
  - 2.3.2 Universes . . . 16
  - 2.3.3 Curry–Howard Correspondence . . . 16
  - 2.3.4 Example of the Embedding in Lean . . . 18
- 2.4 AVATAR: Advanced Vampire Architecture for Theories and Resolution 19
  - 2.4.1 Splitting of Clauses . . . 19
  - 2.4.2 Constructing A-Clauses . . . 19
  - 2.4.3 First-Order Reasoning over A-Clauses . . . 20
  - 2.4.4 SAT Solver and First-Order Reasoning Combined . . . 20

3 Vampire to Lean Proof Translation **21**
- 3.1 Trust Model . . . 21
- 3.2 System Design . . . 22
  - 3.2.1 User Workflow . . . 23
  - 3.2.2 Internal Program Flow . . . 24
- 3.3 Running Example . . . 25
- 3.4 General Structure of the Generated Lean File . . . 26
  - 3.4.1 Structure of the Generated Lean File for Running Example . . . 27

3.5 Preprocessing . . . 29
3.5.1 Steps Representable as Rewrite Rules . . . 29
3.5.2 Rectification . . . 32
3.5.3 CNF Transformation . . . 33
3.5.4 Preprocessing Steps that Introduce new Symbols . . . 35
3.5.5 Predicate Definition . . . 35
3.5.6 Skolemization . . . 36
3.5.7 Summary . . . 40
3.6 Saturation Loop Inferences . . . 40
3.6.1 Proof Replay . . . 41
3.6.2 Example Inference . . . 43
3.6.3 Design Choices for Superposition Inference Rules . . . 45
3.7 AVATAR Inferences . . . 45
3.7.1 Definition Introduction . . . 46
3.7.2 A-Clauses in Inference Rules . . . 46
3.7.3 Contradiction and Component Clauses . . . 46
3.7.4 Splitting . . . 47
3.7.5 AVATAR Refutation . . . 48
3.7.6 Summary . . . 49

4 Experiment and Results **51**
4.1 Experimental Setup . . . 51
4.2 Evaluation of Proof Inference Modifications . . . 52
4.2.1 Pure Predicate Removal . . . 52
4.2.2 New Skolemization Method . . . 53
4.2.3 Combined Methods . . . 54
4.2.4 Proof Replay and Output Generation . . . 54
4.3 Evaluation of Lean Proof Checking . . . 56
4.3.1 Comparisons with Duper . . . 56
4.3.2 Results . . . 58

5 Conclusion and Future Work **63**

Overview of Generative AI Tools Used **65**

List of Figures **66**

List of Tables **66**

List of Algorithms **67**

List of Listings **68**

Acronyms **69**

Bibliography **70**

CHAPTER 1

Introduction

Automated reasoning is a foundational tool in computer science and engineering. Symbolic reasoning can provide trustworthy guarantees about derived conclusions with verifiable certificates of correctness. In other words, an automated reasoner can produce a proof of given conjecture based on formalized assumptions. This is particularly important for safety-critical application such as software analysis [BF25; Ahr+00; Geo+22], checking whether a given program behaves correctly. Well-known examples of the need for such software analyses are the Ariane 5 rocket failure [Lio96] and the Therac-25 radiation therapy machine accidents [LT93], which were both caused by software errors that were potentially preventable by formal software analysis. Further applications of automated reasoning include the formalization and proof of mathematics [Mcc97; US10; Bol+25], cybersecurity [Run22; Jea+24; LOB24], and machine learning with neural networks [BHS26; Des+25].

Various different programs have been developed to perform automated reasoning, such as automated theorem provers (ATPs) and satisfiability modulo theories (SMT) solvers. These programs can automatically find proofs for a formalized conjecture without human intervention. Vampire is a powerful superposition-based ATP which has won all categories in the CASC competition in 2025 [Sut25]. However, the proofs generated by Vampire can be very long and can become very tedious for a human to check, which motivates the need for external proof verification. While the underlying calculus of Vampire is known to be sound, the implementation of the calculus may contain bugs, which could lead to unsound proofs being generated. This would be particularly problematic in safety-critical applications, where the consequences of relying on an unsound proof could be severe.

Proof-checking is already somewhat established for other automated reasoners, such as satisfiability (SAT) solvers, where the underlying theory is propositional logic [WHH14]. However, the set of separate inference rules used by SAT solvers is smaller and simpler

than the rules that are required for a more expressive logic such as first-order logic with equality, which is the underlying logic of Vampire.

Checking saturation calculus proofs has been approached in different ways previously, where a convenient approach is to use another ATP or SMT solver to externally reprove each inference step of a generated proof [Raw+25; Lac+24; Arm+11]. This increases trustworthiness of the proof, but the external solver is still not guaranteed to be sound, although it is unlikely that two different solvers would contain a bug that leads to a similarly unsound proof. However, the main problem with this approach is that some inferences may be hard to verify for an external solver making it infeasible to check the proof. For example, eliminating existential quantifiers is not trivially a sound inference step, since it implicitly relies on the choice axiom, which can be implemented in different ways. It can be therefore hard to check such inference steps.

Alternative approaches rely on interactive theorem provers (ITPs) to check a proof inference-by-inference [MP08; Moh+25]. This method greatly reduces the trusted codebase, as ITPs are engineered to have a small trusted kernel, which is the only part of the system that (typically) needs to be trusted for soundness. The remaining code around the kernel is used to provide a convenient interface for users, providing a rich set of automation and the possibility to implement custom methods within the system. Nevertheless, the use of an ITP for proof checking comes with several challenges as well, such as different underlying formalisms, varying treatment of associative and commutative connectives and the need to detail a proof with more granularity than the original proof.

Furthermore, modern ATPs such as Vampire have a growing number of inference rules and supported theories, which makes it challenging to automate proof-checking for all possible inference steps in an ITP. Hence, an ITP that provides powerful and extensible automation such as Lean is a good candidate for proof-checking Vampire proofs with reconstructing proof details not only in Vampire, but also in Lean, being a midpoint between the fully automated approach using an external solver and the fully manual approach of translating the full proof in every detail.

The goal of this thesis is to produce proof output in Vampire for a subset of inference rules that can be checked for soundness in Lean. The proof reconstruction effort, in other words, the effort of generating details of the proof that are not explicitly given in the originally generated proof is shared between Vampire and Lean.

A major goal is that the full proof from axioms to conclusion is checked in Lean, which means that preprocessing steps that have not been considered in previous works, are also checked.

1.1 Related Work

The Vampire theorem prover [Bár+25] is a state-of-the-art automated theorem prover that won the CASC-30 [Sut25] competition in all categories against competing systems.

It implements several advanced reasoning techniques, most notably AVATAR [Vor14] and ALASCA [Kor+23].

A wide range of proof checkers, more specifically ITPs, exists, each with different design goals and underlying formalism, which may be considered for checking the soundness of proofs generated by Vampire. Isabelle [NWP02] is based on higher-order logic, while most other commonly used proof checkers, such as Rocq [Tea24], Agda [BDN09], and Lean [MU21], are based on variants of dependent type theory. Several ATPs are able to generate proofs tailored to specific proof checkers. For example, Duper [Clu+24] is a superposition based ATP that produces proof terms for Lean. Metis [Hur03] can find proofs for Isabelle, and sauto [Cza20] provides automated proof search for Roqc. The primary purpose of these systems is to assist interactive theorem proving, and as a result their goal is not necessarily to achieve maximal performance in proof search, since the overall process is guided by a human user.

Furthermore, several interactive theorem provers feature so-called "hammers", which invoke an ATP to find a proof obligation arising in the human formulated full proof, and then produce a proof that can be used within the ITP. Translating a proof produced by an ATP into a form that can be reconstructed and understood by the kernel of an ITP is a key building block for using such systems as hammers within interactive theorem proving. Notable examples include Sledgehammer [BN10], which uses several ATPs, and coq-hammer [CK18].

Efforts have been made to produce verifiable proofs in Vampire. Ground Truth [Raw+25] is an approach in which each individual proof step is grounded and then passed to an SMT solver. All inference steps are checked individually at the ground level in order to fit the capabilities of SMT solvers. This grounding reduces the generality of the inference steps, which means that preprocessing steps cannot be included, as well as other inference steps [Raw+25]. Furthermore, SMT solvers themselves are rather big pieces of software and are not completely trusted. Consequently, this approach should be seen as a way to enhance trust in a proof rather than to provide full verification. Another attempt to verify proofs generated by Vampire uses the $\lambda\Pi$ modulo theory calculus [KRS25], with proofs checkable in Dedukti [Ass+23]. This approach checks individual inference steps by manually constructing proof terms inside Vampire to establish the correctness of each step. However, the range of supported inference steps is limited, and preprocessing steps are excluded for this approach as well. All the proof reconstruction effort is left to Vampire, which makes it harder to implement new inference rules, since each proof step needs to be handled manually in Vampire. Therefore, the approach of this thesis is to share the proof reconstruction effort between Vampire and Lean, which allows for more flexibility when engineering the proof output of Vampire.

Furthermore, interfacing efforts have been made from Lean's dependent type theory to less expressive logics such as higher-order logic by lean-auto [Qia+25], which could enable the use of Vampire as a hammer for Lean with further development.

1.2 Research Questions

The main research question of this thesis is:

How can one check the soundness of proofs generated by Vampire end-to-end in Lean?

Furthermore, sub-questions are:

- How can one include preprocessing steps such as e.g Skolemization, which is known to be a non-trivial inference step to proof-check, in the proof checking process?
- How does one effectively structure a saturation calculus proof for an interactive theorem prover?
- How to reconstruct proof details in a way that is effective and flexible for future changes?
- In which way can this method compare to previous approaches such as running an external proof-producing ATP on each inference step?

1.3 Structure of this Thesis

This thesis is structured in five chapters. After this introduction, Chapter 2 introduces the relevant background required for this thesis. First, the basics of first-order logic (with equality) are introduced, which Vampire works in. Then, the dependent type theory underlying Lean is described, and how first-order logic can be incorporated into dependent type theory is explained. Finally, AVATAR is described, which is an important method that Vampire uses.

In Chapter 3, the methods of proof translation are described in detail. This includes a description how to use the proof output of Vampire, the layout of the generated proof output, and implementation details. The proof-replay method is introduced, which reconstructs proof details in Vampire that are discarded during proof search.

The resulting artifact is evaluated in Chapter 4, in which the implemented preprocessing changes, as well as the proof-replay overhead in Vampire, are measured.

Finally, Chapter 5 concludes the thesis and gives an outlook on future work.

Peer-reviewed publication at the ITP 2026 conference The work described in this thesis is partially described in the paper "Lean on Vampire Proofs (Short Paper)" [Bod+26], that was accepted at the 17$^{\text{th}}$ International Conference on Interactive Theorem Proving - ITP 2026. This thesis describes the methods in more detail than the paper and provides additional background as well as a more detailed evaluation.

CHAPTER 2

Background

This chapter provides the background needed for the methods used in the remainder of the thesis. It first describes the underlying logic and the proof-search methods used by Vampire. It then introduces the dependent type theory underlying Lean and presents a method for embedding first-order logic into dependent type theory.

2.1 A Brief Introduction to First-Order Logic

Vampire is a superposition-based automated theorem prover for first-order logic (FOL) with equality [Bár+25]. The goal of this section is to provide enough, albeit informal, detail to understand the concepts needed for proof search[1], but not a detailed description of syntax and semantics, which can be found in standard textbooks such as [Fit96]. No explicit definition of the syntax of FOL is provided in this section, and adheres to the standard syntax of FOL with equality.

Semantics The meaning of a formal language is given by its semantics, which defines the interpretation of the symbols used in the syntax of the language. It also determines what it means for a formula to be true or false, and thus provides the basis for satisfiability and validity, the main properties of interest for automated theorem provers.

Definition 2.1.1 (Structure)**.** A structure S of a first-order language $\mathcal{L}$ is a pair $(D, F) = S$ where D is the domain (a non-empty set), and F is an interpretation function that assigns meanings to the symbols of $\mathcal{L}$.

To give meaning to formulas with free variables (variables that are not bound by quantifiers), an interpretation provides an assignment to these variables.

[1] Different sources name things slightly differently. In this work, the term "model" refers to a structure which satisfies a (set of) formulas, and an interpretation consists of a structure and an assignment.

Definition 2.1.2 (Interpretation)**.** An interpretation I of a first-order language $\mathcal{L}$ is a pair (S, E), where S is a structure of $\mathcal{L}$ and E is the environment; an assignment that maps variables to elements of the domain of S.

Definition 2.1.3 (Satisfiability)**.** A formula ϕ is satisfiable in a structure S, if there exists an environment E such that $(S, E) \models \phi$, that is, that the formula is interpreted as true.

Definition 2.1.4 (Formula is true)**.** A formula is true in a structure S if, for every assignment E of the environment, the interpretation $I = (S, E)$ satisfies the formula. Denoted as $S \models \phi$.

Definition 2.1.5 (Validity)**.** A formula ϕ is valid (denoted as $\models \phi$) if and only if it is true in every structure. A valid formula is also called a tautology.

It is also possible to compare two formulas by their truth values in different interpretations, which leads to the concepts of equivalence and equisatisfiability.

Definition 2.1.6 (Equivalence of formulas)**.** Two formulas ϕ and ψ are equivalent (denoted as $\phi \equiv \psi$) if and only if they are true in the same interpretations, i.e., for every structure S, it holds that $S \models \phi$ if and only if $S \models \psi$.

Definition 2.1.7 (Equisatisfiability)**.** Two formulas ϕ and ψ are equisatisfiable if ϕ is satisfiable if and only if ψ is satisfiable. In other words, there exists a structure S such that $S \models \phi$ if and only if there exists a structure S' such that $S' \models \psi$.

There is an important distinction between equivalence and equisatisfiability, since equisatisfiable formulas may have different structures in which they are true, whereas equivalent formulas must be true in the same structures. This matters for transformations that introduce new symbols not present in the original formula, because the original and transformed formula cannot share the same interpretation, and thus cannot be equivalent, but they can be equisatisfiable.

Plain first-order logic can be used to express a wide variety of statements, but it is often desirable to work with a theory $\mathcal{T}$, i.e., an additional set of axioms that is assumed to be valid. For example, such a theory can be the theory of equality, the natural numbers or arrays. Equality is an elementary theory that underlies many other theories and serves as the foundation for Vampire. At first glance, adding a theory to the base calculus may seem unnecessary, since it appears equivalent to adding the corresponding axioms to the input. However, because the theory is fixed, the calculus can use specific inference rules designed for that theory, which can greatly speed up proof search. Later in this section, the superposition calculus is described, where three of the five inference rules are specifically designed for the theory of equality [NR01].

Definition 2.1.8 (Validity of a formula with respect to a theory)**.** A formula ϕ is valid with respect to a theory $\mathcal{T}$, consisting of the axioms $\mathcal{A}$, if and only if for every interpretation I such that $I \models \mathcal{A}$, it holds that $I \models \phi$.

Definition 2.1.9 (First-order logic with equality)**.** FOL with equality includes the binary predicate symbol $=$ that represents equality. This predicate is interpreted as the equality relation, which has the following axioms:

$$\begin{aligned}
\forall x.x = x & \quad \text{(reflexivity)} \\
\forall x \forall y.x = y \to y = x & \quad \text{(symmetry)} \\
\forall x \forall y \forall z.x = y \wedge y = z \to x = z & \quad \text{(transitivity)} \\
\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n.(x_1 = y_1 \wedge \cdots \wedge x_n = y_n) & \\
\to f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) & \quad \text{(congruence for function symbols)} \\
\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n.(x_1 = y_1 \wedge \cdots \wedge x_n = y_n) & \\
\to P(x_1, \ldots, x_n) \leftrightarrow P(y_1, \ldots, y_n) & \quad \text{(congruence for predicate symbols)}
\end{aligned}$$

Soundness & Completeness The aim of Vampire is to find a proof of validity for a given conjecture ϕ from a set of axioms Γ, or to show that such a proof cannot exist. In other words, Vampire tries to prove that $\Gamma \models \phi$ holds. This is a semantic property, but automatic reasoners operate on a syntactic level: they manipulate formulas and terms without direct reference to their semantics. The rules used for manipulating such formulas and terms are called inference rules of the calculus. To ensure that such syntactic manipulation establishes the desired semantic property, the calculus must be sound.

Definition 2.1.10 (Soundness)**.** A calculus is sound if and only if for every formula ϕ that can be proven from a set of axioms Γ using the inference rules of the calculus (denoted as $\Gamma \vdash \phi$), it holds that $\Gamma \models \phi$.

Furthermore, it is desirable that a calculus is complete. This means that if a formula is valid, then it can always be proven using the inference rules of the calculus.

Definition 2.1.11 (Completeness)**.** A calculus is complete if and only if for every formula ϕ that is valid with respect to a set of axioms Γ ($\Gamma \models \phi$), it holds that $\Gamma \vdash \phi$.

Now that the most important semantic properties of formulas, together with the connection between syntax and semantics provided by soundness and completeness, have been defined, the focus can shift to the purely syntactic manipulation of formulas performed by automated theorem provers.

Normal Forms First, common names for fragments of formulas are introduced, as they are needed to define normal forms. Then the normal forms used by Vampire for proof search are defined.

Definition 2.1.12 (Atomic formula)**.** An atomic formula is a predicate symbol applied to terms, i.e., a formula of the form $P(t_1, \ldots, t_n)$ where P is a predicate symbol and $t_1, \ldots, t_n$ are terms.

Definition 2.1.13 (Literal)**.** A literal L is an atomic formula (i.e., a predicate applied to terms) or its negation.

Definition 2.1.14 (Clause)**.** A clause C is a disjunction of literals, so a formula of the form $L_1 \vee \cdots \vee L_n$ where $L_1, \ldots, L_n$ are literals.

It is useful to define normal forms of formulas, which impose specific syntactic restrictions on their shape. These normal forms are important for proof search in Vampire because they simplify the structure of formulas and make it easier to find suitable inference rules. It is always possible to transform a formula into these normal forms. A proof which shows that any formula can be transformed into conjunctive normal form (CNF) can be found in [Fit96]. Similar proofs can be found for the other normal forms as well, but they are not included here for brevity.

Definition 2.1.15 (Negation normal form)**.** A formula ϕ is in negation normal form (NNF) if and only if the negation symbol only appears directly in front of atomic formulas.

The CNF is probably the best-known normal form, as it is used by SAT solvers in propositional logic and by superposition-based theorem provers alike and is a starting point for many other proofs.

Definition 2.1.16 (Conjunctive normal form)**.** A formula ϕ is in CNF if and only if it is a conjunction of clauses, where a clause is a disjunction of literals, and a literal is an atomic formula or its negation. So the formula is in CNF if and only if it is of the form

$$\phi = \bigwedge_{i=0}^{n} C_i = \bigwedge_{i=0}^{n} \bigvee_{j=0}^{m_i} L_{i,j}$$

where each C_i is a clause and $L_{i,j}$ a literal.

Within this thesis, the abbreviation CNF will be conflated with the *clause* normal form, largely used by Vampire. One can interpret clauses as sets of literals, and a formula as a set of clauses, which is the common representation used in superposition-based theorem provers. This treats the associativity and commutativity implicitly, which is convenient for proof search, but makes it harder to reconstruct the original formula from the generated proof.

While processing, Vampire treats all formulas to be implicitly quantified by universal quantification, which is related to the prenex normal form (for universal quantifiers):

Definition 2.1.17 (Prenex normal form)**.** A formula ϕ is in prenex normal form if and only if it is of the form $Q_1 x_1 \ldots Q_n x_n \psi$, where $Q_1, \ldots, Q_n$ are quantifiers (i.e., $\forall$ or $\exists$) and ψ is a quantifier-free formula.

Syntactic Manipulations An important property, used a lot within the thesis in the form of rewriting, is that formulas can be manipulated by replacing subformulas with equivalent formulas. It is also a foundation of the inference rules of the superposition calculus, which rely on replacing subterms with equal terms. It is a consequence of the replacement theorem:

Theorem 2.1.1 (Replacement theorem)**.** Let $\phi[X]$ be a formula containing a subformula X and S a structure (for the respective language). Let Y be a formula in the same language as $\phi[X]$ and X. If $X \equiv Y$ is true in S, then $\phi[X] \equiv \phi[Y]$ is true in S.

Proof. Theorem 8.2.1 in [Fit96]. □

Furthermore, to manipulate formulas, substitutions can be used, which instantiate variables with terms.

Definition 2.1.18 (Term substitution)**.** A substitution σ is a mapping from variables to terms, which can be applied to a term t to obtain a new term $t\sigma$ by replacing each variable x in t with $\sigma(x)$.

Sometimes, two substitutions need to be applied in sequence, which can be expressed by the composition of substitutions. Special care is needed when defining substitution composition, since the order of application matters.

Definition 2.1.19 (Composition of substitutions)**.** The composition of two substitutions σ and θ is a substitution $\sigma\theta$ with the following property: For every term t, it holds that $t(\sigma\theta) = (t\sigma)\theta$.

The definition of substitutions for terms can simply be extended to formulas by applying the substitution to each term in the formula.

These substitutions can be used (e.g. within the superposition calculus) to make two terms syntactically equal so that they can then be rewritten using the replacement theorem. A substitution, which makes two terms syntactically equal, is called a unifier of the two terms.

Definition 2.1.20 (Unifier)**.** A unifier of two terms t_1 and t_2 is a substitution σ such that $t_1\sigma = t_2\sigma$.

There may be multiple different unifiers for two terms, but there exists a special unifier, called the most general unifier. It is the most general substitution that unifies the two terms. The meaning of most general is that any other unifier can be obtained from the most general unifier by applying a further substitution. The most general unifier (mgu) is unique up to variable renaming [Fit96].

Definition 2.1.21 (Most general unifier)**.** A unifier σ of two terms t_1 and t_2 is a mgu if and only if for every unifier σ' of t_1 and t_2, there exists a substitution θ such that $\sigma' = \theta\sigma$.

Automated theorem provers rely on the most general unifiers since they allow to get the most general inference rules. Keeping only the most general forms of formulas is useful because it keeps the search space smaller, thereby saving memory and time.

Subsumption Another concept that is important for keeping the search space small is subsumption, a relation between clauses that can be used to decide which clauses are redundant and can be removed from the search space without losing completeness of the calculus.

Definition 2.1.22 (Subsumption)**.** A clause C subsumes a clause D if and only if there exists a substitution σ such that $C\sigma \subseteq D$, where $C\sigma$ is the set of literals obtained by applying the substitution σ to each literal in C.

2.2 Superposition Calculus; or, How Vampire Finds Proofs

The superposition calculus is a sound and refutationally complete calculus for first-order logic with equality [BG94]. Refutational completeness means that if a set of formulas is unsatisfiable, then the system can derive a contradiction (i.e., the empty clause) from it. This is sufficient for proving validity, since a conjecture ϕ is valid with respect to a set of axioms Γ if and only if the set of formulas $\Gamma \cup \{\neg\phi\}$ is unsatisfiable.

2.2.1 Saturation Loop

In Vampire, proof search revolves around a saturation loop, which applies inference rules to derive new clauses from the existing ones. The saturation loop works on a set of clauses S, which represents the current state of the proof search.

To show that a conjecture ϕ is a logical consequence of a set of axioms Γ in the refutational saturation process, the conjecture is negated and added to the set of formulas $S = \Gamma \cup \{\neg\phi\}$. [2] After preprocessing (converting all first-order formulas to CNF resulting in a set of clauses), inference rules of the superposition calculus are applied to derive new clauses from the clauses in S. These inference rules will be described in more detail later.

The saturation loop selects how inference rules are applied to the existing clauses in S, typically maintaining a set of active clauses and a set of passive clauses. Active clauses

[2] Notice how this is a set of formulas instead of a single formula with logical conjunction. This point of view will be important later on, since the set has no ordering and thus implicitly uses associativity and commutativity of $\wedge$.

are those that are used for generating inferences, while passive clauses are those that have not yet been selected [KV13]. Various strategies exist for selecting the clauses to process such as the Otter loop [Mcc03], the Discount loop [DKS97] as well as a limited resource strategy [RV03] to name some. The saturation loop derives new clauses until either the empty clause can be derived, or no further non-redundant clauses can be derived by the available inference rules.

The conjecture is implied if the empty clause, $\square$, can be derived from S using the inference rules of the superposition calculus.[3] The case where the empty clause can be derived is also the only one that is considered for proof-checking in this work. For a complete description, although not treated by this work, the other cases also need to be considered.

If the empty clause cannot be derived, and no further non-redundant clauses can be derived by the available inference rules, *saturation* has been reached [BG94]. If the calculus is complete,[4] saturation means that a model of S can be constructed, which shows that the input clauses are consistent. Therefore, the conjecture is not valid with respect to the axioms and a counterexample can be constructed.

As a final option, the saturation process could run indefinitely, which would also show that the input clauses are satisfiable and thus that the conjecture is not valid with respect to the axioms. However, in practice, this never occurs because computational resources are finite, so the process will be stopped after a certain time limit or memory limit is reached. If this is the case, no conclusion can be drawn about the validity of the conjecture with respect to the axioms.

The superposition calculus is proven to be both refutationally sound and complete [BG94], so all of these cases yield correct conclusions about the validity of the conjecture with respect to the axioms. In practice, however, such guarantees may not hold for an implementation of the calculus in a given ATP. Bugs may be present in the implementation, potentially causing both unsoundness and incompleteness. This is the main motivation for this work: checking proofs for soundness.

2.2.2 Inference Rules

The inference rules in Figure 2.1 are the main rules used in the superposition calculus implemented in Vampire. Resolution and factoring are rules from the resolution calculus for first-order logic without equality.

The other three rules all deal with equality, where equality resolution and equality factoring are specialized versions of the former rules. Superposition is the main inference rule for reasoning about equality, since it allows the term s' occurring in the literal L to be replaced by another equal term t. In essence, this allows terms to be rewritten using equal terms [BG94; NR01].

[3] Requiring the soundness of the calculus.

[4] The theoretical superposition calculus is refutationally complete [BG94].

Resolution

$$\frac{\underline{L} \vee C \qquad \underline{\neg L'} \vee D}{(C \vee D)\sigma}\ \sigma = \mathrm{mgu}(L, L')$$

Factoring

$$\frac{\underline{L} \vee \underline{L'} \vee C}{(C)\sigma}\ \sigma = \mathrm{mgu}(L, L')$$

Superposition

$$\frac{\underline{s = t} \vee C \qquad L[s'] \vee D}{(\underline{L[t]} \vee C \vee D)\sigma}\ \sigma = \mathrm{mgu}(s, s')$$

Equality Resolution

$$\frac{\underline{s \neq t} \vee C}{C\sigma}\ \sigma = \mathrm{mgu}(s, t)$$

Equality Factoring

$$\frac{\underline{s = t} \vee \underline{s' = t'} \vee C}{(s = t \vee t \neq t' \vee C)\sigma}\ \sigma = \mathrm{mgu}(s, s')$$

Figure 2.1: Inference rules of the superposition calculus. L and L' are literals, C and D are clauses, s, t, s', and t' are terms, and σ is a mgu. The underlined literals are selected by the selection function, which is very important for the effectiveness of proof search in Vampire, in addition to subsumption conditions. However, these conditions are not relevant for proof-checking in this work, as they merely restrict when the rule is applied and do not affect its soundness.

In addition to these base rules, Vampire uses further rules that are not necessarily required by the calculus. These additional rules speed up proof search because they are more specific and simplify the set of formulas by removing formulas that are less general, i.e., formulas that are subsumed by newly generated, more general formulas. These rules are called simplification rules. One of the most important simplification rules is the forward demodulation rule [GKR20].

$$\frac{\underline{s = t} \qquad \cancel{\underline{L[s']} \vee D}}{(L[t] \vee D)\sigma}\ \sigma = \mathrm{mgu}(s, s')$$

It has the same form as the superposition rule, but it is only applied if the left premise is a unit clause, i.e., a clause with only one equality literal. Additionally, the right premise needs to be subsumed by the generated clause, effectively deleting the old clause and replacing it with the new one.

2.2.3 Further Inferences and Theory Reasoning

Vampire also uses various other inference rules, next to the main inference rules of the superposition calculus. Most relevant for this work are the inference rules that transform formulas into normal forms, which are used for proof search since the inference rules of

the superposition calculus are designed for clauses. Relevant inference steps will be described in more detail in Chapter 3.

Furthermore, Vampire also has specific inference rules for reasoning in theories, such as the theory of arrays, integer arithmetic, reals, and others [Bár+25]. Checking these inference rules is not part of this work, as the focus is on first-order logic with equality without additional theories.

2.3 Dependent Type Theory; or, How Lean Checks Proofs

Many modern interactive theorem provers, including Rocq [Tea24], Agda [BDN09] and Lean [MU21] use dependent type theory as their underlying formalism [Car19; Tea24; BDN09]. The basis of dependent type theory is the untyped lambda calculus, extended by types. Different variants of the lambda calculus add types in different ways, which leads to type theories that can be classified by the lambda cube introduced in [Bar91]. They differ in their expressiveness and in the permitted dependency of types on terms and other types. The most expressive type theory in the lambda cube is the calculus of constructions, which allows both types depending on terms and types depending on other types. Rocq and Lean are both based on the calculus of constructions, with inductive definitions and further extensions [Tea24; Car19].

In the sequel, a somewhat informal and incomplete introduction to dependent type theory is given, which should be sufficient to understand the methods used in this work. For a more detailed introduction, the reader is referred to [Rij25].

Syntactic reasoning within dependent type theory is performed by inference rules, which are used to derive judgements about the types of terms and the validity of types.

A basic reasoning rule, which allows to deduce the type of a term from an already derived equality judgement, is for example the following rule:

$$\frac{\Gamma \vdash a \doteq b : A}{\Gamma \vdash a : A}$$

The assumption of this rule is a judgement of equality between two terms a and b of type A in the context Γ. Dissecting this symbol by symbol, Γ is the context, which is a set of assumptions about the types of variables. In this example, it does not play an important role and is simply carried along in the inference rules. The symbol $\vdash$ separates the assumptions on the left from the conclusion on the right. The symbol $\doteq$ represents judgmental equality. Finally, the symbol $:$ represents the typing relation, which is used to express that a term has a certain type. So the complete inference rule can be expressed in natural language as follows: if we have equality between two terms a and b of type A in the context Γ, then we can conclude that a has type A in the context Γ.

Reasoning in dependent type theory requires several such inference rules which allow deriving the type of a term or to derive that a type is valid. For Lean, these rules are described in the work [Car19], which gives a description of the type theory underlying Lean.

2.3.1 Special Types

Additional rules are used to specify the behavior of terms of special types, such as the Π type, the Σ type and inductive types.

Π Type: Dependent Function Types The Π type represents functions that take arguments of a certain type and return values of another type that can depend on the argument.

An example using dependent function types is the following definition of the concatenation function for polymorphic vectors:

$$\texttt{concat} : \Pi_{A:\mathcal{U}}\Pi_{n:\mathbb{N}}\Pi_{m:\mathbb{N}}\text{Vec } A\ n \to \text{Vec } A\ m \to \text{Vec } A\ (n+m)$$

It takes two vectors of lengths n and m and returns a vector whose length is the sum of the lengths of the input vectors. The length of the vector is directly encoded in its type, which is a defining feature of dependent type theory. Non-dependent functions that do not have a return type depending on the argument type can also be expressed in dependent type theory by using the Π type; however, the arrow symbol $\to$ is usually used to represent these functions.

Σ Type: Dependent Pair Types The Σ type allows one to express dependent pair types, i.e., pairs of values where the type of the second value can depend on the first value. An example of a dependent pair type is the following definition of the type of a list:

$$\text{List} : \Sigma_{n:\mathbb{N}}\text{Vec } A\ n$$

which is a dependent pair type consisting of a natural number n and a vector of length n. This type can be used to express lists of any length, or as a return type for a function that returns vectors of any length. The non-dependent version of the Σ type is written as the product type $\times$, which is the special case where the second type does not depend on the first type.

Inductive Types Another important building block of the dependent type theory used in Lean are inductive types, which allow types to be defined by specifying their constructors, induction principles (or recursors), and computation rules [Car19; Rij25]. For example, the natural numbers can be defined as an inductive type with two constructors: zero and successor.

Type definition:

$$\frac{}{\vdash \mathbb{N}\ \text{Type}}$$

Constructor definitions[5]:

$$\frac{}{\vdash 0_{\mathbb{N}} : \mathbb{N}}$$

$$\frac{}{\vdash \text{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}$$

These two constructors already allow construction of any natural number by applying the successor constructor to zero a certain number of times. To reason about them, they are also equipped with an induction principle, which allows one to reason about the elements of the inductive type. For the natural numbers, the induction principle has two cases: one for the base case of zero and one for the inductive case of successor. The following rule is the induction principle for natural numbers which, given that $P(n)$ is a proper type as well as the base case p_0 and the inductive case p_{succ}, gives us a function $\text{ind}_{\mathbb{N}}$ that takes a natural number n and returns an element of $P(n)$.

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n)\ \text{Type} \qquad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \qquad \Gamma \vdash p_{\text{succ}} : \Pi n : \mathbb{N}.P(n) \to P(\text{succ}_{\mathbb{N}}(n))}{\Gamma \vdash \text{ind}_{\mathbb{N}}(p_0, p_{\text{succ}}) : \Pi n : \mathbb{N}.P(n)}$$

It allows us to conclude a property of all natural numbers by showing the property just for the constructors. For natural numbers, this corresponds to the usual induction principle. To apply a term obtained from the induction principle, suitable computation rules (that define what happens when the function $\text{ind}_{\mathbb{N}}(p_0, p_{\text{succ}})$ is applied to an element of $\mathbb{N}$) are also introduced.

More generally, inductive types are always defined by a set of constructors used to build the elements of the type, some form of induction principle that allows one to reason about the elements of the inductive type, and computation rules that allow the induction principle to be used for computation with those elements [Rij25].

Empty Type and Unit Type Two special inductive types are the empty type and the unit type. The empty type has no constructors, so it has no elements, while the unit type has one constructor of the unit type with suitable inductive principles. It has exactly one element [Rij25].

Coproduct Type Another useful inductive type is the coproduct type $A + B$, which represents the disjoint union of two types A and B. It can be used to express a type that can be either of type A or of type B. To construct it, two constructors are required, one for each type, which take an element of the respective type and return an element of the coproduct type [Rij25]:

$$\frac{\Gamma \vdash A\ \text{Type} \qquad \Gamma \vdash B\ \text{Type}}{\Gamma \vdash \text{inl} : A \to A + B}$$

[5]Notice how this is the first time we make a judgement of the type of a term, before only judgements over types were made.

$$\frac{\Gamma \vdash A \text{ Type} \qquad \Gamma \vdash B \text{ Type}}{\Gamma \vdash \text{inr} : B \to A + B}$$

This means that if one has an element of type $A + B$, then it contains either an element of type A or an element of type B. The inductive principle for the coproduct type recombines the two cases by requiring two functions, one taking an element of type A and the other taking an element of type B, and returning an element of some type P [Rij25]:

$$\text{ind}_+ : (\Pi_{x:A} P(\text{inl}(x))) \to (\Pi_{y:B} P(\text{inr}(y))) \to \Pi_{z:A+B} P(z)$$

One function takes an element of type A and returns an element of some type P, while the other function takes an element of type B and returns an element of the type P (with P possibly dependent on the given element). When both functions are provided, the induction principle yields a dependent function consuming the coproduct type and computing as expected.

2.3.2 Universes

Previous definitions used the symbol $\mathcal{U}$ without explanation. This symbol represents a universe, which can informally be thought of as a type of types. More formally, a universe $\mathcal{U}$ contains an encoding of types as elements of the universe, and a type family $\mathcal{T}$ that maps elements of the universe to types. Universes need to satisfy further properties, such as closure under Π. A more rigorous definition can be found in [Rij25].

Multiple Universes Universes are used in some type theories to avoid paradoxes such as Girard's paradox [Gir72] (similar to Russell's paradox in set theory). These paradoxes typically arise from self-reference, which in type theory occurs when types are allowed to be elements of themselves. To avoid this paradox, multiple universes are used to form a hierarchy [Rij25]. These hierarchies can have different features depending on the type theory. For example, Rocq and Lean use different approaches: Rocq has a cumulative hierarchy of universes, while Lean does not use cumulativity but instead features explicit universe polymorphism for types [Car19]. Furthermore, the lowest level of the hierarchy typically has a special role and name, namely the universe of propositions. In both Lean and Rocq, this universe is called Prop. In Lean it has several special properties, such as impredicativity and proof irrelevance, which are not present in the other universes. For most of this work, the details of the universe hierarchy are not relevant, but in Section 2.4 it will be important when converting between propositions and booleans in Lean, which are in two different universes.

2.3.3 Curry–Howard Correspondence

Until now, the description of dependent type theory has not yet explored the connection between dependent type theory and proof-checking. The Curry–Howard correspondence bridges this gap by relating types to propositions and terms to proofs, and vice versa. To show that a theorem is true in a proof assistant based on dependent type theory, one

formalizes the statement of the theorem as a type. To prove the theorem, it suffices to construct a term of that type, which serves as the proof.

All of the previously introduced types can be used to map between first-order logic and dependent type theory, where a table of such a mapping is given in Table 2.1.

In fact, dependent type theory is more expressive than FOL, but for the purpose of this work, only the correspondence between FOL and dependent type theory is relevant.

First-order Logic	**Dependent Type Theory**
Proposition	Type
Proof	Term
Predicate	Type family
Functions	Functions
$\top$	Unit type
$\bot$	Empty type $\emptyset$
$\wedge$	Product type (non-dependent) $A \times B$
$\vee$	Coproduct type (non-dependent) $A + B$
$\Rightarrow$	Function type (non-dependent) $A \to B$
$\neg P$	$P \to \emptyset$
$\forall x, P(x)$	Dependent function type $\Pi_{x:A} P(x)$
$\exists x, P(x)$	Dependent pair type $\Sigma_{x:A} P(x)$
$x = y$	Propositional equality $x = y$

Table 2.1: Curry–Howard correspondence of first-order logic in dependent type theory [Rij25].

The interpretation of logic in dependent type theory is largely captured by the table, but there are some differences to note.

Constructive Logic The Curry–Howard correspondence is of constructive nature, which means that it does not assume the law of the excluded middle ($a \vee \neg a$ is valid) or the axiom of choice (used in classical logic). As a result, it limits the types of proofs that can be expressed to constructive proofs, which are a subset of the proofs that are valid in classical logic. Notably, proof by contradiction is not derivable in constructive logic without additional axioms. While some parts of mathematics are developed only constructively, many mathematical proofs rely on proof by contradiction, double negation elimination, or the axiom of choice, which is also the case for all proofs by Vampire. Every (refutational) proof by Vampire is a proof by contradiction. As shown in recent work [ŠR26], some proofs found by Vampire may be transformed to constructive proofs, but this is not always possible.[6]

To bridge this gap, axioms can be added to dependent type theory to make it classical. For example, in Lean, propositional extensionality, a law over quotient types related to

[6]Transformation for the fragment of FOL with equality "equational Horn" is shown in the paper

functional extensionality, and the axiom of choice are added as axioms [Car19]. These axioms allow to derive the law of the excluded middle and double negation elimination, making it possible to express classical proofs in Lean.

Proof Irrelevance In classical logic, the specific proof of a theorem does not matter, which is called proof irrelevance. In dependent type theory, this relationship is more complex, since proofs are terms. Different proofs correspond to different terms with the same type, thereby being able to distinguish different proofs of the same theorem. This would mean that, for two different proof terms p_1 and p_2 of the same type P, one cannot (immediately) assert $p_1 = p_2$. This contrasts with the idea of proof irrelevance in classical logic.

When considering classical logic, multiple proofs of the same theorem are also possible, but the proofs are normally not considered part of the logic itself, whereas in dependent type theory they are which is the cause of this difference. A more detailed view of interpreting logic in dependent type theory is given in [Rij25]; for this work, it suffices to note that interactive theorem provers have carefully designed their inference systems to ensure that the embedding works as expected [Car19].

2.3.4 Example of the Embedding in Lean

Finally, to show that the implementation of the embedding in Lean is as expected, the definition of the disjunction operator $\vee$ is taken as an example in Listing 2.1. As shown in Table 2.1, the disjunction operator $\vee$ is constructed by a coproduct type $A + B$ in dependent type theory, which is defined as an inductive type with two constructors `inl` and `inr` and an induction principle `Or.elim`. It also shows how closely related the Lean code is to the mathematical definitions, which is a main feature of Lean.

```
1  inductive Or (a b : Prop) : Prop where
2    | inl (h : a) : Or a b
3    | inr (h : b) : Or a b
4
5  theorem Or.elim {c : Prop} (h : Or a b) (left : a → c) (right : b → c) : c :=
6    match h with
7    | Or.inl h => left h
8    | Or.inr h => right h
```

Listing 2.1: Main elements of the definition of "Or" ($\vee$) in Lean Core. `inl` and `inr` are the constructors and `Or.elim` is the induction principle for this inductively defined type.

2.4 AVATAR: Advanced Vampire Architecture for Theories and Resolution

AVATAR is an architecture for reasoning about the propositional structure of formulas, as well as theory reasoning in the superposition calculus [Vor14; Bár+25]. It allows Vampire to reason about the propositional structure of formulas, which helps to speed up proof search by allowing the use of propositional reasoning techniques employed by SAT solvers. A full description of AVATAR is beyond the scope of this work and can be found in the introductory paper [Vor14]. The relevant details of AVATAR for this work are briefly discussed next.

2.4.1 Splitting of Clauses

To reason about the propositional structure of formulas, AVATAR splits clauses into components and gives them names. At first, we introduce splitting:

Given a clause $C = \forall x_1, \ldots, x_n, y_1, \ldots, y_m. (C_1[x_1, \ldots, x_n] \vee C_2[y_1, \ldots, y_m])$, where $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ are disjoint sets of variables, one can split the quantifier as follows:

$$\forall x_1, \ldots, x_n. C_1[x_1, \ldots, x_n] \vee \forall y_1, \ldots, y_m. C_2[y_1, \ldots, y_m]$$

Let S be a set of clauses, then we have: $S \cup C$ is unsatisfiable, if and only if $S \cup C_1$ and $S \cup C_2$ are unsatisfiable [Vor14]. Hence, the original clause C can be replaced by the two clauses C_1 and C_2 without affecting the unsatisfiability of the set of clauses. This can be generalized to splitting a clause into n clauses. In the context of AVATAR, these split clauses are called *components* and are identified with propositional variables.

To keep track of the names and meanings of the newly found components, *assertions* and *A-clauses* are introduced. An assertion is a finite set of propositional variables. An A-clause is a pair consisting of a clause and an assertion. An A-clause is denoted as follows:

$$D \leftarrow A$$

where D is the clause and A is the assertion. The assertion A is a set of propositional variables. It uses the $\leftarrow$ symbol as it is essentially an implication, reading as: if the assertion A holds, then the clause D holds.

2.4.2 Constructing A-Clauses

To make use of A-clauses and splitting, Vampire splits clauses into components and then constructs A-clauses from the components. For example, given a clause C and its components C_1 and C_2, the following A-clauses are constructed:

$$C_1 \leftarrow \{sA_1\} \quad C_2 \leftarrow \{sA_2\}$$

where sA_1 and sA_2 are two new propositional assertions corresponding to the components C_1 and C_2, respectively. Since sA_1 and sA_2 are now simple propositional variables,

they can be used in propositional reasoning, simplifying the first-order structure of the components and only keeping the propositional structure in the form of the assertions.

2.4.3 First-Order Reasoning over A-Clauses

The previously introduced saturation rules can now be modified to work with A-clauses:

$$\frac{D_1 \quad \dots \quad D_n}{D} \quad \rightarrow \quad \frac{D_1 \leftarrow A_1 \quad \dots \quad D_n \leftarrow A_n}{D \leftarrow A_1 \cup \dots \cup A_n}$$

These new inference rules carry along the assertions for the premises, but otherwise stay the same as the original calculus.

2.4.4 SAT Solver and First-Order Reasoning Combined

To reason about a problem using AVATAR, the SAT solver and the first-order reasoning of Vampire are combined to work together. In essence, Vampire tries to split clauses into components, names them, and passes the propositional formulas corresponding to the components to the SAT solver.

The SAT solver either finds a satisfying assignment for the propositional formula, or shows that it is unsatisfiable. If the formula is unsatisfiable, a refutation is found and proof search is finished. If the formula is satisfiable (i.e., there exists a satisfying assignment) then A-clauses (relevant for the satisfying assignment) are added to the saturation set, and the first-order reasoning of Vampire is used to derive new A-clauses, whose corresponding propositional assertions are then again passed to the SAT solver. This process is repeated until either a refutation is found, or saturation is reached.

Special care needs to be taken to properly handle simplification and subsumption of A-clauses, which is covered in more detail in the original paper on AVATAR [Vor14]. In conclusion, AVATAR introduces modifications and additional rules to standard Vampire proofs:

- modified A-clause inferences
- propositional variable declarations and definitions
- splitting of clauses into components
- SAT refutations

CHAPTER 3

Vampire to Lean Proof Translation

This chapter describes the methods used to translate proofs found by Vampire into suitable Lean proof objects. First, it presents a general system overview of the architecture and implementation. Then, it describes reconstruction steps in Vampire that recover details discarded during proof search for performance reasons but required for proof-checking. To simplify proof-checking, proof search (in particular preprocessing) was also modified to use transformations that are more suitable for reconstruction. Finally, the translation and reconstruction procedures in Lean are described, which are used to show the validity of Vampire inference steps in Lean.

3.1 Trust Model

The main goal of this work is to increase the trust in the soundness of Vampire proofs. Therefore, it is crucial to understand what one needs to be able to trust the checked proofs. Crucially for this work, *no* trust[1] is required in both Vampire and the proof output it produced because it is checked independently of Vampire starting from the axioms to the conclusion of the conjecture.

Only one software component needs to be trusted, which is the Lean ITP, and more specifically, the kernel of Lean, which is the part of the system responsible for checking the validity of proof terms. When a proof using AVATAR is checked, the trust needs to be extended to the (Lean) verified SAT proof checker employed by the tactic `bv_decide`, and thus essentially the Lean compiler [Böv+25].

[1] up to the straightforward translation from input syntax to output syntax and the preservation of semantic meaning

Trusted Components A list of components that need to be trusted to establish soundness of the checked proof:

- Lean proof assistant
- Translation of assumptions and conclusion from input syntax to output syntax
- Equivalence of the semantics between input and Lean output

The translation from input (SMT-LIB or TPTP) to Lean syntax may lead to a semantic difference, depending on the definitions used for symbols in the translation procedure. However, the correspondence for first-order logic in dependent type theory is well-established and uses only basic theorems in Lean. Nevertheless, a bug in the translation procedure may lead to proving something different than what was originally intended.

Hence, if a proof is successfully checked by Lean, a step to further increase trust is to verify that the final theorem proven in Lean, which is the translation of the original conjecture, is semantically equivalent to the original conjecture. For the fragment of first-order logic with equality, this is not expected to be a problem as the corresponding definitions are simple and well-established, but if this approach is extended in the future (with more complex theories, for example), then a semantic mismatch between the original conjecture and the final theorem in Lean may arise due to different interpretations of concepts.

If the translation from input syntax to Lean syntax was found to be satisfactory, additional steps could be taken to further increase trust in the result, such as checking the proof with tools like `lean4checker` or using a different kernel to check the proof such as `nanoda_lib` [Bai26] to check the proof independently of Lean's kernel.

3.2 System Design

There are two main components in the system: Vampire and Lean, which can both be used to obtain details for the proof found by Vampire. The effort of inferring necessary proof details can be distributed between the two programs, which enables many engineering decisions that can strongly influence proof-checking performance. The guiding philosophy of the implementation in this work is to perform tasks where they are most natural or easiest to carry out.

This means that Vampire almost never handles proof terms directly, but instead outputs instructions to Lean on how to create these terms. This is feasible because Lean already has powerful proof automation features and a wide range of tactics that can be used to construct suitable proof terms. Another design goal is to keep the implementation in Vampire mostly independent of the actual inference rules to keep future maintenance easier. Previous approaches [Raw+25; KRS25] reproduced implementation details in the reconstruction code which created maintenance issues when the implementation of the inference rules in Vampire changed. This was avoided by calling the original

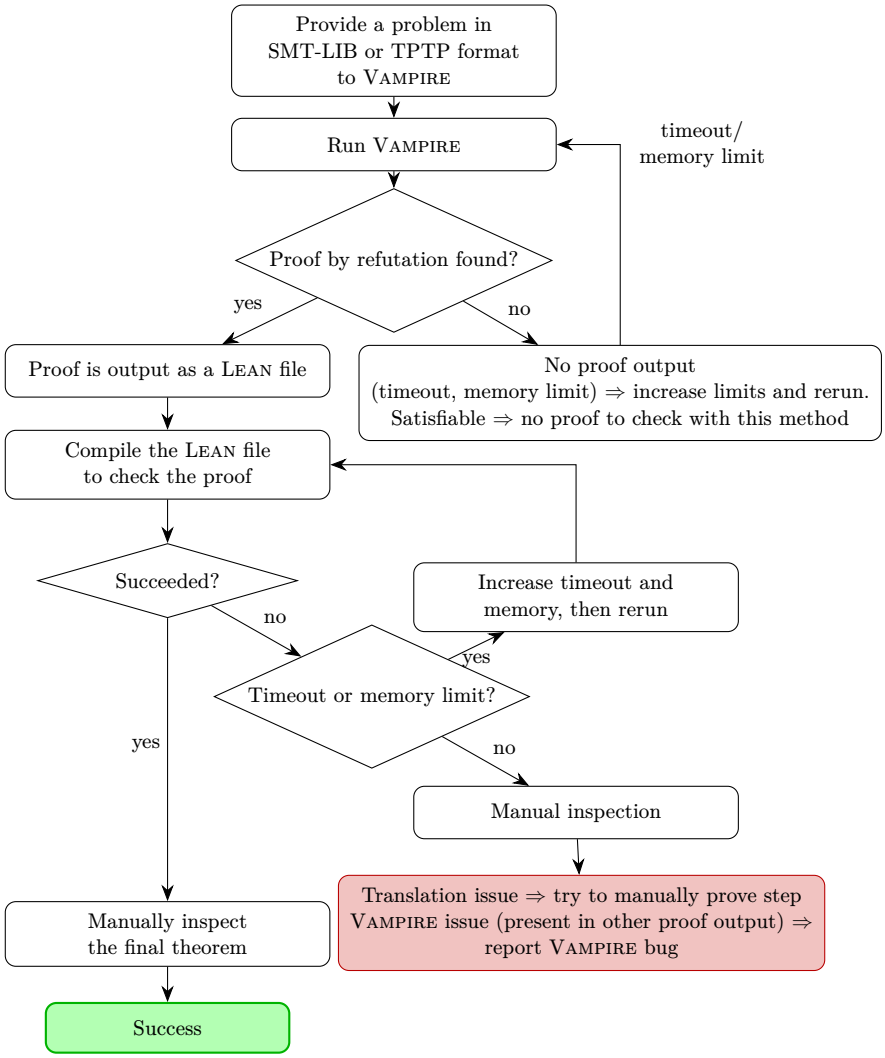


Figure 3.1: Workflow for checking a Vampire proof in Lean.

implementation of the inference rules without modification and then reconstructing the proof details by proof replay, which is described in more detail in Section 3.6.1.

3.2.1 User Workflow

The workflow for checking a proof is shown in Figure 3.1. Typically, a user provides a problem in SMT-LIB or TPTP format to Vampire, which, after successful proof search, outputs a Lean input file. This file can then be checked by Lean (given that the suitable library is imported). To further establish validity, a manual check of the final theorem in the output is useful to ensure that the theorem corresponds to the original assumptions and conjectures from the input problem, which has the same semantics as the original problem given in SMT-LIB or TPTP format.

If something fails in this process, the failure needs to be examined manually. There are mainly two possible reasons:

- *The proof translation and reconstruction are incorrect*: The proof found by Vampire is correct, but there is a problem with the reconstruction. This can be fixed manually by proving the problematic inference step.
- *The proof found by Vampire is incorrect*: The proof is incorrect, which could be caused by a bug in the implementation of Vampire. This can be checked by comparing the Lean proof output with the "normal" proof output of Vampire. Furthermore, another theorem prover could be invoked on the same problem to see if it produces a different result. If there appears to be a real problem, it should be reported to the Vampire developers.

It should be noted that only default Vampire proof search is supported by the implemented method. Vampire has various options, invoking different proof search strategies, which are typically controlled by the portfolio mode of Vampire. These options may trigger unsupported inferences or transformations, which are not yet implemented for proof-checking.

3.2.2 Internal Program Flow

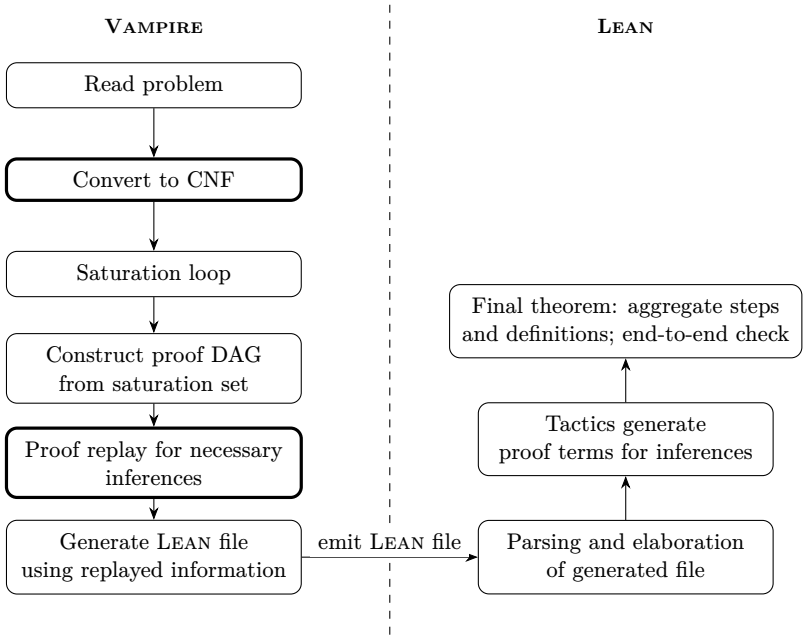


Figure 3.2: High level overview of the processing in both programs: left shows Vampire proof search, right shows Lean proof-checking. The thick boxes indicate where information is recorded in Vampire for proof printing.

A more detailed view of the internal program flow of proof-checking is given in Figure 3.2. Extra information for proof reconstruction is collected in Vampire at several steps during the proof search. While performance would suffer when collecting lots of information during the saturation loop, collecting information for some preprocessing steps is acceptable as it only happens once per problem and is typically not the main bottleneck in proof search. The remaining missing details are reconstructed later by proof replay, described in Section 3.6.1.

3.3 Running Example

To illustrate the applied techniques, a running example is used, which is a puzzle with the following rules:

1. Someone who lives in Dreadbury Mansion killed Aunt Agatha.
2. The only residents are Aunt Agatha, the Butler, and Charles.
3. Everyone hates anyone they kill.
4. No killer is richer than their victim.
5. Charles hates nobody whom Aunt Agatha hates.
6. Aunt Agatha does not hate the Butler.
7. Aunt Agatha hates everyone except the Butler.
8. The Butler hates everyone not richer than Aunt Agatha.
9. The Butler hates everyone Aunt Agatha hates.
10. Nobody hates everyone.
11. Aunt Agatha is not the Butler.

This puzzle can be formalized in first-order logic with equality, where the goal is to prove that Aunt Agatha killed herself. It is the problem `PUZ001` in the TPTP library [Sut24; Pel86]. The used formalization of the problem is given in Listing 3.1.

$$\exists x.(\text{lives}(x) \wedge \text{killed}(x, \text{agatha})) \quad (1)$$

$$\text{lives}(\text{agatha}) \quad (2.1)$$

$$\text{lives}(\text{butler}) \quad (2.2)$$

$$\text{lives}(\text{charles}) \quad (2.3)$$

$$\forall x.(\text{lives}(x) \rightarrow (x = \text{agatha} \vee x = \text{butler} \vee x = \text{charles})) \quad (2.4)$$

$$\forall x, y.(\text{killed}(x, y) \rightarrow \text{hates}(x, y)) \quad (3)$$

$$\forall x, y.(\text{killed}(x, y) \rightarrow \neg\text{richer}(x, y)) \quad (4)$$

$$\forall x.(\text{hates}(\text{agatha}, x) \rightarrow \neg\text{hates}(\text{charles}, x)) \quad (5)$$

$$\forall x.(x \neq \text{butler} \rightarrow \text{hates}(\text{agatha}, x)) \quad (6,7)$$

$$\forall x.(\neg\text{richer}(x, \text{agatha}) \rightarrow \text{hates}(\text{butler}, x)) \quad (8)$$

$$\forall x.(\text{hates}(\text{agatha}, x) \rightarrow \text{hates}(\text{butler}, x)) \quad (9)$$

$$\forall x.(\exists y \neg\text{hates}(x, y)) \quad (10)$$

$$\neg(\text{agatha} = \text{butler}) \quad (11)$$

Listing 3.1: Formalization of the Dreadbury Mansion Puzzle used as the running example.

```
...input...
12. ! [X0] : ? [X1] : ~hates(X0,X1) [input(axiom)]
...preprocessing steps...
28. ! [X0] : ~hates(X0,sK1(X0)) [skolemisation 12,27]
29. killed(sK0,A) [cnf 26]
...cnf transformation...
... resolution steps...
46. ~killed(X0,A) | hates(B,X0) [res 39,36]
48. hates(B,X0) | B = X0 [res 40,38]
53. B = sK0 | C = sK0
     | A = sK0 [res 34,30]
55. (1) <=> A = sK0 [avatar def]
57. A = sK0 <- (1) [avatar component 55]
59. (2) <=> C = sK0 [avatar def]
61. C = sK0 <- (2) [avatar component 59]
63. (3) <=> B = sK0 [avatar def]
65. B = sK0 <- (3) [avatar component 63]
66. (1) | (2) | (3) [avatar split 53,63,59,55]
67. hates(C,A) <- (2) [sup 44,61]
70. hates(B,sK0) [res 46,29]
..resolution steps...
76. A = B <- (2) [res 72,38]
77. $false <- (2) [fwd sub res 76,42]
78. ~2 [avatar contradiction 77]
... superposition and resolution steps ...
... sat refutation steps ...
95. $false [avatar sat refutation s5]
```

Listing 3.2: Excerpt of Vampire proof for the Dreadbury Mansion Puzzle. Several parts of the proof are omitted for brevity, but the main structure is present. It includes preprocessing such as skolemization, resolution and superposition steps, and AVATAR inferences. Constant symbols like Agatha are abbreviated by their initial letters (A,B,C) .

The proof generated by Vampire for this problem is quite long; a shortened version is given in Listing 3.2, which shows important steps of the proof. It is written as a sequence of inferences, where each inference is labeled with a number and justification. The justification consists of the inference rule used (e.g., `res` for resolution). It includes AVATAR inference steps, which is described in more detail in Section 2.4.

3.4 General Structure of the Generated Lean File

The generated Lean file has three sections:

1. **Preamble**: This section contains necessary imports required for the checking of the proof, such as a custom Lean library:
   a) Imports, linting directives

b) Declaration of the (inhabited) type ι, representing the FOL domain

c) Declaration of uninterpreted function symbols (present in the input)

d) Declaration of uninterpreted predicate symbols (present in the input)

e) Declaration of introduced symbols (e.g. Skolem functions, predicate names)

2. **Inferences**: This section contains most inference steps of the proof. Notable exceptions are rules which introduce new symbols (e.g., skolemization and predicate naming) and one part of the CNF transformation for performance reasons. Details about these inferences steps are given in Sections 3.5 and 3.6.

3. **Final Theorem**: This part of the proof is the final theorem which proves the original conjecture. It is formulated as a single theorem that takes (a subset of) the premises in the input problem as assumptions from which the conjecture is derived. If the input file has no conjecture defined, `False` is derived instead, showing that the input problem is unsatisfiable. In total there are four main sections in the proof:

   a) Definition of the problem statement as a Lean theorem

   b) (Arbitrary) definitions for variables that do not appear in the assumptions or the conjecture, but are used in the proof.

   c) Negation of the conjecture if a conjecture is defined in the input (proof by contradiction)

   d) Sequence of combining previously shown inference steps as well as additional definition steps.

3.4.1 Structure of the Generated Lean File for Running Example

Listing 3.3 shows a simplified and shortened version of the previously described full proof file. Each section is marked with comments indicating the start and end of each section. In the preamble, the original constants A, B, C (Agatha, Butler and Charles) are declared as variables of type ι, which is the type representing the FOL domain. Then, the introduced Skolem constants and functions as well as predicate symbols for AVATAR inferences are declared as variables. After these necessary declarations, the second section of the file contains the individual inference steps, which are proven one by one. Finally, the last section contains the full proof of the conjecture that Agatha killed herself, following from the axioms.

In the beginning of the `fullProof` theorem, the assumptions are introduced with the names `step1-step13`, after which the proof proceeds by applying proof by contradiction to the goal. Inferences that introduce new symbols (like skolemization) and AVATAR definitions also show up in this section `step28`, since this is the place where these introduced symbols can be defined and used between multiple inference steps. Then, the previously shown sound inference steps (e.g. `inf_s95`) are combined step by

step. The final step in the proof of `fullProof` is the demonstration that `False` has been derived via `exact step95`, where `step95` is the inference step deriving `False` from its premises.

```
-- preamble --
variable {A B C : ι}
variable {sK1 :  ι → ι}
  {sK0 : ι}
variable {sA1 sA2 sA3 : Prop}
--- end of preamble --
--- start of inferences --
-- step 21 ennf transformation
theorem inf_s21 :
  (∀ v0 : ι, ((hates A v0) → (¬(hates C v0)))) →
  (∀ v0 : ι, ( (¬(hates C v0)) ∨ (¬(hates A v0)))) := by
  -- proof --

-- further inferences --

theorem inf_s95 : (sA1 ∨ sA2 ∨ sA3) → (¬sA2) → (¬sA3) → (¬sA1)
    → False := by
  -- proof --
-- end of inferences --
-- start of final theorem --
theorem fullProof : ... → (killed A A) := by
  intros step1 step5 step6 step7 step8 step9 step10 step11 step12 step13
  apply Classical.byContradiction; intro step15
  -- inferences --
  -- step28 skolemisation
  exists_prenex at step12
  let ⟨sK1,step28'⟩ := step12
  have step28 : (∀ v0 : ι, (¬((hates) v0 ((sK1) v0)))) :=
    by symm_match using step28'
  -- cnf transformation --
  -- step55 avatar definition
  let sA1 := A=sK0
  have step55 : (sA1 ↔ (A=sK0)) := Iff.rfl
  have step57 := inf_s57  step55
  -- more steps --
  have step95 := inf_s95 step66 step78 step88 step94
  exact step95
-- end of final theorem --
```

Listing 3.3: Simplified and shortened Lean code for the entire proof of the Dreadbury Mansion Puzzle.

3.5 Preprocessing

Converting input formulas to CNF (which is represented by a set of clauses in Vampire) is performed in several steps. Only the relevant steps for the fragment of first-order formulas (FOF) formulas with equality, in the default configuration of Vampire will be described here. Depending on the input problem and settings, Vampire may perform additional steps.

1. Simplification of $\top$ and $\bot$
2. Flattening
3. (Optional) Remove unused predicate definitions [2]
4. Conversion to equivalence negation normal form (ENNF)
5. Flattening
6. Predicate Naming
7. Conversion to NNF
8. Flattening
9. Skolemization
10. Conversion to CNF
11. Removal of tautologies

Simplification, flattening, and conversion to ENNF, NNF and CNF can be represented with rewriting rules, which are relatively easy to replicate in Lean. The other steps are either semantic transformations (removal of pure literals and unused predicate definitions) or introduce new symbols (predicate naming and skolemization), which can therefore only be equisatisfiability transformations and need more care to proof in Lean.

3.5.1 Steps Representable as Rewrite Rules

Rewrite rules can represent many of the equivalence transformations conducted by Vampire during preprocessing. A rewrite rule $F \rightsquigarrow G$ means that F is replaced by G in a formula. The rules are chosen such that F and G are logically equivalent and, due to the replacement theorem 2.1.1, the resulting formula is logically equivalent to the original one. Depending on the order and choice where the rewrite rules are applied, the resulting formula can be different syntactically, but it will always be logically equivalent to the original formula. For the purpose of recreating the same formula in Lean from the input formula as Vampire, it is important that the used rewrite rules do not only produce a logically equivalent formula, but also syntactical equivalent. This is easily, achieved when the rewrite rules are confluent. This means that the same result is obtained regardless of the order in which the rules are applied. At least for the rewrite rules for the conversion to NNF, the rules employed by Vampire are not confluent, which means that the order of application needs to be replicated between Vampire and Lean.

[2] This step normally also removes pure literals. However, proof-checking for this step was not developed within this thesis, and an option to disable this transformation was added for this work.

$$
\begin{aligned}
F_1 \wedge \ldots \wedge F_n \wedge (G_1 \wedge \ldots \wedge G_m) &\rightsquigarrow F_1 \wedge \ldots \wedge F_n \wedge G_1 \wedge \ldots \wedge G_m \\
F_1 \vee \ldots \vee F_n \vee (G_1 \vee \ldots \vee G_m) &\rightsquigarrow F_1 \vee \ldots \vee F_n \vee G_1 \vee \ldots \vee G_m \\
\forall x_1, \ldots, x_n \forall y_1, \ldots, y_m F &\rightsquigarrow \forall x_1, \ldots, x_n, y_1, \ldots, y_m F \\
\exists x_1, \ldots, x_n \exists y_1, \ldots, y_m F &\rightsquigarrow \exists x_1, \ldots, x_n, y_1, \ldots, y_m F \\
\neg\neg F &\rightsquigarrow F
\end{aligned}
$$

Figure 3.3: Rewrite rules used by Vampire for the flattening step (making use of associativity of $\wedge$ and $\vee$ as well as simultaneous quantification)[RV18]

$$
\begin{aligned}
\neg(F_1 \wedge \ldots \wedge F_n) &\rightsquigarrow \neg F_1 \vee \ldots \vee \neg F_n \\
\neg(F_1 \vee \ldots \vee F_n) &\rightsquigarrow \neg F_1 \wedge \ldots \wedge \neg F_n \\
F_1 \rightarrow F_2 &\rightsquigarrow \neg F_1 \vee F_2 \\
\neg\neg F &\rightsquigarrow F \\
\neg(F_1 \leftrightarrow F_2) &\rightsquigarrow F_1 \otimes F_2 \\
\neg(F_1 \otimes F_2) &\rightsquigarrow F_1 \leftrightarrow F_2 \\
\neg\forall x_1, \ldots, x_n F &\rightsquigarrow \exists x_1, \ldots, x_n \neg F \\
\neg\exists x_1, \ldots, x_n F &\rightsquigarrow \forall x_1, \ldots, x_n \neg F \\
\neg\top &\rightsquigarrow \bot \\
\neg\bot &\rightsquigarrow \top
\end{aligned}
$$

Figure 3.4: Rewrite rules for conversion to ENNF[RV18]

$$
\begin{aligned}
&\text{All rules from ENNF} + \\
F_1 \leftrightarrow F_2 &\rightsquigarrow (F_1 \rightarrow F_2) \wedge (F_2 \rightarrow F_1); \\
F_1 \otimes F_2 &\rightsquigarrow (F_1 \vee F_2) \wedge (\neg F_1 \vee \neg F_2).
\end{aligned}
$$

Figure 3.5: Rewrite rules for conversion to NNF[RV18]

Non-confluence of the NNF transformation Consider the formula $\neg(F_1 \leftrightarrow F_2)$. This formula can be rewritten in different ways. As an example, two strategies for applying the rewrite rules are given here, which lead to different results.

Example: Left-outermost reduction:

$$\neg(F_1 \leftrightarrow F_2) \rightsquigarrow F_1 \otimes F_2 \rightsquigarrow (F_1 \vee F_2) \wedge (\neg F_1 \vee \neg F_2)$$

Example: Left-innermost reduction (shortened):

$$
\begin{aligned}
&\neg(F_1 \leftrightarrow F_2) \rightsquigarrow \neg((F_1 \rightarrow F_2) \wedge (F_2 \rightarrow F_1)) \rightsquigarrow \\
&\neg(F_1 \vee \neg F_2) \vee \neg(\neg F_1 \vee F_2) \rightsquigarrow (\neg F_1 \wedge F_2) \vee (F_1 \wedge \neg F_2)
\end{aligned}
$$

With the first reduction, the resulting formula is already in CNF, while the second one is not, and is in disjunctive normal form (DNF) instead. This shows that matching the

order in which rewrite rules are applied between Vampire and Lean is very important to ensure that the same clauses are generated. Fortunately, the employed `simp` tactic features annotations like ↓, that can be used to match Vampire's outermost reduction strategy for the NNF step.

Handling Rewriting Transformations in Lean

Simplification of ⊤ and ⊥, Flattening, ENNF and NNF transformations are all treated similarly in the Lean file with a custom tactic. These tactics essentially just wrap the core `simp` tactic [Lea26a] with a set of rewrite rules. These rewrite rules can be expressed as theorems of the form $F \leftrightarrow G$, or with definitional equality $F = G$. Once such a rewrite rule is proven in Lean, it can be used with the `simp` tactic to rewrite in a formula. Proof terms for such rewrite rules are generated using suitable congruence lemmas, and previously proven theorems for the equivalence of the involved logical connectives.

Example: Flattening The flattening step is used as the simplest example for the demonstration of the rewriting transformations. It uses only three rewrite rules, which are the associativity of ∧ and ∨ as well as the double negation elimination. The quantification rules do not need to be explicitly stated, since the embedding of FOL in Lean does not differentiate between the two forms. A custom tactic can be defined via a macro, which wraps the `simp` tactic with the required rewrite rules as shown in Listing 3.4. The rewrite rules are proven as theorems (already provided by Lean Core).

```
syntax "flattening" "at" ident : tactic
macro_rules
  | `(tactic| flattening at $a) =>
    `(tactic | simp (config := {failIfUnchanged := false}) only
            [and_assoc, or_assoc, not_not] at $a:ident)

-- excerpt from Classical.lean
theorem not_not : ¬¬a ↔ a := Decidable.not_not

-- excerpt from SimpLemmas.lean
theorem and_assoc : (a ∧ b) ∧ c ↔ a ∧ (b ∧ c) :=
  Iff.intro (fun ⟨⟨ha, hb⟩, hc⟩ => ⟨ha, hb, hc⟩)
            (fun ⟨ha, hb, hc⟩ => ⟨⟨ha, hb⟩, hc⟩)

theorem or_assoc : (a ∨ b) ∨ c ↔ a ∨ (b ∨ c) :=
  Iff.intro ...
```

Listing 3.4: (Simplified) Tactic definition for the flattening step including used lemmas.

The application of this tactic on a concrete inference step of the running example is shown in Listing 3.5. At first, it introduces the premise as a local symbol `h` in the context. Then, the `flattening` tactic is applied to `h`, which rewrites the formula in `h`

```
theorem inf_s18 :
  (∀ v0 : ι, (((A=v0) ∨ (B=v0) ∨ (C=v0)) ∨ (¬(lives v0)))) →
  (∀ v0 : ι, ( (A=v0) ∨ (B=v0) ∨ (C=v0)  ∨ (¬(lives v0)))) := by
  intro h
  flattening at h<;>
  exact h

-- (Simplified) proof term constructed by simp for the theorem above
theorem inf_s18.{u} : ... :=
  fun h =>
  Eq.mp
    (forall_congr fun v0 => Eq.trans
      (propext (@or_assoc (A = v0) (B = v0 ∨ C = v0) ¬lives v0))
      (congrArg
        (Or (A = v0))
        (propext (@or_assoc (B = v0) (C = v0) (¬lives v0)))
      )
    )
  h
```

Listing 3.5: Flattening tactic applied on inference step of running example.

using the rewrite rules defined in Listing 3.4. Finally, the rewritten formula is exactly the same as the conclusion and the goal can be closed by `exact h`.

Below the application of the custom tactic, Listing 3.5 also shows the proof term constructed by `simp` in Lean. It uses congruence lemmas as well as propositional extensionality and transitivity of equality. For multiple rewrite steps, this proof can be very large, as well as tedious to generate in Vampire. Furthermore, Vampire does not necessarily need to know Lean internals (which congruence lemmas to apply when, etc.) to generate a checkable proof due to the split of responsibilities chosen in the architecture, showing the benefits of the chosen architecture for proof generation.

3.5.2 Rectification

Rectification is a preprocessing step that ensures all bound variables have unique names so that no "accidental" variable capture can occur.

Definition 3.5.1 (Rectification)**.** A formula is rectified if all free variables of this formula are distinct from its bound variables and each bound variable occurs in exactly one quantifier. [RV18]

In Lean, this would be straightforward to check, since it does not consider variable names as relevant and just uses the positions of variables, which define the corresponding de Bruijn indices. However, Vampire does not simply rename variables, but may also change the order of quantifiers which does not change the meaning of the formula, but leads to a syntactic difference which needs to be handled in the proof-checking. To

```
theorem inf_s3 :
  (¬((∃ v0 v1 : ι, (a v0 v1)) ↔ (∃ v0 v1 : ι, a v0 v1))) →
  (¬((∃ v0 v1 : ι, (a v0 v1)) ↔ (∃ v2 v3 : ι, a v3 v2))) := by
  intro h
  try simp only [forall_const, exists_const, -iff_self, -eq_self] at h
  have r0 (P :ι→ι→Prop) : (∃ v0 v1, P v0 v1) ↔ (∃ v1 v0, P v0 v1) :=
    Iff.intro (fun f  => let ⟨v0, v1, hP⟩ := f
      Exists.intro v1 (Exists.intro v0 hP))
      (fun f  => let ⟨v1, v0, hP⟩ := f
      Exists.intro v0 (Exists.intro v1 hP))
  conv in (∃ v2 v3 : ι, (a v3 v2)) =>
    rw [r0]
  symm_match using h
```

Listing 3.6: Demonstration of a rectification Proof in Lean. The variables are simultaneously renamed and the order of quantifiers is changed.

show this proof step to be sound in Lean, first a rewrite rule that changes the order of quantifiers is proven, which is then used to match the order of quantifiers between Vampire and Lean.

Example of Rectification

The running example does not contain such a rectification step, therefore another example is taken to demonstrate the employed method. Listing 3.6 shows how the change in order of the quantifiers is handled in Lean. The lemma `r0` is the rewrite rule which shows that the order of the two existential quantifiers can be changed. This example is only using two quantifiers, but the proof also generalizes to permutations of many quantifiers. Using the `conv` tactic, the position in the goal formula where the rewrite rule needs to be applied is targeted, which may be nested deep inside a formula. To ensure the correct position is targeted, the goal formula is rewritten which guarantees uniqueness of the rewritten vaiables due to the previously applied rectification step. The rewrite rule is applied using `rw`, matching the goal to the premise of the rectification step. Finally, the `symm_match` tactic is used to match the order of equalities between inferences, which may be changed by Vampire due to the symmetry of equality. This works for both $\forall$ and $\exists$ quantifiers, which need slightly different proofs for rewrite rules, but the same overall subproof structure is used for both cases.

3.5.3 CNF Transformation

Transformation from NNF to CNF is the last step in the preprocessing phase, which transforms the formula into a set of clauses, which is the input to the saturation loop. This transformation is also based on two rewrite rules that distribute $\vee$ over $\wedge$:

$$\begin{aligned} (F_1 \wedge F_2) \vee F_3 &\rightsquigarrow (F_1 \vee F_3) \wedge (F_2 \vee F_3) \\ F_1 \vee (F_2 \wedge F_3) &\rightsquigarrow (F_1 \vee F_2) \wedge (F_1 \vee F_3) \end{aligned}$$

Internally, Vampire does not directly apply these rewrite rules, but instead collects all the required literals and then constructs the resulting clause. Furthermore, the internal representation in Vampire of the formula is changed from a single formula to a set of clauses.

```
26. lives(sK0) & killed(sK0,A) [skolemisation 1,25]
...
29. killed(sK0,A) [cnf transformation 26]
30. lives(sK0) [cnf transformation 26]
```

Listing 3.7: Part of the running example proof CNF transformation step.

The proof in Vampire has an individual step for each clause added to the set of clauses, which can be seen from a fraction of the running example in Listing 3.7. The translation to Lean could be treated with individual theorems for each generated clause; however, this would lead to performance problems for large formulas, since the same rewrite steps need to be repeated for each generated clause. Instead, this transformation is treated only in the last section of the proof, where the rewriting is done once and the resulting clauses are named only locally. A section of the generated Lean file containing the CNF transformation for the running example is shown in Listing 3.8.

```
1  --from skolemization step 26
2  have step26 : (lives sK0) ∧ (killed sK0 A) ...
3  --cnf transformation
4  have step26' := by
5    prenexify at step26
6    cnfify at step26
7    exact step26
8  let ⟨s26c0, s26c1⟩ := step26'
9  ac_nf0 at s26c0 s26c1
10 have step29 : (killed sK0 A) := by
11   try simp only
12   ac_nf0
13   assumption
14 have step30 : (lives sK0) := by
15   try simp only
16   ac_nf0
17   assumption
```

Listing 3.8: Partial Lean output that corresponds to the CNF transformation for the running example.

Here, there is a challenge because the generated clauses in Lean may not have the same order as the generated clauses in Vampire. To resolve this, the clauses are brought into a normal form by sorting the literals[3] in each clause, after which they can be simply searched for by using the `assumption` tactic.

[3]The term order is defined internally in Lean.

All clauses in a Vampire proof are implicitly universally quantified, which is not the case for Lean which always needs explicit quantification. To resolve this mismatch, the formula which needs to be transformed to CNF is first brought to prenex normal form, and then brought to CNF moving the $\forall$ quantifiers in front of each individual clause. This way, each clause is explicitly quantified in the same way that Vampire treats them implicitly. The quantifiers are moved in front of clauses by rewrite rules, which are applied by the `cnfify` tactic after applying the distribution rules over the prenexed formula:

$$\begin{aligned} \forall x.(C \wedge D[x]) &\quad\rightsquigarrow\quad C \wedge \forall x.D[x] \\ \forall x.(C[x] \wedge D) &\quad\rightsquigarrow\quad \forall x.C[x] \wedge D \\ \forall x.(C[x] \wedge D[x]) &\quad\rightsquigarrow\quad \forall x.C[x] \wedge \forall x.D[x] \end{aligned}$$

Finally, depending on the order in which prenexification happens, the order of quantified variables may change, which is handled when necessary with the same method as described in Section 3.5.6.

3.5.4 Preprocessing Steps that Introduce new Symbols

Predicate definition introduction and skolemization are preprocessing steps that create new symbols. They do not have an individual theorem in the second section of the generated Lean file, but are included in the last section of the output. Vampire uses predicate definition introduction as a step in preprocessing to avoid exponential blowup. It is related to Tseitin transformation, but instead of giving each immediate subformula a new name, only "large" subformulas are named.

Skolemization also introduces new symbols, however for a different reason. It is a technique to remove existential quantifiers from a formula, by replacing the existentially quantified variable by a fresh Skolem function. This is possible due to the *axiom of choice*, which is implicitly assumed in Vampire and a fundamental axiom of Lean.

3.5.5 Predicate Definition

Predicate naming is treated in the third section of the generated Lean file, the full proof theorem. A new predicate symbol is introduced for a subformula, as well as a clause which defines the new predicate symbol as being equivalent to the subformula. This new clause, introduced by Vampire, may use only one direction (depending on polarity) of the equivalence, which is sufficient for the proof search. However, this needs to be handled properly in the translation to Lean. An example of the code generated for a predicate definition introduction step is shown in Listing 3.9. At first, the new predicate symbol p is defined using the `let` keyword. Then, the clause which is introduced by Vampire is derived from this definition. The tactic tries to match the direction of implication of the introduced clause in Lean by simply trying all three possibilities in order, until it finds the correct one.

```
-- step48 predicate definition introduction
let p v0  := (∃ v1, (R v1 ∧ ¬gt v0 v1)) ∧ (∃ v2, ¬U v2 ∧ ¬gt v0 v2)
have step48 : ∀ v0,
 (∃ v1, (R v1 ∧ ¬gt v0 v1)) ∧ (∃ v2, ¬U v2 ∧ ¬gt v0 v2) ∨ ¬p v0 := by
 intros v0
 have s : (∃ v1, (R v1 ∧ ¬gt v0 v1)) ∧ (∃ v2, ¬U v2 ∧ ¬gt v0 v2) ↔ p v0 :=
 ↪  Iff.rfl
 (first
   | exact s
   | exact or_comm.mp (imp_iff_not_or.mp s.mpr)
   | have res := (or_comm.mp (imp_iff_not_or.mp s.mpr));
     simp only[or_assoc] at res; trivial
   | exact imp_iff_not_or.mp s.mp)
```

Listing 3.9: Simplified Lean snippet for predicate naming.

3.5.6 Skolemization

To treat existential quantification in formulas, Vampire uses the technique of skolemization. A quantified variable can be replaced by a fresh function that depends on the outer quantified variables. It only applies to formulas in NNF, which means that complications arising from negated quantifiers do not need to be treated, since negations only occur in front of atoms.

There are several methods to perform skolemization, each of which can lead to different results. For example, quantifiers can be "mini-scoped" or "maxi-scoped" which changes the position of quantifiers in the formula to be as far inside subterms or outside as possible before checking which variables are used for the Skolem function [RSV16].

Vampire's default skolemization works by applying the following transformations to the original formula [RSV16]:

$$\begin{aligned} \forall x.\varphi[y_1,\ldots,y_n,x] &\quad\rightsquigarrow\quad \varphi[y_1,\ldots,y_n,x] \\ \exists x.\varphi[y_1,\ldots,y_n,x] &\quad\rightsquigarrow\quad \varphi[y_1,\ldots,y_n,f(y_1,\ldots,y_n)] \end{aligned}$$

All variables not bound by a quantifier are implicitly universally quantified. The $\exists$-quantified variable is changed to a function that depends on all variables occurring within the subformula where the rewrite happens.

This method of skolemization was found to be hard to check for soundness in Lean, and was therefore modified. This is a new method, which is typically called *structural* skolemization.

Structural skolemization uses the following rule:

$$\forall y_1,\ldots,y_n\,(C \circ \exists x.\varphi[y_1,\ldots,y_k,x] \circ D) \quad\rightarrow\quad C \circ \varphi[y_1,\ldots,y_k,f(y_1,\ldots,y_n)] \circ D$$

where C and D are arbitrary (possibly empty) subformulas and $\circ$ is any binary connective. The introduced function f is dependent on *all* variables bound by universal quantifiers in the subterm it occurs in.

This is different from the previous method because the used Skolem function is now dependent on all bound variables instead of only those occurring in the subformula. In Chapter 4, experiments will show that this skolemization approach does not deteriorate proof search in experiments. Nevertheless, this approach is easier to automatically check in Lean, which is discussed next.

Checking *structural* Skolemization in Lean

The method for showing the soundness of skolemization makes use of transforming a formula into existential prenex normal form. It is similar to standard prenex normal form (Definition 2.1.17), however only existential quantifiers occur in the prefix, and universal quantifiers may occur in the body. This means, to transform a formula to existential prenex normal form, all $\forall$ quantifiers are left at the original position in the formula, but $\exists$ are moved outwards in front of the formula.

In standard first-order logic, this is typically impossible without changing the meaning of the formula since the following transformation is not sound:

$$\forall x.\exists y.P(x,y) \not\equiv \exists y.\forall x.P(x,y)$$

However, using higher-order quantification (allowing to quantify over function symbols), a sound rule can be formulated in dependent type theory[4]:

$$(\forall(x:\alpha),\exists(y:b\ x),P(x,y)) \leftrightarrow (\exists f:(x:\alpha\to b\ x),\forall(x:\alpha),P(x,f(x))) \quad \text{skolem rewrite}$$

This means that it is possible to shift an $\exists$ quantifier to the left of a $\forall$ quantifier by existentially quantifying over a fresh function f, which depends on the variable x, that is bound by the $\forall$ quantifier. When prenexing like this, the resulting functions have exactly the same dependencies as the Skolem functions introduced by the structural skolemization method, which is why this method was implemented in Vampire.

Treating binary connectives Just being able to move $\exists$ before $\forall$ is not sufficient to show skolemization in Lean, and additional rewrite rules are required to move the existential quantifier outwards of binary connectives. Skolemization is applied to formulas in NNF, which means that only $\wedge$ and $\vee$ need to be treated. The following rewrite rules are used:

$$\begin{aligned} (\exists x.C[x]) \circ D &\quad\rightsquigarrow\quad \exists x.(C[x] \circ D) \quad \text{left rewrite} \\ D \circ \exists x.C[x] &\quad\rightsquigarrow\quad \exists x.(D \circ C[x]) \quad \text{right rewrite} \end{aligned}$$

where $C[x]$ and D are subformulas where x does not occur free in D, and $\circ \in \{\wedge,\vee\}$ These rewrite rules are not confluent, which means that the order of application needs to be replicated between Vampire and Lean.

[4]Notice, that $b\ x$ is typically just of the generic inhabited type of ι or a function of multiple arguments resulting in ι. The theorem is stated in the general form implemented in Lean, where b may depend on x

Due to early design choices, a specialized rewriting procedure was implemented for this skolemization method, to match the order between Vampire and Lean which is detailed in the next section.

Once the formula is brought into existential prenex normal form, the existential quantifiers can be replaced by appropriate Skolem constants and functions due to the axiom of choice, which is a fundamental axiom of Lean. The axiom of choice provides an arbitrary element of a non-empty type, which is exactly what our Skolem function is.

Transforming into Prenex Normal Form

The prenex normal form transformation is used as a requisite for the proof of several preprocessing inferences such as skolemization and NNF to CNF transformation. It is important to know the order of variables in the prenex normal form, which is determined by the order of application of the rewrite rules shown in the example below:

$$(\exists x.C[x] \vee \exists y.D[y]) \rightsquigarrow \exists x.(C[x] \vee \exists y.D[y]) \rightsquigarrow \exists x\ y.(C[x] \vee D[y]) \qquad \text{(left-right)}$$

$$(\exists x.C[x] \vee \exists y.D[y]) \rightsquigarrow \exists y.(\exists x.C[x] \vee D[y]) \rightsquigarrow \exists y\ x.(C[x] \vee D[y]) \qquad \text{(right-left)}$$

Please note that this is a special case. Since this rule does not correspond to an individual Vampire inference rule (but is a building block for proving inference rules), the order of application of the rewrite rules is not determined by Vampire, but can be chosen freely.

The design choice was to first simplify with only left rewrite rules and then right rewrite rules, both steps also including the Skolem rewrite rule. This procedure is repeated until neither the left nor the right rules can be applied further. The code in Lean for this procedure is shown in Listing 3.10. The code which replicates this behavior in Vampire is not shown here, but it uses a simplified representation of the formula as a tree that branches on the binary connectives.

```
syntax "exists_prenex" (" at " ident)? : tactic

macro_rules
`(tactic| exists_prenex) => `(tactic| repeat (first |
    simp only [or_exists_prenex_l, and_exists_prenex_l, Classical.skolem] |
    simp only [or_exists_prenex, and_exists_prenex, Classical.skolem]))
`(tactic| exists_prenex at $a:ident) => `(tactic | repeat (first |
    simp only [or_exists_prenex_l, and_exists_prenex_l, Classical.skolem] at
    ↪ $a:ident |
    simp only [or_exists_prenex, and_exists_prenex, Classical.skolem] at
    ↪ $a:ident))

theorem and_forall_prenex (ι : Type u) [hι : Nonempty ι]
                          (A : Prop) (B : ι → Prop) :
    (A ∧ (∀ v0 : ι, B v0)) ↔ (∀ v0 : ι, (A ∧ B v0)) := by
  constructor
  · intro h v0
    have a := h.left
    have b := h.right v0
    exact And.intro a b
  · intro h
    constructor
    · have a := h (Classical.choice hι)
      exact a.left
    · intro v0
      exact (h v0).right
```

Listing 3.10: Tactic definition for prenex normal form transformation and example of the proof and definition of a rewrite rule used in this transformation.

Running Example: Skolemization

Skolemization is applied twice in the running problem's proof. The corresponding LEAN code that is used to show soundness of the skolemization procedure is shown in Listing 3.11 for one of the inferences of the running example. The first line applies the prenex normal form transformation to the formula in the premise of step 12, which is the formula before skolemization. The second line uses the result of the first line to introduce a new symbol $sK1$ for the Skolem function as well as the skolemized function `step28'`.

Finally, the last line ensures that the skolemized formula is exactly the same as the conclusion of step 28 as derived by VAMPIRE, by using the `symm_match` tactic, which is a custom tactic that corrects for syntactic differences for the symmetry of the equivalence relation which is often used by VAMPIRE in this step.

The syntax `let ⟨sK1,step28'⟩ := step12` is syntactic sugar for the usage of two theorems: `choose {α : Sort u} {p : α → Prop} (h : ∃ x, p x) : α` and `choose_spec {α : Sort u} {p : α → Prop} (h : ∃ x, p x) : p (choose h)`

The first allows to introduce a new symbol $sK1$ for the Skolem function, while the second

```
-- step28 skolemization
exists_prenex at step12
let ⟨sK1,step28'⟩ := step12
have step28 : (∀ v0 : ι, (¬(hates v0 (sK1 v0)))) := by
              symm_match using step28'
```

Listing 3.11: Skolemization of the running example inference step 28. This block is in the final block in the generated Lean file as it defines a new symbol.

one returns the skolemized formula. This syntax can be generalized for multiple Skolem functions as well, which is required when multiple existential quantifiers are prenexed together.

3.5.7 Summary

Most preprocessing steps are sufficiently captured by rewrite rules. Since Vampire does not keep the direction of equalities and treats $\vee$ and $\wedge$ as associative and commutative, and may change the order of them without additional inference steps, special care needs to be taken to replicate these inferences in Lean.

Inferences that introduce new symbols, as well as the final CNF transformation, are treated only in the last section of the generated Lean file. A table indicating where each rule occurs is given in Table 3.1.

Inference Section	Full Proof Section
Simplification of $\top$ and $\bot$	Predicate naming
Rectification	Skolemization
Flattening	CNF transformation
ENNF transformation	
NNF transformation	

Table 3.1: Summary of preprocessing inferences and where they are reconstructed in the generated Lean proof output.

3.6 Saturation Loop Inferences

A large part of the proof output of Vampire consists of the steps of the superposition calculus, which are the inference rules introduced in Figure 2.1. Previous approaches to proof-checking of Vampire proofs have focused on checking only these inference rules [Raw+25; KRS25].

The approach taken in Ground Truth [Raw+25] passes grounded inference steps to an SMT-solver to check the validity of each inference step. This grounding is performed by substituting the variables of the inference steps by the unifier found by Vampire in

the premises and conclusions. This approach is limited to checking only the grounded inference steps, which is not what Vampire actually derives during proof search. However, it has the large benefit that it is relatively easy to implement and also has access to theory reasoning capabilities of the SMT-solver. While this approach does not satisfy the goal of this thesis, the idea of passing ground instances to another prover can be translated to the approach in this thesis because Lean has a powerful proof automation procedure `grind` [Lea26b]. It is not as fast as a state-of-the-art SMT-solver, but it is native to dependent type theory and produces verifiable proofs in Lean. The main idea is therefore to instantiate the inference rules with the unifier found by Vampire and then passing the result to `grind` within a theorem in Lean.[5]

3.6.1 Proof Replay

All superposition inference rules presented in Section 2.2.2 make use of a unifier σ which is an essential part of the inference step. However, the unifier is not stored during proof search, since this would amount to a significant amount of data to be stored for each inference, slowing down proof search as well as increasing memory usage. When doing this recording a straightforward way, for each variable in the clause the substituted term needs to be stored. To avoid this linear overhead in the amount of variables, the two previous approaches [Raw+25; KRS25], only a maximum of two information with constant size requirements are stored per inference, sufficient to reconstruct the unifier. However, this approach was found to be impractical, as it required reproducing fragile implementation details to reconstruct the unifier. To solve this problem in another way, "proof replay" was introduced in this work to reconstruct the missing information, mostly in the form of a unifier.

The idea is simple: take the premises of the inference step and apply the same inference rule that occurs in the proof while recording all required additional information. However, this is not quite as simple as the inference rules can give slightly different results when rerun, mostly because Vampire does treat $\vee$ as associative and commutative ($\wedge$ implicitly), as well as equality as symmetric and therefore "unordered", which can lead to a different order of literals and different directions of equality. Additionally, it is important to keep the selection function the same during proof replay as it was in proof search. The selection function determines which literals may be selected for inferences, and a different selection function would lead to different clauses and would restrict inferences to a different set of possible inferences. Furthermore, from two clauses, there may be multiple ways to apply the same inference rule. This means that multiple different clauses may be generated, and the one that was used in the original proof needs to be identified. Finally, in some inference steps, variables are renamed or normalized, which needs special care to ensure that the correct un-normalized unifier, is recorded.

[5] Note that this approach is not the same as checking only the grounded inference step like in [Raw+25], because the variables are still universally quantified for the proven theorem. One could think of the variables being introduced arbitrarily, but fixed.

High-level pseudocode of the proof replay is given in Algorithm 3.1. The necessary modifications to the inference rules are presented in Algorithm 3.2.

Algorithm 3.1: A high-level pseudocode of the proof replay implemented in Vampire.

```
   Data: Problem p
 1 initVampire(p)
 2 recordedOrdering := ordering
 3 proof := proofSearch()
   //Proof output starts here
 4 setupReplayEnvironment(recordedOrdering)
 5 outputPreamble()
 6 for s ∈ proof do
 7 |   if needsReplay(s) then
   |   |   //hooked inference rule recording in recorderInstance
 8 |   |   Inferences := runInferenceRule(s)
 9 |   |   for c ∈ Inferences do
10 |   |   |   if α − equivalent(c,s) then
11 |   |   |   |   recordings := getRecorderInstance().recordings
12 |   |   |   |   map := extractRenaming(recordings[c], s)
13 |   |   |   |   u := extractUnifier(recordings[c])
14 |   |   |   |   printProofStep(s, map, u)
15 |   |   |   |   break
16 |   |   |   end
17 |   |   end
18 |   else
19 |   |   printProofStep(s)
20 |   end
21 end
22 finalizeProof()
```

Resolving α-Equivalence, Associativity and Commutativity

When replaying an inference step, the generated clause may not be exactly the same as the clause in the original proof, but may differ in the order of literals, the direction of equalities and variables might be renamed. Therefore, it is not trivial to determine whether the generated clause corresponds to the inference step in the original proof. To resolve this, the generated clause is checked for α-equivalence to the original clause, which means that they are the same up to renaming of variables. The implementation uses existing Vampire code for checking multi-literal variants of clauses, `MLVariant`. This code treats $\vee$ using set semantics, so the order of literals does not matter as well as the direction of equalities. If the generated clauses match and the variables are renamed

Algorithm 3.2: Inference recording hook

```
1  Function runInferenceRule(premises):
2      ...
3      Specific inference logic (unifier is generated here)
4      ...
5      generatedClause := result
       //Hook here
6      if env.reconstruction then
7          recorder := getRecorderInstance()
8          recorder.recordClause(generatedClause, premises, unifier)
9      end
10     ...
11 end
```

between them, also the substitution $\rho_{\text{old-rec}}$ can be obtained, which matches the permuted variables of the generated (recorded) clause to the original clause.

To obtain the unifier for the complete inference step σ, the recorded inference step σ_{rec} is combined with the renaming $\rho_{\text{old-rec}}$ as a composition $\sigma = \rho_{\text{old-rec}}\sigma_{\text{rec}}$. Note, that no inverse renaming is required, since σ_{rec} always maps variables to terms of the original input clauses and only the variable names of the newly generated clause may differ, which is handled by the renaming $\rho_{\text{old-rec}}$.

Replay Overhead In contrast to earlier approaches [Raw+25; KRS25], which record additional reconstruction data already during saturation, this approach avoids adding bookkeeping to the saturation loop.

In the newly contributed design, replay is executed only for steps occurring in proof output, i.e., after proof search has finished and only for steps that require reconstruction. This shifts overhead away from the critical path of clause generation and selection. Consequently, slowdowns during proof search are not expected; some overhead in proof output generation is expected, but since each rule has been applied once already during proof search, the time this takes is manageable. Quantitative measurements of this overhead are reported in Chapter 4.

3.6.2 Example Inference

To illustrate the reconstruction of an inference step, we take a resolution rule occurring in the running example. On step 70 of the proof in Listing 3.2, Vampire performs resolution:

```
theorem inf_s70 :
  (∀ v0 : ι, ¬(killed v0 A) ∨ (hates B v0)) →
  (killed sK0 A) →
  (hates B sK0) := by
  intros h0 h1
  have i0 := h0 sK0
  have i1 := h1
  grind only [cases Or]
```

Listing 3.12: Lean code corresponding to the inference rule of step 70 of the running example.

$$\frac{\forall x, \neg\text{killed}(x, \text{A}) \vee \text{hates}(\text{A}, x) \qquad \text{killed}(sK_0, \text{A})}{\text{hates}(\text{A}, sK_0)}\ \text{res}$$

The resolved upon literals are $\neg\text{killed}(x, \text{A})$ and $\text{killed}(sK_0, \text{A})$, which unify with the mgu $\sigma = \{x \mapsto sK_0\}$.

When generating the Lean proof output, the unifier is not known because it is not saved during proof search. To reconstruct the unifier, the same two clauses are taken and the Vampire resolution rule is applied to them again. This time, with an environment that hooks the inference rule to record the unifier as described in Algorithm 3.1 and Section 3.6.1. This unifier can then be used to generate the Lean proof shown in Listing 3.12.

Reading this listing from top to bottom, the theorem statement starts with the name `inf_s70`. It is defined to have two premises, which are the two clauses used for resolution, and the conclusion. From a dependent type theory perspective, this means that `inf_s70` is of the type of a function that takes two arguments, which are proofs of the two premises, and returns a proof of the conclusion. To show that the type of `inf_s70` is inhabited (i.e., the theorem is valid), we construct a suitable proof term using Lean tactics, which is marked by the **by** keyword.

The first tactic that is used is `intros h0 h1`, which takes the assumptions of the theorem and introduces them as local variables in the proof context. Hence, the premises have now the names `h0` and `h1` in the proof context.

The next two lines use the `have` tactic, which allows one to instantiate the quantifiers in the premises. For this step, the recorded unifier $\sigma = \{x \mapsto sK_0\}$ is required, which can be seen in the **have** `i0 :=` `h0 sK0` line. This line instantiates the quantifier with sK_0 and names the resulting formula `i0`. The next line does the same for the second premise, but since it is already ground, no instantiation is required.

Finally, the `grind` tactic is used to generate a proof term for the resolution conclusion. As already mentioned `grind` is a proof automation tactic, that is based on a combination of proof search techniques - similar in capabilities to an SMT-solver. It is not suitable

for generating complicated unification steps, but the now instantiated terms, which are ground, can be handled by `grind`.

In this example, it might not be clear why one would need to use a proof automation tactic like `grind` instead of applying a suitable resolution theorem directly. However, when more complicated clauses are involved, Vampire does not necessarily keep the order of literals in the clauses, requiring the proof to use suitable applications of associativity and commutativity of `∨`. Furthermore, the direction of equalities may also be changed, which can make it difficult to apply a suitable resolution theorem directly. `grind` is able to handle these complications, which is why it is used for generating the proof term for the conclusion of the inference step. Similar problems were encountered on translating cvc5 (an SMT-solver) proofs to Lean [CBA26].

3.6.3 Design Choices for Superposition Inference Rules

Usage of `grind` The approach of calling `grind` to generate the proof term is convenient because it allows to handle the complications mentioned above. For sufficiently simple inference steps, it might even be powerful enough to prove the conclusion without needing correct unification. If, for some reason, Vampire then produces a wrong unifier which somehow still leads to a valid inference step, the current approach would not be able to detect this. In a sense this does not matter, because the generated proof is still correct (if the rest of the proof is correct), but from the perspective of a Vampire developer, it would be important to catch such a bug.

Selection Function Furthermore, the calculus only allows certain applications of the inference rules, depending on the selection function. This approach does not consider any of these restrictions, which could also be important information for developers. Furthermore, if saturated inferences should be checked at some point in the future, that would necessarily require reasoning over the selection function.

Formulation of the Inference Rule as a Theorem One could include each inference rule as a lemma using `have lemma_x := by` in one big proof. This could potentially reduce repetition of the same formulas in the theorem statements. The current approach repeats each clause at least twice, once as a conclusion and once as a premise. Nevertheless, the current approach was still deemed beneficial for readability and maintainability of the generated proof, as well as for avoiding performance problems encountered with large contexts in one big proof. Furthermore, splitting the proof into multiple files containing multiple theorems can also enable easier parallelizability of proof checking, which may be important for larger proofs.

3.7 AVATAR Inferences

Vampire's AVATAR reasoning combines the saturation loop with a propositional SAT solver. It is one reason for the performance of Vampire, but it also introduces addi-

```
theorem inf_s77 : (sA2→((A=B))) → ¬(A=B) →
  (sA2→False) := by
  intros h0 h1 x2
  have i0 := h0 x2
  have i1 := h1
  grind only [cases Or]
```

Listing 3.13: Handling of A-clauses in the proof output. This is the inference step of step 77 of the running example, which is an inference on a regular clause and an A-Clause.

tional inference rules that need to be checked. Fortunately, Lean provides the tactic `bv_decide`, which can be used to check SAT inferences. Additional inferences used can be translated by similar methods to previously discussed transformations, such as predicate definition introduction as well as other rewrite rules. In this section, inference rules as well as modifications to the previously established inference steps are discussed that are required to check AVATAR inferences. To make the discussion more concrete, the running example is used, which contains all AVATAR inferences.

3.7.1 Definition Introduction

When a clause is split as described in Section 2.4, a new symbol needs to be introduced representing the propositional variable. This is very similar to the predicate definition introduction, and even simpler since the bi-equivalence is always used. The new symbol is defined using the **let** keyword, and the clause introduced by Vampire is derived from this definition which can be seen in Listing 3.3.

3.7.2 A-Clauses in Inference Rules

To handle A-clauses in the proof output, every (saturation loop) inference rule must be able to handle A-clauses as well as normal clauses. The AVATAR assertions are included as assumptions in each premise and conclusion of the inference rule, which is naturally represented using the $\rightarrow$ connective. When instantiating the premises and goal variables, the AVATAR assertions of the goal also need to be instantiated, as seen in the first line of the Lean code in Listing 3.13. The variable `x2` corresponds to the assumption `sA2`. When instantiating the premises with the found unifiers, the AVATAR assertions are applied first, then the variables are instantiated, which properly handles the assertions. In the given example, no variables need to be instantiated, and `x2` is used for the assertion of the first premise.

3.7.3 Contradiction and Component Clauses

Contradiction and component clauses provide the interface between the propositional reasoning of the AVATAR solver and the first-order reasoning of the saturation loop. These two inferences are mostly a technicality and have occurred in similar forms in

```
-- step 61 avatar component clause
theorem inf_s61 : (sA2 ↔ C=sK0) →
  (sA2→(C=sK0)) := by
  intro h component
  have new := h.mp component
  (first | trivial | ac_nf at new ⊢ | grind [cases Or])
```

```
-- step 78 avatar contradiction clause
theorem inf_s78 : (sA2→False) →
  ¬sA2 := by
  intro h
  try simp only [imp_false, imp_iff_not_or, not_not] at h
  exact h
```

Listing 3.14: AVATAR component and contradiction clause inferences from the running example.

previous inference rules: predicate definition introduction and false simplification. The component clause shown in Listing 3.14 takes the bi-implication from the definition to an implication, sufficient for proof search. The AVATAR contradiction clause displayed in Listing 3.14 transforms the refutation found in the superposition calculus to a new SAT proposition, which is the negated assertion.

3.7.4 Splitting

```
theorem inf_s66 : ((B=sK0) ∨ (C=sK0) ∨ (A=sK0)) →
  (sA3 ↔ B=sK0) →
  (sA2 ↔ C=sK0) →
  (sA1 ↔ A=sK0) →
  (sA1 ∨ sA2 ∨ sA3) := by
  intros h0 h1 h2 h3
  try rw[h1]
  try rw[h2]
  try rw[h3]
  try simp only [imp_iff_not_or] at h0
  prenexify
  prenexify at h0
  intros
  have newForm := h0
  simp only [not_and_or, not_not, eq_comm] at newForm
  simp only [eq_comm]
  ac_nf at newForm ⊢ <;>
  grind only [cases Or]
```

Listing 3.15: Simplified Lean output for splitting a clause into variable disjoint parts used in AVATAR for SAT solving.

The translation of splitting a clause into variable disjoint parts is essentially a repeated

application of rewrite rules that use the equivalences between formulas with the new symbols for original formulas. Recovering the employed splitting is essential for this step, which is currently not recorded during proof search, but is recovered during proof output generation. The code for a split rule in the running example is shown in Listing 3.15. At first, all the hypotheses are introduced, where the first one, `h0`, is the clause that is split, and `h1` to `h3` are the symbol definitions made in previous steps. Using the symbol definitions, the (propositional) goal is rewritten to match `h0` using the `rw` tactic. For this inference, this would (almost) be sufficient, but to match every possible inference Vampire makes, several steps are taken to transform the formula to the correct form. At first, the formula is normalized and transformed to prenex normal form. After that, the order of the variables is matched between the original and generated formula by the `intros` and `have` pair. In the example, no variables are introduced but generally this is required. Special care to match the order of quantifiers between original and generated formula is handled analogously to the CNF transformation. Finally, the formula is normalized again, which typically concludes the proof of this inference. If this is not sufficient, `grind` is called as a backup, which can handle further ordering differences of equalities.

3.7.5 AVATAR Refutation

The final step that concludes an AVATAR proof is to show that the constructed SAT formula is unsatisfiable. Internally, Vampire uses the SAT solver CaDiCaL [Bie+24] [6] for this step, which can produce a proof of unsatisfiability in the DRAT format. Vampire has access to this proof and can emit it. In a previous approach to checking Vampire proofs [KRS25], the SAT proof was also translated in detail to be checked by the ITP. However, in this work, it was decided to not translate the SAT proof, which may potentially be large, but instead to use the `bv_decide` tactic in Lean, which can decide the satisfiability of propositional formulas. It reruns a SAT solver internally, and then checks this (newly found) proof. This is not only convenient, but may also be faster than translating the proof, since Lean does not check the proof in the kernel, but instead trusts a verified DRAT proof checker implemented in Lean [Böv+25]. The tactic `bv_decide` is designed for bitvector reasoning, but can also be used for Boolean formulas. However, all variables that were used so far in the proof are of type `Prop`, which is the type of propositions, different from the Type `Bool`, that is higher in the type hierarchy. Nevertheless, using an additional assumption that the propositional variables are in the `decidable` type class, we can transform the propositional formula to Boolean variables using rewrite rules.

The code for the AVATAR refutation of the running example is shown in Listing 3.16. The first theorem `inf_s95'` is the refutation of the propositional formula using the `Bool` type, which is proven using the `bv_decide` tactic. To make the result usable

[6] This is not completely true, as z3 (SMT solver) may also be instead of CaDiCaL. Since this thesis focuses on checking only the first-order with equality fragment without theories, relying on a SAT solver is sufficient.

```
theorem inf_s95' (sA1' sA2' sA3' : Bool) :
  (sA1' || sA2' || sA3') → (!sA2') → (!sA3') → (!sA1') → False := by
  intro h
  bv_decide

theorem inf_s95 : (sA1 ∨ sA2 ∨ sA3) → (¬sA2) → (¬sA3) → (¬sA1) → False := by
  classical
  have satProof := inf_s95' (decide sA1) (decide sA2) (decide sA3)
  rewrite_decide_eq [sA1 sA2 sA3]
  sat_norm
  exact satProof
```

Listing 3.16: AVATAR refutation of the running example. The first theorem `inf_s95'` is the refutation of the propositional formula using the `Bool` type, while the second theorem `inf_s95` uses the `Prop` type.

with the rest of the proof, the second theorem `inf_s95` transforms `inf_s95'` to a theorem using the `Prop` type, which is the type of propositions used in the rest of the proof. This is achieved by transforming the goal to the same form as `inf_s95'` using rewrite rules. Notice the use of the tactic `classical` in the proof of `inf_s95`, which allows to use that all propositions are decidable, which is required for this transformation.

3.7.6 Summary

Including AVATAR inferences is necessary to check the full proof output of Vampire. It requires additional treatment of the assertions of the A-clauses in general inference rules, which can be treated in a natural way. Existing Lean automation enables checking the AVATAR refutation without the need to also translate the SAT proof. Splitting needs the most recovery work within Vampire and Lean.

CHAPTER 4

Experiment and Results

In this chapter, our Vampire to Lean proof translation method developed in this thesis is evaluated. At first, the two modifications to Vampire preprocessing (skolemization, pure predicate removal) that were required to check the full proof output are evaluated with respect to their impact on proof search performance. Then, the proof replay method for reconstructing the unifier of inference steps is evaluated.

After that, the time that is required to check generated Lean proofs is evaluated, and the results are compared with two other methods invoking the ATP Duper.

4.1 Experimental Setup

All experiments used benchmarks from version 9.2.1 of the TPTP library [Sut24]. To run benchmarks, the tool BenchExec [Bey16] was used, with a custom configuration for Vampire and Lean. All experiments were conducted on a server with two AMD Epyc 7502 2.5GHz processors and 1TB RAM. The time and memory limits were different between experiments and are specified in the respective sections. The version of Vampire used for all experiments is based on version 5.0.1. The evaluated Git commit of Vampire is: e001d3e[1]. The used version of Lean is v4.29.0 with the corresponding version of the required library VampLean v4.29.0. Lean was run with default options, except for an increased thread stack size of 64MiB (`-s 655536`), which was required for some of the larger proofs. It includes a dependency on the `mathlib` library, which has version v4.29.0 as well. Finally, for all steps that used Duper, the version v4.29.0 was used.

The benchmark set for the evaluation of Vampire proof generation consists of all problems from the TPTP library which are in CNF and FOF format. In total, there are 17603

[1] e001d3e6021cab23a4f49eaacd9b2dca59e284b4

problem instances, 9259 of which are in FOF format and 8344 in CNF format. These problems include both satisfiable and unsatisfiable problems. The problems which are reported as refuted (and hence have a proof output), are passed to Lean for proof checking.

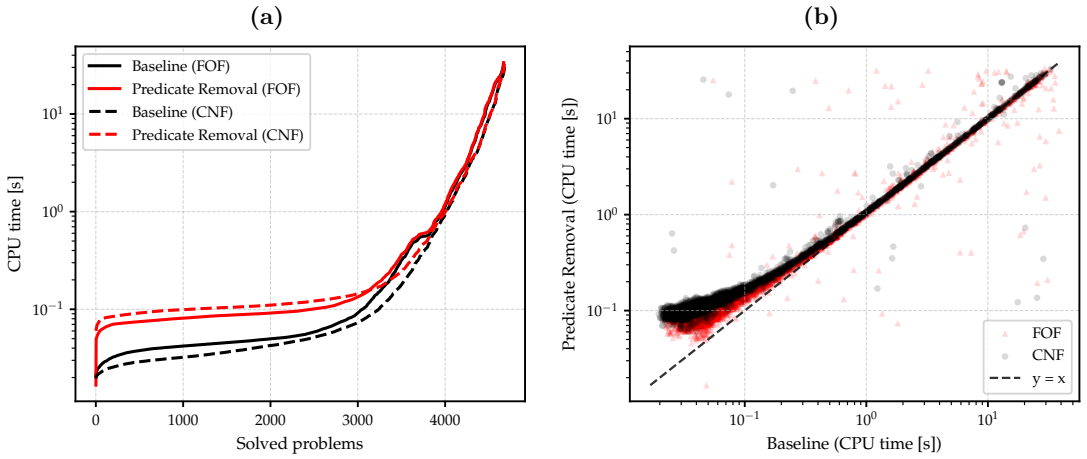
4.2 Evaluation of Proof Inference Modifications

Vampire was configured to use the Discount saturation loop, with otherwise default settings. A limit of 100-giga-instructions was used to limit proof search, which roughly corresponds to a time limit of 20-40 seconds on the used hardware. Furthermore, each process was limited to one core and 16GB of RAM.

To make sure that the modifications to preprocessing do not have vastly detrimental effect on proof search performance, the modified version of Vampire is compared to the unmodified version on the same benchmark set.

4.2.1 Pure Predicate Removal

The pure predicate removal is a preprocessing step that removes predicates that only occur with one polarity in the input problem. This is a hard step to check in Lean, since it requires semantic reasoning about the input problem. To avoid this complication, an option `-ppr` was added to Vampire to disable this preprocessing step.



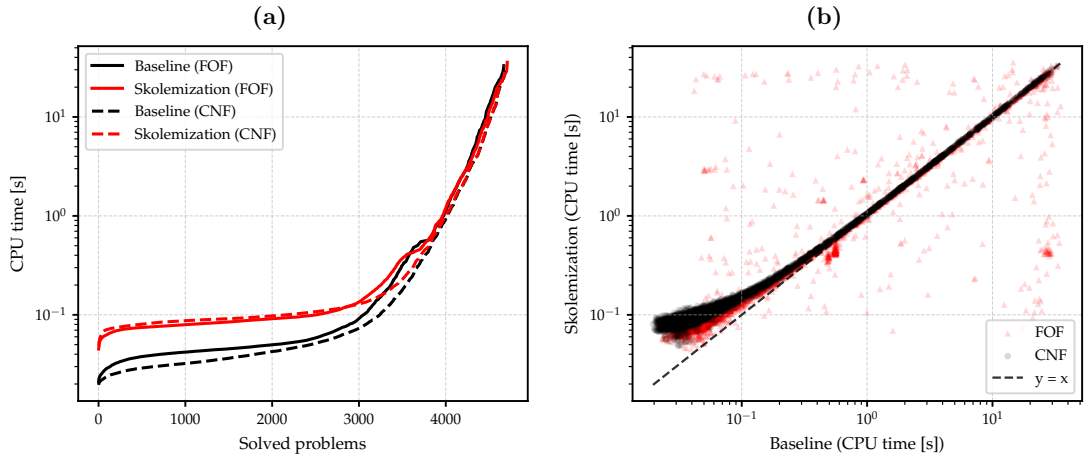
Fragment	Baseline	PPR	Δ
CNF	4681	4678	-3
FOF	4668	4670	2
Total	9349	9348	-1

Figure 4.1: Cactus plot (**a**) and scatter plot (**b**) comparing the runtimes of refuted and satisfiable problems between baseline Vampire and with the option `-ppr off` that disables pure predicate removal. The table shows the total number of solved problems for each fragment and the total.

The difference in runtime is shown in Figure 4.1. Both plots show that the baseline version of Vampire with pure predicate removal is slower for "easy" problems which most likely is due to an unoptimized implementation overhead of the pure predicate removal as well as slightly longer search times due to the additional predicate. For harder problems (that take longer to solve), the time between the two versions is quite similar. The scatter plot shows that some problems are solved faster with pure predicate removal, while others are solved faster without it. The table shows that the total number of solved problems is almost the same for the chosen instruction limit. It can be concluded, that disabling pure predicate removal does not significantly affect the performance of Vampire, and hence it is a good choice to disable it for the sake of proof checking.

4.2.2 New Skolemization Method

As already stated in Section 3.5.6, the default skolemization method of Vampire is hard for proof checking using Lean and was replaced by a new method, more suitable for proof checking. As shown in the Table in Figure 4.2, the new skolemization method actually solves 41 more problems than the old one, which could potentially be explained by restricting the search space of viable unifications. However, this was not investigated in detail, and the complicated interactions during proof search make it difficult to determine an exact reason for this improvement.



Fragment	Baseline	New Skolemization	Δ
CNF	4681	4681	0
FOF	4668	4709	**41**
Total	9349	9390	**41**

Figure 4.2: Cactus plot (**a**) and scatter plot (**b**) comparing the runtimes of refuted and satisfiable problems between baseline Vampire and with the option `--skolemization syntactic` that changes the skolemization method. Points where only one of the two versions solved the problem within the timeout are included, but points where neither version solved the problem are not included. The table shows the total number of solved problems for each fragment and the total.

It is also clearly visible that this modification does not have a significant effect on the CNF problems, which is strongly expected since skolemization only needs to be applied to FOF problems, because CNF problems feature no existential quantification. They are still included in this analysis to check that no overall problems are introduced by including the new skolemization into the codebase. It is further noteworthy that the new skolemization method is faster for harder problems, which is indicated by the first crossover point at roughly 0.3 seconds in the cactus plot 4.2 **a)**. Up to this point, the old skolemization method performs better for the employed benchmark set. Furthermore, notice there is a cluster of points on the right side of the scatter plot which are solved by the structural skolemization method and correspond to the problems `CSR116+*` in TPTP, which can now be solved in the given timeout. This cluster contributes substantially to the increased number of solved problems, for which the new method seems to be exceptionally beneficial.

It can be concluded that the new skolemization method, which is simpler for proof checking, does not significantly affect the performance of Vampire and even improves it slightly on the given benchmark.

4.2.3 Combined Methods

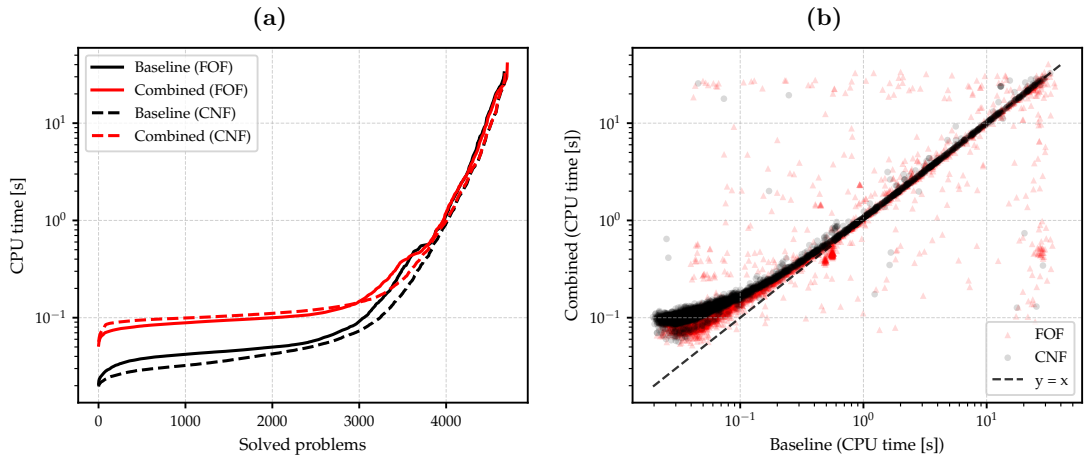
Since both methods are employed at the same time for proof checking, also the combined effect of both methods needs to be evaluated since they may not interact trivially. It can be seen in Figure 4.3 that the combined effect of both options is very similar to the combined effect of both options separately, however losing some of the improvements in the FOF fragment.

In general, the combined effect of both options is a slight performance increase for the tested settings. In conclusion, this means that the modifications of inference for proof checking do not have a significant negative effect on the performance of Vampire and even improve it slightly for "hard" problems.

4.2.4 Proof Replay and Output Generation

Replaying inferences to reconstruct the unifier as well as recording some additional information during proof search is expected to add some overhead to Lean proof output generation. To evaluate the overhead of proof replay, the time from the combined method is compared to the time that is required to generate the Lean proof output (using structural skolemization and pure predicate removal off).[2] The results are shown in Figure 4.4. For many problems, the time for proof replay and proof printing does not add a significant overhead. However, some problems show a time overhead up to a factor of approximately 6. The only problem with a higher time overhead is problem `PLA044+1` with a 21.5-fold time increase. Such outliers have also been observed in previous work [KRS25], which even reported a time increase of 65 times for one problem.

[2] used options: `--skolemization syntactic --proof_extra lean --proof leancheck -om lean -sa discount -ppr off -i 100000`



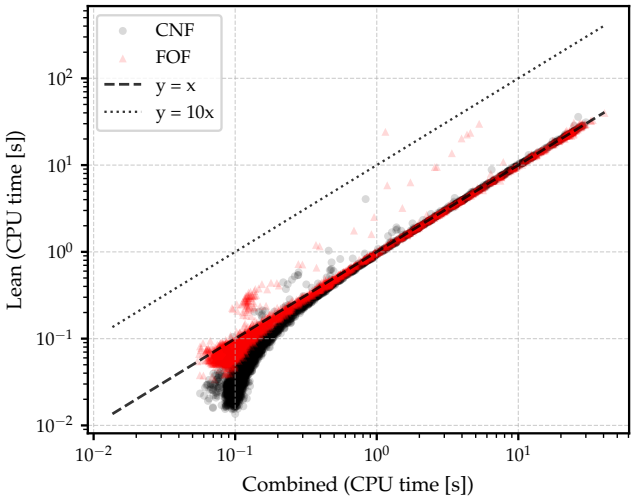
Fragment	Baseline	New Skolemization	Δ
CNF	4681	4678	-3
FOF	4668	4704	**36**
Total	9349	9382	**33**

Figure 4.3: Cactus plot (**a**) and scatter plot (**b**) comparing the runtimes of refuted and satisfiable problems between baseline Vampire and with the options `--skolemization syntactic -ppr off` as explained previously. Points where only one of the two versions solved the problem within the timeout are included, but points where neither version solved the problem are not included. The table shows the total number of solved problems for each fragment and the total.

This method performed all necessary reconstruction within Vampire (including applications of associative and commutative properties), which is not the case for the method in this thesis, which is a likely explanation for the lower overall time increase experienced for Lean proof replay and output generation. Four further problems are lost due to a timeout of 60 seconds. For these problems, the generated proof has been investigated with a longer timeout, and it was found that the generated Lean files are typically large (up to 650 MB, with over 100000 inference steps). Clearly, replaying such a high number of inference steps is expected to take a long time, as well as constructing and writing the proof to disk.

To determine the average time increase in proof output generation, a linear regression was performed on the runtimes. It was found that the proof generation only adds $1.0\% \pm 0.3\%$ overhead on average.

In conclusion, the modifications to Vampire for proof checking do not add a significant overhead to proof search, and the proof replay method for reconstructing the unifier of inference steps adds only a small overhead to proof output generation on average, although some outliers with a much higher time increase exist. Overall, more proofs are found including generation of the proof output, compared to baseline within the given instruction and time limits due to the new skolemization method.



Fragment	Baseline	Lean output	Δ
CNF	4681	4677	-4
FOF	4668	4698	**30**
Total	9349	9375	**26**

Figure 4.4: Scatter plot comparing the time required for proof search with the combined method to the time required for generating the Lean proof output.

4.3 Evaluation of Lean Proof Checking

It has been shown that generating suitable proof output for Lean does typically not incur significant overhead, however, the generated proof output still needs to be checked in Lean, which is evaluated in this section. For this evaluation, the resulting Lean files from Section 4.2.4 are used. In total, these are 8334 problems, 4118 in CNF and 4216 problems in FOF. These numbers are smaller than previously reported "solved" problems, because only refuted problems produce Lean proof output, excluding satisfiable problems from the evaluation. A timeout of 1000 s is used for this step as well as a memory limit of 64 GB. Furthermore, Lean is run with the option `-s 65536` to increase the maximum stack size, which is required for large proofs.

4.3.1 Comparisons with Duper

To be able to compare the results with other (related) approaches, two versions using the ATP Duper were implemented and evaluated as well. These two methods are briefly described next:

Duper on Inferences This method uses the proof skeleton generated by Vampire, but instead of providing the unifier for each inference step, Duper is called on each inference step to reprove the inference without any additional information. This method is closer to the Ground Truth approach [Raw+25], but rather than using an SMT solver, a proof-producing ATP is used. Some translations, such as skolemization and predicate removal are kept the same as in the Lean method described in this thesis, to avoid extra complications for Duper. The preamble also stays the same, which is required to be

```
...
-- step 53 resolution
theorem inf_s53 : (∀ v0 : ι, ¬(lives v0) ∨ B = v0 ∨ C = v0 ∨ A = v0) →
  (lives sK0) →
  (B = sK0 ∨ C = sK0 ∨ A = sK0) := by
  duper [*]
...
theorem inf_s95 : (sA1 ∨ sA2 ∨ sA3) → ¬sA2 → ¬sA3 → ¬sA1 → False := by
  duper [*]

theorem fullProof : ... := by
  apply Classical.byContradiction; intro step15
  ...
  -- step28 skolemisation
  exists_prenex at step12
  let ⟨sK1,step28'⟩ := step12
  have step28 : (∀ v0 : ι, ¬(hates v0 (sK1 v0))) := by symm_match using
  ↪  step28'
  have step29 : killed sK0 A := by
    duper [*]
  ...
  have step44 := inf_s44  step35 step29
  ...
  let sA3 := B = sK0
  ...
  exact step95
end vamproof
```

Listing 4.1: Simplified and shortened Lean file for the Duper on Inferences method corresponding to the running example. The general proof structure of Vampire is kept, and the individual inference steps are re-proven using Duper.

able to use the same symbols and definitions. An example of the generated Lean file for this method is shown in Listing 4.1 which is the translated running example.

Duper on Minified Problem A second method to use Duper is inspired by the use of Metis in Sledgehammer [BN10]. The proof that Vampire generates is completely discarded, except that only the axioms that are directly used in the proof by Vampire are given to Duper. This means that only relevant assumptions are included for Duper, which can greatly reduce the search space, and thus speed up proof search. This approach is not necessarily useful to check whether the found Vampire proof is sound, since it does not check the Vampire proof itself but only checks if the conclusion can be derived from the used axioms. Nevertheless, it is an interesting comparison between the translation and "native" proof search. Especially when aiming to build a hammer, this comparison is of relevance since approaches like Sledgehammer employ a similar method.

```
...
variable {ι : Type u}
variable [inst : Inhabited ι]
variable {sK1 :  ι → ι}
  {sK0 A B C : ι}
  {killed hates richer :  ι → ι → Prop}
  {lives :  ι → Prop}
variable {sA1 sA2 sA3 : Prop}

theorem fullProof :
  (∃ v0 : ι, lives v0 ∧ killed v0 A) →
  ...
  (∀ v0 : ι, ∃ v1 : ι, ¬hates v0 v1) → ¬(A = B) →
  killed A A := by
  duper [*]
```

Listing 4.2: Minimized problem passed to Duper. In the case of the running example, no unnecessary axioms were included, so no minimization occurred.

4.3.2 Results

In Table 4.1, a summary of the results of successfully checked proofs in Lean is shown. Within the given time and memory limits, 8237 of 8334 proofs found by Vampire using the modified inference rules (skolemization, primitive predicate removal) were successfully checked in Lean from assumption to conclusion. Hence, 98.8% of all found proofs within the CNF and FOF fragments were checked with this approach, largely increasing the confidence in the soundness of these Vampire proofs. The remaining 1.2% of proofs were not checked successfully, which is due to timeouts, memory limits or stack overflows. These failures are the results of the large size of the generated Lean files taxing the Lean parser, evaluator and kernel.

TPTP	Vampire	Lean			
		Success	Timeout (1000s)	OOM (64GB)	Stack overflow
CNF	4118	4062	31	24	1
FOF	4216	4175	35	3	3

Table 4.1: Number of refutation proofs found by Vampire with new skolemization and pure predicate removal off (column 2) vs Lean proof reconstructions (columns 3–6), sorted by success/failure reasons. Success means that Lean fully validates the Vampire proof. Notice that only refutation proofs are included in this table, compared to previous analyses which also included saturated problems.

Figure 4.5 shows a cactus plot comparing the runtimes of checking generated Lean proofs to the two methods using Duper. Many problems are checked by Lean within seconds. It has to be noted that the minimum time that has been observed for checking a proof in Lean is approximately 1.8 s as rather large dependencies of Mathlib and Duper are loaded for each run.

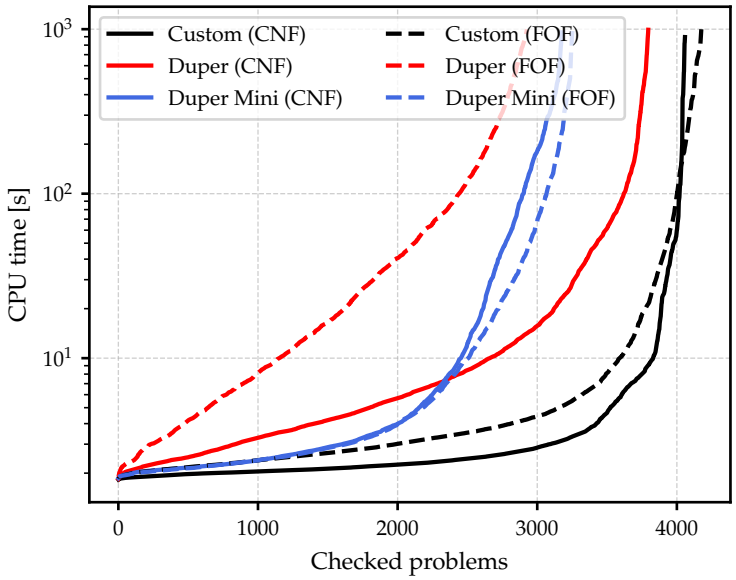


Figure 4.5: Cactus plot comparing the runtimes of checking generated Lean proofs to the two methods using Duper as described previously. "Duper" corresponds to the method invoking Duper on inferences, while "Duper Mini" corresponds to the method invoking Duper on the minified problem.

	Successfully checked proofs		
Method	CNF	FOF	Total
New Method	4062	4175	8237
Duper on inferences	3795	2926	6721
Duper on minified problem	3175	3252	6427

Table 4.2: Total amount of successfully checked proofs for the method developed in this thesis vs. the two methods using Duper.

Comparison with Duper methods The method from this thesis solves more problems than both methods using Duper as expected. Details are shown in Table 4.2. Moreover, it is interesting to see that the method using Duper to reconstruct individual inferences performs worse for the FOF fragment than running Duper on the minified problem. For the CNF fragment, the method of invoking Duper on the minified problem is faster for "simple" problems, but for harder problems, reconstructing inferences with Duper is faster. This may be explained by the fact that for harder problems, the search space of the minified problem is too large, while reconstructing individual inferences with Duper can be more efficient since it only needs to find a proof for a single inference step. Furthermore, the difference between CNF and FOF shows, that reconstructing the transformations performed by Vampire from FOF to CNF is rather time consuming. This suggests that the method of reconstructing these inferences may not be optimal.

In Figure 4.6, scatter plots comparing the runtimes between the methods are shown. For the CNF fragment, there are almost no cases for which the methods using Duper are faster than checking the full generated Lean proof. This is not the case for the FOF fragment, which is faster for some problems using Duper (in both ways). This is a further confirmation that the translation of the preprocessing steps could be improved, since multiple minified problems are checked faster than the full generated Lean proof. This could be due to the fact that Duper performs less transformation steps or that these steps are easier to check than the steps generated by Vampire. However, on average the custom method is faster than both methods using Duper. An exponential regression ($t_{\text{Duper}} = C \cdot t_{\text{Cus.}}^{E}$) between the runtimes of methods on successfully checked proofs (not including timeouts) gave an exponent of $E = 1.3$ for Duper on minified problems and $E = 1.8$ for Duper on inferences.

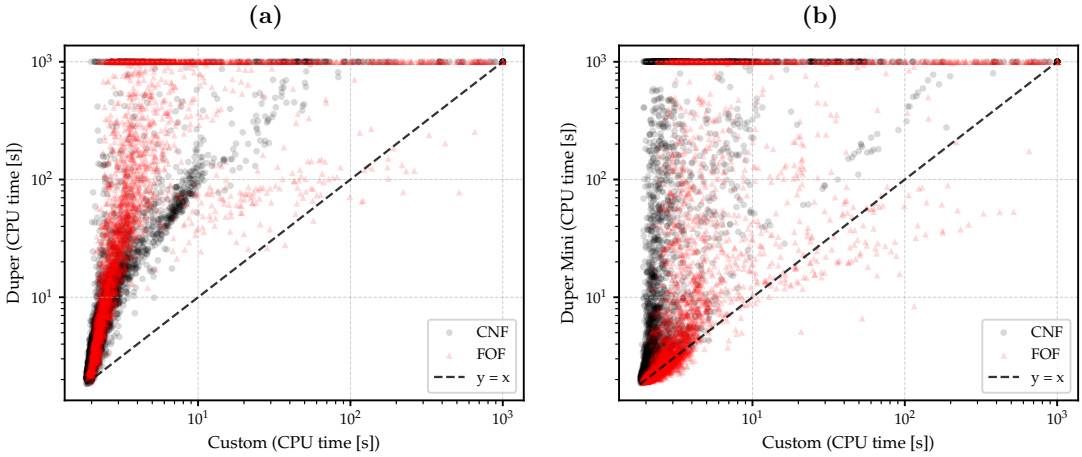


Figure 4.6: Scatter plots comparing the two methods invoking Duper to the full method. A line $y = x$ is included to indicate the point where two methods have the same runtime.

Analysis of Runtime vs. Problem size The dependence on the amount of theorems and size of the generated Lean file vs. the required checking time is investigated in Figure 4.7. The increasing time of checking proofs with the number of theorems is expected. However, the superlinear increase of checking time with the number of theorems indicates performance issues with the checking of large proofs. The reason for this increase can be due to many reasons, such as the tendency of problems with larger formulas to have more theorems, or implementation details in Lean which cause an increase of checking time with the number of theorems, due to e.g. larger contexts. Plot **(b)** in Figure 4.7 shows the checking time vs. file size, which shows slightly better asymptotic behaviour, indicating an approximately linear increase in logarithmic scale, which indicates an exponential with an exponent of roughly $1.5 - 2.5$ depending on the branch. The red branch with the steepest slope, visible in both scatter plots, belongs to problems `CSR113+*` - `CSR116+*` in TPTP. These seem to suffer from particularly large CNF transformations due to a large input formula and potential performance problems

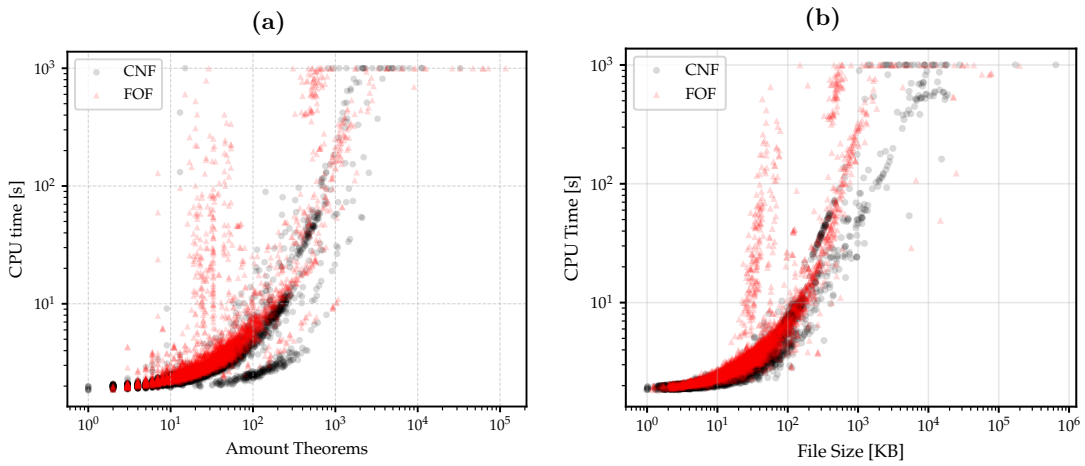


Figure 4.7: Scatter plot comparing the number of theorems as well as the size of the Lean input file to the time required for checking the proof.

in the applied Lean tactic for checking the generated proof. Replacing this with a more efficient implementation may reduce the checking time for these problems.

CHAPTER 5

Conclusion and Future Work

This thesis presented a novel method for checking the soundness of Vampire proofs in Lean completely from assumptions to conclusion, which has not been achieved with any previous method. This required modifications to the inference rules of Vampire, which were evaluated to not have a significant negative effect on the performance of Vampire and even improve it slightly for some problems. A translation to Lean was introduced, which is flexible to be adapted to further inference steps and provides a middle ground between full reconstruction in Vampire and full proof search in Lean. The method was evaluated on a large benchmark set, showing that 99% of all found proofs within the CNF and FOF fragments were checked successfully in Lean, increasing the confidence in the soundness of these Vampire proofs.

There are many possibilities for future work, such as adding support for more logic fragments, like typed first order logic, or higher-order logic (currently only supported by a specific branch of Vampire). Furthermore, support for reasoning about theories such as arithmetic operations in different domains or the theory of arrays could be added. Additionally, advanced reasoning rules employed by Vampire such as induction could be supported. Using a suitable translation layer from Lean to Vampire like `lean-auto` [Qia+25], a hammer for Lean using Vampire as a backend could be built, employing the translation presented in this thesis. To increase the performance of proof checking, a more suitable, intermediate language could be developed for outputting the proof from Vampire, which is easier to check than the generated Lean files.

Options for future development are numerous, and the method developed in this thesis provides a starting point for further research in this direction, leading to a significant increase of trust in the results of Vampire. With the current advances of large language models in the field of autoformalization and theorem proving, the developed method may also find application by combining large language models with Vampire providing verifiable proofs for sub-problems suggested by the language model.

Overview of Generative AI Tools Used

All ideas and methods described in this thesis were developed by the author.

- GitHub Copilot was used for suggestions while editing Vampire and Lean code as well as for writing the thesis.
- For writing this thesis, GitHub Copilot was used to improve writing style, correct for grammar and spelling mistakes.

For the latter task, a prompt like the following was used:

"Improve the writing of the Background chapter. Focus on fixing spelling mistakes and improve the flow. Do not change the meaning and do not add new information."

GitHub Copilot may use several language models such as GPT-4o, GPT-5 mini, GPT-5.3-Codex or Claude Haiku.

List of Figures

2.1 Superposition calculus inference rules 12

3.1 Workflow for proof checking 23
3.2 Internal proof-checking workflow 24
3.3 Flattening rewrite rules 30
3.4 ENNF rewrite rules 30
3.5 NNF rewrite rules 30

4.1 PPR runtime comparison 52
4.2 Skolemization runtime comparison 53
4.3 Combined-method runtime comparison 55
4.4 Proof replay and output overhead 56
4.5 Lean vs Duper cactus plot 59
4.6 Lean vs. Duper scatter plots 60
4.7 Problem size vs. Lean checking time 61

List of Tables

2.1 Curry–Howard correspondence 17

3.1 Preprocessing inference summary 40

4.1 Lean proof reconstruction outcomes 58
4.2 Total amount of successfully checked proofs for the method developed in this thesis vs. the two methods using Duper. 59

List of Algorithms

3.1 Proof replay pseudocode 42
3.2 Inference recording hook 43

List of Listings

2.1 Lean definition of Or 18
3.1 Dreadbury Mansion formalization 25
3.2 Excerpt of a Vampire proof 26
3.3 Simplified full Lean proof 28
3.4 Flattening tactic definition 31
3.5 Flattening inference example 32
3.6 Rectification proof example 33
3.7 CNF transformation in Vampire proof 34
3.8 CNF transformation in Lean output 34
3.9 Predicate naming in Lean 36
3.10 Prenex tactic and rewrite rule 39
3.11 Skolemization example (step 28) 40
3.12 Resolution inference in Lean (step 70) 44
3.13 A-clause handling in Lean (step 77) 46
3.14 AVATAR component and contradiction 47
3.15 AVATAR split clause in Lean 47
3.16 AVATAR refutation in Lean 49
4.1 Duper on inferences: Lean file 57
4.2 Minified problem for Duper 58

Acronyms

ATP automated theorem prover. 1–4, 11, 51, 56

CNF conjunctive normal form. xiv, 8, 10, 27, 29, 30, 33–35, 38, 40, 48, 51–56, 58–60, 63, 68

DNF disjunctive normal form. 30

ENNF equivalence negation normal form. 29, 30, 40

FOF first-order formulas. 29, 51–56, 58–60, 63

FOL first-order logic. 5, 7, 17, 27, 31

ITP interactive theorem prover. 2, 3, 21, 48

mgu most general unifier. 9, 10, 12, 44

NNF negation normal form. 8, 29–31, 33, 36–38, 40

SAT satisfiability. 1, 8, 19–21, 45–49

SMT satisfiability modulo theories. 1–3, 40, 41, 44, 45, 48, 56

Bibliography

[Ahr+00] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, et al. „The Approach: Integrating Object Oriented Design and Formal Verification“. In: JELIA. Vol. 1919. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 21–36. DOI: 10.1007/3-540-40006-0_3.

[Arm+11] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, et al. „A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses“. In: CPP. Vol. 7086. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 135–150. DOI: 10.1007/978-3-642-25379-9_12.

[Ass+23] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, et al. „Dedukti: A Logical Framework Based on the λΠ-calculus modulo Theory“. In: *CoRR* abs/2311.7185 (2023). DOI: 10.48550/ARXIV.2311.07185. arXiv: 2311.07185.

[Bai26] Chris Bailey. *Ammkrn/Nanoda_lib*. Apr. 8, 2026. URL: https://github.com/ammkrn/nanoda_lib.

[Bár+25] Filip Bártek, Ahmed Bhayat, Robin Coutelier, Márton Hajdu, Matthias Hetzenberger, et al. „The Vampire Diary“. In: CAV. Vol. 15933. Cham: Springer Nature Switzerland, 2025, pp. 57–71. DOI: 10.1007/978-3-031-98682-6_4.

[Bar91] Henk Barendregt. „Introduction to Generalized Type Systems“. In: *J. Funct. Program.* 1.2 (Apr. 1991), pp. 125–154. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796800020025.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. „A Brief Overview of Agda – a Functional Language with Dependent Types“. In: TPHOLs. Vol. 5674. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78. DOI: 10.1007/978-3-642-03359-9_6.

[Bey16] Dirk Beyer. „Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016)“. In: TACAS. Vol. 9636. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 887–904. DOI: 10.1007/978-3-662-49674-9_55.

[BF25] Raven Beutner and Bernd Finkbeiner. „Checking Satisfiability of Hyperproperties Using First-Order Logic“. In: ATVA. Vol. 15055. Cham: Springer Nature Switzerland, 2025, pp. 198–211. DOI: 10.1007/978-3-031-78750-8_10.

[BG94] Leo Bachmair and Harald Ganzinger. „Rewrite-Based Equational Theorem Proving with Selection and Simplification“. In: *J. Log. Comput.* 4.3 (1994), pp. 217–247. ISSN: 0955-792X, 1465-363X. DOI: 10.1093/logcom/4.3.217.

[BHS26] Clark Barrett, Thomas A. Henzinger, and Sanjit A. Seshia. „Certificates in AI: Learn but Verify“. In: *Commun. ACM* 69.1 (Jan. 2026), pp. 66–75. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3737447.

[Bie+24] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, et al. „CaDiCaL 2.0“. In: CAV. Berlin, Heidelberg: Springer-Verlag, July 26, 2024, pp. 133–152. DOI: 10.1007/978-3-031-65627-9_7.

[BN10] Sascha Böhme and Tobias Nipkow. „Sledgehammer: Judgement Day“. In: IJCAR. Vol. 6173. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 107–121. DOI: 10.1007/978-3-642-14203-1_9.

[Bod+26] Jonas Bodingbauer, Márton Hajdu, Laura Kovács, Axel Polaczek, and Michael Rawson. *Lean on Vampire Proofs (Short Paper)*. Mar. 27, 2026. DOI: 10.48550/arXiv.2603.26342. arXiv: 2603.26342 [cs]. Pre-published.

[Bol+25] Matthew Bolan, Joachim Breitner, Jose Brox, Nicholas Carlini, Mario Carneiro, et al. *The Equational Theories Project: Advancing Collaborative Mathematical Research at Scale*. 2025. DOI: 10.48550/ARXIV.2512.07087. arXiv: 2512.07087. Pre-published.

[Böv+25] Henrik Böving, Siddharth Bhat, Luisa Cicolini, Alex Keizer, Léon Frenot, et al. „Interactive Bitvector Reasoning Using Verified Bit-Blasting“. In: OOPSLA. Vol. 9. Oct. 9, 2025, pp. 3259–3285. DOI: 10.1145/3763167.

[Car19] Mario Carneiro. „The Type Theory of Lean“. MA thesis. Carnegie Mellon University, 2019.

[CBA26] Joshua Clune, Haniel Barbosa, and Jeremy Avigad. „Hint-Based SMT Proof Reconstruction“. In: TACAS. Vol. 16505. Springer, Apr. 11, 2026, pp. 255–275. DOI: 10.1007/978-3-032-22752-2. arXiv: 2601.14495 [cs].

[CK18] Łukasz Czajka and Cezary Kaliszyk. „Hammer for Coq: Automation for Dependent Type Theory“. In: *J. Autom. Reasoning* 61.1–4 (June 2018), pp. 423–453. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-018-9458-4.

[Clu+24] Joshua Clune, Yicheng Qian, Alexander Bentkamp, and Jeremy Avigad. „Duper: A Proof-Producing Superposition Theorem Prover for Dependent Type Theory“. In: *LIPIcs Vol. 309 ITP 2024* 309 (2024), 10:1–10:20. ISSN: 1868-8969. DOI: 10.4230/LIPICS.ITP.2024.10.

[Cza20] Łukasz Czajka. „Practical Proof Search for Coq by Type Inhabitation“. In: IJCAR. Vol. 12167. Cham: Springer International Publishing, 2020, pp. 28–57. DOI: 10.1007/978-3-030-51054-1_3.

[Des+25] Remi Desmartin, Omri Isac, Grant Passmore, Ekaterina Komendantskaya, Kathrin Stark, et al. „A Certified Proof Checker for Deep Neural Network Verification in Imandra“. In: *LIPIcs Vol. 352 ITP 2025* 352 (2025), 1:1–1:21. ISSN: 1868-8969. DOI: 10.4230/LIPICS.ITP.2025.1.

[DKS97] Jörg Denzinger, Martin Kronenburg, and Stephan Schulz. „DISCOUNT - a Distributed and Learning Equational Prover“. In: *J. Autom. Reasoning* 18.2 (Apr. 1, 1997), pp. 189–198. ISSN: 1573-0670. DOI: 10.1023/A:1005879229581.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. 2. Graduate Texts in Computer Science. New York: Springer Science+Business Media, 1996. 326 pp. ISBN: 978-1-4612-7515-2.

[Geo+22] Pamina Georgiou, Bernhard Gleiss, Ahmed Bhayat, Michael Rawson, Laura Kovács, et al. „The RAPID Software Verification Framework“. In: FMCAD. TU Wien Academic Press, Oct. 2022, pp. 255–260. DOI: 10.34727/2022/isbn.978-3-85448-053-2_32.

[Gir72] J. Girard. „Interpretation Fonctionelle et Elimination Des Coupures Dans l'aritmetique d'ordre Superieur“. In: 1972. URL: https://www.semanticscholar.org/paper/Interpretation-fonctionelle-et-elimination-des-dans-Girard/e1a1c345ce8ab4c11f176f1c42bcfc6a62ef4e3c.

[GKR20] Bernhard Gleiss, Laura Kovács, and Jakob Rath. „Subsumption Demodulation in First-Order Theorem Proving“. In: IJCAR. Vol. 12166. LNCS. Cham: Springer International Publishing, 2020, pp. 297–315. DOI: 10.1007/978-3-030-51074-9_17.

[Hur03] Joe Hurd. „First-Order Proof Tactics in Higher-Order Logic Theorem Provers“. In: STRATA. 2003, pp. 56–68. URL: http://www.gilith.com/papers.

[Jea+24] Simon Jeanteur, Laura Kovács, Matteo Maffei, and Michael Rawson. „CryptoVampire: Automated Reasoning for the Complete Symbolic Attacker Cryptographic Model“. In: IEEE SP. San Francisco, CA, USA: IEEE, May 19, 2024, pp. 3165–3183. DOI: 10.1109/SP54263.2024.00246.

[Kor+23] Konstantin Korovin, Laura Kovács, Giles Reger, Johannes Schoisswohl, and Andrei Voronkov. „ALASCA: Reasoning in Quantified Linear Arithmetic“. In: TACAS. Vol. 13993. Cham: Springer Nature Switzerland, 2023, pp. 647–665. DOI: 10.1007/978-3-031-30823-9_33.

[KRS25] Anja Petković Komel, Michael Rawson, and Martin Suda. „Case Study: Verified Vampire Proofs in the LambdaPi-calculus Modulo“. In: *CoRR* (2025). DOI: 10.48550/ARXIV.2503.15541.

[KV13] Laura Kovács and Andrei Voronkov. „First-Order Theorem Proving and Vampire“. In: CAV. Vol. 8044. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–35. DOI: 10.1007/978-3-642-39799-8_1.

[Lac+24] Hanna Lachnitt, Mathias Fleury, Leni Aniva, Andrew Reynolds, Haniel Barbosa, et al. „IsaRare: Automatic Verification of SMT Rewrites in Isabelle/HOL“. In: TACAS. Vol. 14570. Cham: Springer Nature Switzerland, 2024, pp. 311–330. DOI: 10.1007/978-3-031-57246-3_17.

[Lea26a] Developers Lean. *Simp*. Lean Community. Apr. 2026. URL: https://leanprover-community.github.io/extras/simp.html.

[Lea26b] Developers Lean. *The Grind Tactic*. Mar. 2026. URL: https://lean-lang.org/doc/reference/4.29.0/The--grind--tactic/.

[Lio96] J.L Lions. *ARIANE 5 Failure - Full Report*. Ariane 5 Flight 501 Failure. July 19, 1996. URL: https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html.

[LOB24] Evan Laufer, Alex Ozdemir, and Dan Boneh. „zkPi: Proving Lean Theorems in Zero-Knowledge“. In: CCS. Salt Lake City UT USA: ACM, Dec. 2, 2024, pp. 4301–4315. DOI: 10.1145/3658644.3670322.

[LT93] N.G. Leveson and C.S. Turner. „An Investigation of the Therac-25 Accidents“. In: *Computer* 26.7 (July 1993), pp. 18–41. ISSN: 1558-0814. DOI: 10.1109/MC.1993.274940.

[Mcc03] William Mccune. *OTTER 3.3 Reference Manual*. ANL/MCS-TM-263, 822573. Oct. 27, 2003, ANL/MCS-TM–263, 822573. DOI: 10.2172/822573.

[Mcc97] William Mccune. „Solution of the Robbins Problem“. In: *J. Autom. Reasoning* 19.3 (Dec. 1997), pp. 263–276. ISSN: 0168-7433, 1573-0670. DOI: 10.1023/A:1005843212881.

[Moh+25] Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, et al. „Lean-Smt: An SMT Tactic for Discharging Proof Goals in Lean“. In: CAV. Vol. 15933. Cham: Springer Nature Switzerland, 2025, pp. 197–212. DOI: 10.1007/978-3-031-98682-6_11.

[MP08] Jia Meng and Lawrence C. Paulson. „Translating Higher-Order Clauses to First-Order Clauses“. In: *J. Autom. Reasoning* 40.1 (Jan. 2008), pp. 35–60. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-007-9085-y.

[MU21] Leonardo De Moura and Sebastian Ullrich. „The Lean 4 Theorem Prover and Programming Language“. In: CADE. Vol. 12699. Cham: Springer International Publishing, 2021, pp. 625–635. DOI: 10.1007/978-3-030-79876-5_37.

[NR01] Robert Nieuwenhuis and Albert Rubio. „Paramodulation-Based Theorem Proving“. In: *Handbook of Automated Reasoning (in 2 Volumes)*. Elsevier and MIT Press, 2001, pp. 371–443. DOI: 10.1016/B978-044450813-3/50009-6.

[NWP02] *Isabelle/HOL*. Vol. 2283. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002. DOI: 10.1007/3-540-45949-9.

[Pel86] Francis Jeffry Pelletier. „Seventy-Five Problems for Testing Automatic Theorem Provers“. In: *J. Autom. Reasoning* 2.2 (June 1, 1986), pp. 191–216. ISSN: 1573-0670. DOI: 10.1007/BF02432151.

[Qia+25] Yicheng Qian, Joshua Clune, Clark Barrett, and Jeremy Avigad. „Lean-Auto: An Interface between Lean 4 and Automated Theorem Provers“. In: CAV. Vol. 15933. Cham: Springer Nature Switzerland, 2025, pp. 175–196. DOI: 10.1007/978-3-031-98682-6_10.

[Raw+25] Michael Rawson, Andrei Voronkov, Johannes Schoisswohl, and Anja Petković Komel. „Ground Truth: Checking Vampire Proofs via Satisfiability modulo Theories“. In: CADE. Vol. 15943. Cham: Springer Nature Switzerland, 2025, pp. 136–149. DOI: 10.1007/978-3-031-99984-0_8.

[Rij25] Egbert Rijke. *Introduction to Homotopy Type Theory*. Cambridge Studies in Advanced Mathematics. Cambridge: Cambridge University Press, 2025. DOI: 10.1017/9781108933568.

[RSV16] Giles Reger, Martin Suda, and Andrei Voronkov. „New Techniques in Clausal Form Generation“. In: GCAI. 2016, pp. 11–3. DOI: 10.29007/dzfz.

[Run22] Neha Rungta. „A Billion SMT Queries a Day (Invited Paper)“. In: CAV. Vol. 13371. Cham: Springer International Publishing, 2022, pp. 3–18. DOI: 10.1007/978-3-031-13185-1_1.

[RV03] Alexandre Riazanov and Andrei Voronkov. „Limited Resource Strategy in Resolution Theorem Proving“. In: *J. Symb. Comput.* First Order Theorem Proving 36.1 (July 1, 2003), pp. 101–115. ISSN: 0747-7171. DOI: 10.1016/S0747-7171(03)00040-3.

[RV18] Giles Reger and Andrei Voronkov. *Vampire Manual*. internal. 2018.

[ŠR26] Artjoms Šinkarovs and Michael Rawson. *When Agda Met Vampire*. Feb. 21, 2026. DOI: 10.48550/arXiv.2602.18844. arXiv: 2602.18844 [cs]. Pre-published.

[Sut24] Geoff Sutcliffe. „Stepping Stones in the TPTP World“. In: *Automated Reasoning*. Vol. 14739. Cham: Springer Nature Switzerland, 2024, pp. 30–50. DOI: 10.1007/978-3-031-63498-7_3.

[Sut25] Geoff Sutcliffe. *CASC-30 Results*. The CADE ATP System Competition. July 28, 2025. URL: https://tptp.org/CASC/30/WWWFiles/DivisionSummary1.html.

[Tea24] The Coq Development Team. *The Coq Proof Assistant*. Zenodo, Sept. 4, 2024. DOI: 10.5281/zenodo.14542673.

[US10] Josef Urban and Geoff Sutcliffe. „Automated Reasoning and Presentation Support for Formalizing Mathematics in Mizar“. In: *Intelligent Computer Mathematics*. Vol. 6167. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 132–146. DOI: 10.1007/978-3-642-14128-7_12.

[Vor14] Andrei Voronkov. „AVATAR: The Architecture for First-Order Theorem Provers“. In: CAV. Vol. 8559. Cham: Springer International Publishing, 2014, pp. 696–710. DOI: 10.1007/978-3-319-08867-9_46.

[WHH14] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. „DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs“. In: SAT. Vol. 8561. Cham: Springer International Publishing, 2014, pp. 422–429. DOI: 10.1007/978-3-319-09284-3_31.