Tao's Equational Proof Challenge Accepted

Lydia Kondylidou[1], Jasmin Blanchette[1], and Marijn J.H. Heule[2]

[1] Ludwig-Maximilians-Universität München, Munich, Germany
{l.kondylidou,jasmin.blanchette}@lmu.de
[2] Carnegie Mellon University, Pittsburgh, United States
marijn@cmu.edu

Abstract. In the context of the Equational Theories Project, Terence Tao posed the challenge of finding alternatives to a complicated 62-step proof found by the Vampire superposition prover. We introduce a proof minimization tool called Krympa. Using a combination of brute force and heuristics, and exploiting both Vampire and the Twee equational prover, the tool reduces the 62-step proof to 20 steps, each corresponding to a rewrite. In an empirical evaluation, it also performs well on 1431 equational problems originating from the same project, reducing in particular a 151-step proof to only 10 steps.

Keywords: Theorem provers · Equational logic · Proof minimization.

1 Introduction

The Equational Theories Project [7], launched in September 2024 by Fields medalist Terence Tao, aims at exploring the relations between different equational theories of magmas. A *magma* is a basic algebraic structure consisting of a set equipped with a single binary operation $\diamond$ closed on that set. The project's first phase, concluded in April 2025, focused on equational laws for magmas that contain at most four applications of $\diamond$.

The project uses the Lean [22] proof assistant to express proofs and counterexamples but depends on automatic theorem provers and other external tools. The problems explored in the project's first phase all fall within first-order logic's unit equality fragment: They consist of a $\forall$-quantified equation as the sole axiom and a $\forall$-quantified equation as the proof goal, or conjecture.

For the problem $650 \Longrightarrow 448$, where 650 denotes the axiom $\forall x, y, z.\ x = x \diamond (y \diamond ((z \diamond x) \diamond y))$ and 448 denotes the conjecture $\forall x, y, z.\ x = x \diamond (y \diamond (z \diamond (x \diamond z)))$, the Vampire [5] superposition prover found a particularly complex proof, with 62 inference steps, excluding clausification and Skolemization. Given that the proof is unintelligible, Tao challenged the community to find "an alternate proof, by whatever means you wish—human, semi-automated, or automated" [28].

One idea could be to run a specialized equational prover, Twee [24], instead of Vampire, but this results in a very long, 137-step proof. Another approach would be to use Lean's automation, such as the `aesop` [20], `canonical` [23], `duper` [9], and `grind` [1] tactics and the LeanHammer [30], to reconstruct and compress consecutive superposition steps, in the style of Sledgehammer's structured proof reconstruction [6, Sect. 6.3]. This would yield a shorter and more high-level proof, in which each step may combine multiple rewrites. Our approach is orthogonal. Our working hypothesis is that Vampire's 62-step proof, which emerged as the byproduct of a saturation process, is likely suboptimal. By mixing and matching proofs generated by different automatic provers, as proposed by Sutcliffe et al. [27], we hope to achieve a shorter, simpler proof.

We introduce Krympa, a tool that minimizes equational proofs by decomposing them into independently provable components and reassembling them into more concise, intelligible proofs. Specifically, starting from a Vampire-generated proof, the tool transforms it into a direct proof (Sect. 3) and analyzes its inferences to break it down into intermediate results that serve as candidate lemmas. Each of these lemmas is then proved independently using Vampire and Twee (Sect. 4), the two leading systems in the unit equality division of CASC 2025 [26]. The resulting proofs are then combined into a single proof using heuristics that favor shorter derivations (Sect. 5).

Given the 62-step Vampire proof of $650 \implies 448$, our tool produces a 20-step proof, where 13 steps are generated by Twee (Sect. 6). In a larger empirical evaluation, we applied the tool to 1431 provable implications from the Equational Theories Project and obtained positive results (Sect. 7). In particular, the tool reduced a 151-step Vampire proof to 10 steps.

Our tool is implemented in Rust, OCaml, and Python. Its source code is available at `https://github.com/kondylidou/Krympa`. The files associated with Tao's challenge and our empirical evaluation data are also available online [19].

2 Background

We briefly review the Vampire and Twee automatic provers and their associated proof formats.

2.1 Vampire and Superposition Proofs

Vampire is a saturation-based theorem prover for first-order logic with equality based on the superposition calculus [4]. It implements highly optimized search strategies and data structures, and integrates techniques such as literal selection, term orders, redundancy elimination, strategy scheduling, and portfolios.

Superposition works on implicitly $\forall$-quantified clauses. A preprocessor performs clausification and Skolemization. For example, the axiom $\forall x.\ \mathsf{f}(x) = \mathsf{g}(x)$ is transformed into $\mathsf{f}(x) = \mathsf{g}(x)$, where x is a free variable, and the conjecture $\forall x.\ \mathsf{f}(x) = \mathsf{g}(x)$ is negated and transformed into $\mathsf{f}(\mathsf{sk}) \neq \mathsf{g}(\mathsf{sk})$, where sk is a Skolem constant. The objective is to derive the contradictory clause $\perp$. For the unit equality fragment, the calculus's two relevant inference rules are as follows:

$$\frac{t \neq u}{\perp}\ \text{equality resolution} \qquad \frac{t = t' \qquad s[u] \bowtie s'}{\mu(s[t'] \bowtie s')}\ \text{superposition}$$

The equality resolution rule has one premise, $t \neq u$, one conclusion, $\perp$, and one side condition: that t and u are unifiable. The superposition rule has two premises and one conclusion. The $\bowtie$ symbol denotes either $=$ or $\neq$ throughout the rule. The $=$ and $\neq$ operators are commutative; for example, the premise $t = t'$ can match the equation $\mathsf{f}(\mathsf{a}) = \mathsf{b}$ either as is or as $\mathsf{b} = \mathsf{f}(\mathsf{a})$. The premises are assumed to have disjoint sets of variables, which can be achieved by renaming. Also in the rule, $s[\,]$ is a term with a hole, the terms $s[u]$ and $s[t']$ are obtained by filling the hole in $s[\,]$ with u and t', and μ is a most general unifier of t and u. For example, the most general unifier of the terms $\mathsf{h}(\mathsf{a}, y)$ and $\mathsf{h}(x, \mathsf{b})$ is $\{x \mapsto \mathsf{a},\ y \mapsto \mathsf{b}\}$; applying it on both terms yields $\mathsf{h}(\mathsf{a}, \mathsf{b})$. Finally, the rule has further side conditions, not shown here, that restrict the search space.

Example 1. A subtle case of the superposition rule arises when both premises are the same clause. Consider the following rule instance, where the variable in the second premise has been renamed to avoid a clash:

$$\frac{\mathsf{f}(\mathsf{f}(x)) = \mathsf{g}(x) \qquad \mathsf{f}(\mathsf{f}(x')) \neq \mathsf{g}(x')}{\mathsf{f}(\mathsf{g}(x)) \neq \mathsf{g}(\mathsf{f}(x))}\,\text{superposition}$$

This instance is obtained by taking $t := \mathsf{f}(\mathsf{f}(x))$, $t' := \mathsf{g}(x)$, $\bowtie := \neq$, $s[\,] := \mathsf{f}([\,])$, $u := \mathsf{f}(x')$, $s' := \mathsf{g}(x')$, and $\mu = \{x' \mapsto \mathsf{f}(x)\}$. Applying the unifier μ to both premises yields the equation $\mathsf{f}(\mathsf{f}(x)) = \mathsf{g}(x)$ and the disequation $\mathsf{f}(\mathsf{f}(\mathsf{f}(x))) \neq \mathsf{g}(\mathsf{f}(x))$. The inference replaces the subterm $\mathsf{f}(\mathsf{f}(x))$ in the disequation with $\mathsf{g}(x)$ using the equation as a left-to-right rewrite rule, and derives the conclusion. ■

Example 2. Vampire implements *parallel superposition*, a variant of the superposition rule in which multiple subterms that match a term are replaced. The following inference illustrates this:

$$\frac{\mathsf{b} = \mathsf{a} \qquad \mathsf{h}(\mathsf{b}, \mathsf{a}, \mathsf{b}) \neq \mathsf{h}(\mathsf{a}, \mathsf{b}, \mathsf{a})}{\mathsf{h}(\mathsf{a}, \mathsf{a}, \mathsf{a}) \neq \mathsf{h}(\mathsf{a}, \mathsf{a}, \mathsf{a})}\,\text{parallel superposition}$$ ■

Superposition proofs are represented in a linear format. They are refutational and show how to derive $\perp$ from the input axioms and the negated conjecture.

Example 3. The following is a linear superposition proof from clauses 1–3:

1. $\mathsf{a} = \mathsf{b}$	axiom
2. $\mathsf{f}(x) = x$	axiom
3. $\mathsf{h}(\mathsf{f}(\mathsf{b}), \mathsf{a}) \neq \mathsf{h}(\mathsf{a}, \mathsf{f}(\mathsf{b}))$	negated conjecture
4. $\mathsf{h}(\mathsf{b}, \mathsf{a}) \neq \mathsf{h}(\mathsf{a}, \mathsf{b})$	by parallel superposition from 2 and 3
5. $\mathsf{h}(\mathsf{a}, \mathsf{a}) \neq \mathsf{h}(\mathsf{a}, \mathsf{a})$	by parallel superposition from 1 and 4
6. $\perp$	by equality resolution from 5

■

2.2 Twee and Structured Equational Chain Proofs

Twee is an automatic prover specialized for equational reasoning. It is based on the unfailing completion procedure [3], an extension of Knuth–Bendix completion [18]. In the DISCOUNT and Waldmeister tradition [8], Twee's proofs are structured as a sequence of lemmas, where each lemma and the conjecture are proved by a chain of equalities. Twee introduces lemmas if they are needed more than once. Twee proofs are arguably more readable than Vampire proofs. As with superposition, quantifiers are eliminated by a preprocessor.

Example 4. The following is a Twee-style proof of goal 1 from axioms 1 and 2:

Axiom 1: $\mathsf{a} = \mathsf{b}$

Axiom 2: $\mathsf{f}(x) = x$

Lemma 3: $\mathsf{f}(\mathsf{b}) = \mathsf{a}$
Proof:
$\mathsf{f}(\mathsf{b})$
$=$ { by axiom 1 right-to-left }
$\mathsf{f}(\mathsf{a})$
$=$ { by axiom 2 }
a

Goal 1: $\mathsf{h}(\mathsf{f}(\mathsf{b}), \mathsf{a}) = \mathsf{h}(\mathsf{a}, \mathsf{f}(\mathsf{b}))$
Proof:
$\mathsf{h}(\mathsf{f}(\mathsf{b}), \mathsf{a})$
$=$ { by lemma 3 }
$\mathsf{h}(\mathsf{a}, \mathsf{a})$
$=$ { by lemma 3 right-to-left }
$\mathsf{h}(\mathsf{a}, \mathsf{f}(\mathsf{b}))$ ■

3 Proof Redirection

Vampire generates proofs by refutation, whereas our mix-and-match approach requires direct proofs. To bridge this gap, we transform Vampire proofs into direct proofs. In the following sections, we will always use direct proofs.

To redirect a proof by refutation in equational logic, we first introduce $\exists$ quantifiers for Skolem constants and $\forall$ quantifiers for variables. For example, $\mathsf{h}(x, \mathsf{sk}) \neq x$ is transformed into $\exists z.\, \forall x.\, \mathsf{h}(x, z) \neq x$. Then we apply the contrapositive to all inferences in which a premise and the conclusion are disequations to obtain positive equations. Thus, the inference

$$\frac{\mathsf{h}(\mathsf{a}, y) = \mathsf{b} \qquad \mathsf{h}(x, \mathsf{sk}) \neq x}{\mathsf{b} \neq \mathsf{a}}\text{superposition}$$

becomes

$$\frac{\forall y.\, \mathsf{h}(\mathsf{a}, y) = \mathsf{b} \qquad \mathsf{b} = \mathsf{a}}{\forall z.\, \exists x.\, \mathsf{h}(x, z) = x}$$

Equality resolution inferences from a premise $t \neq t$ are omitted since their contrapositives derive trivial equations.

Example 5. The following is a direct proof obtained from Example 3's proof by refutation.

1. $\mathsf{a} = \mathsf{b}$	axiom
2. $\forall x.\, \mathsf{f}(x) = x$	axiom
3. $\mathsf{h}(\mathsf{b}, \mathsf{a}) = \mathsf{h}(\mathsf{a}, \mathsf{b})$	from 1 and $\mathsf{h}(\mathsf{a}, \mathsf{a}) = \mathsf{h}(\mathsf{a}, \mathsf{a})$
4. $\mathsf{h}(\mathsf{f}(\mathsf{b}), \mathsf{a}) = \mathsf{h}(\mathsf{a}, \mathsf{f}(\mathsf{b}))$	from 2 and 3 ■

4 Proof Generation for Intermediate Lemmas

Our approach starts by translating the main theorem into a TPTP [13] input problem and running Vampire to produce an initial proof. This proof is turned into a direct proof, then decomposed into intermediate lemmas. For each lemma, we generate corresponding problems, with the objective of proving them using Vampire and Twee. Three problem variants are generated:

1. *Big-step problems* contain the axioms together with the lemma as the conjecture, and nothing else. This allows us to investigate whether a radically new proof, with different intermediate steps, can be found.
2. *Small-step problems* contain the axioms together with the lemma as the conjecture, and all lemmas derived prior to this lemma in the initial proof as additional axioms. This allows us to investigate whether a somewhat similar variant of the original derivation can be found.
3. *Abstracted problems* are variants of big-step problems that contain the axioms together with an abstracted version of the lemma as the conjecture. Specifically, selected subterms of the lemma—for example, expressions such as $x \diamond y$ that do not contain nested applications—are replaced by fresh variables. This allows us to investigate whether a more general version of the lemma is provable, ideally with a shorter, more abstract proof.

Each problem is submitted to the two provers. If a proof is found for a small-step problem, we expand it to recursively include the shortest proofs of the lemmas used as axioms for the axioms referenced in the proof. Ties are broken arbitrarily. Note that abstracted problems might be unprovable.

Next, we compare the proofs of the three problem variants corresponding to the same lemma. If the abstracted problem has the shortest proof, the lemma it proves is replaced in all small-step problems where it appears as an axiom with the generalized lemma from the abstracted problem. Each updated small-step problem is then re-proved, and if the result has fewer steps, we replace the small-step problem's proof with it.

The length of a Vampire-generated proof is the number of steps of its redirected proof, excluding preprocessing. For Twee, the length of a proof is the cumulative number of equalities in the equality chains. Thus, the Vampire proof in Example 5 has two steps, and the Twee proof in Example 4 has four steps.

5 Proof Construction for the Main Theorem

Based on the intermediate lemmas' proofs generated in the previous phase, our approach constructs a proof of the main theorem. The proof generally consists of three segments. The first segment starts with the axioms and ends with the derivation of a so-called *departure lemma*. The second segment derives a so-called *arrival lemma*. The third segment derives the conjecture. Different candidates are considered as the departure and arrival lemmas, yielding different proofs. The proof with the fewest steps is chosen.

Specifically, we first identify up to six intermediate lemmas that arise close to the end of the initial proof, including the conjecture, and consider them as candidate arrival lemmas. For each of these, we consider its transitive dependencies as candidate departure lemmas. Then, for each candidate departure lemma, we construct a problem with the axioms and the departure lemma's dependencies as the axioms and the departure lemma itself as the conjecture. We run both provers and, if at least one succeeds, we use the shorter result as the proof of the first segment, unless an even shorter proof was generated in the previous phase.

Next, for each pair of candidate departure and arrival lemmas, we generate a new problem with the original axioms, the departure lemma, and its dependencies as axioms and the arrival lemma as the conjecture. We run both provers and, if at least one succeeds, we use the shorter result as the proof of the second segment, unless an even shorter proof was generated earlier. Finally, we generate a new problem with the original axioms, the departure lemma, its dependencies, and the arrival lemma as axioms and the original conjecture as the conjecture. We run both provers and, if at least one succeeds, we use the shorter result as the proof of the third segment, unless an even shorter proof was generated earlier.

Without the separation into segments, proof minimization could be intractable due to combinatorial explosion. We chose to work with three segments as a trade-off between performance and flexibility.

Example 6. Before we review the three-segment proof construction approach in detail, let us look at an example. The following sketch represents an initial seven-step Vampire-generated redirected proof of a theorem $A \Longrightarrow C$:

A	axiom
L_1	from A and A
L_2	from A and L_1
L_3	from L_1 and L_2
L_4	from L_2 and L_3
L_5	from L_3 and L_4
L_6	from A and L_5
C	from L_5 and L_6

Here, A denotes the axiom, and $L_1, \ldots, L_6$ are the lemmas used to derive the conjecture C.

In the first phase, for each lemma $L_1, \ldots, L_6$, we construct big-step, small-step, and abstracted problems and try to prove them using Vampire and Twee, retaining the shortest proof for each lemma. Suppose the following: The shortest proof of L_1 has one step and is obtained from its big-step problem using Vampire; for L_2 and L_3, the shortest proofs are obtained from their small-step problems using Twee; for L_4, the shortest proof is obtained from its abstracted problem using Twee; and for L_5 and L_6, the shortest proofs are obtained from their small-step problems using Vampire.

In the next phase, the last five lemmas, $L_2, \ldots, L_6$, and the conjecture C are considered as candidate arrival lemmas. We focus on L_6. The proof below, found by Vampire for L_6's small-step problem, is the shortest proof for L_6:

A	axiom
L_1	from A and A
L_2	from A and L_1

L_3	from L_1 and L_2
L_4	from L_2 and L_3
L_5	from L_3 and L_4
L_6	from A and L_5

This proof happens to be identical to the first six steps of the initial proof, but in general it could be different.

Next, lemmas L_1 to L_5 are considered as candidate departure lemmas. We focus on L_3. The proof of conjecture C is constructed by concatenating three segments. For the first segment, we create a new problem with A, L_1, and L_2 as axioms, since they are dependencies of the departure lemma L_3 in the above proof of L_6, and L_3 as the conjecture. We run both provers on this problem and obtain a two-step Vampire proof of L_3 from A, L_1, and a new lemma L_2'. Since L_1 is treated as an axiom, we must include its proof to obtain a complete proof of L_3. In the first phase, we found a one-step Vampire proof of L_1 from the axiom A, so we use it. In summary, the proofs of L_1 and L_3 form the first segment, which consists of one step for L_1 and two steps for L_3.

For the second segment, we create a new problem with A, L_1, L_2', and L_3 as axioms and the arrival lemma L_6 as the conjecture. We run both provers on this problem and obtain a two-step Twee proof of L_6 from L_1 and L_3. Together with the first segment, this yields a five-step proof of L_6. Since this proof is shorter than the six-step proof of L_6 presented above, it is used as the second segment.

For the third segment, we create a new problem with A, L_1, L_2', the departure lemma L_3, and the arrival lemma L_6 as axioms and C as the conjecture. We run both provers on this problem and obtain a two-step Twee proof of C from L_2' and L_3. Since this proof does not use the arrival lemma L_6, the second segment is excluded from the result. Concatenating the first and third segments yields a new five-step proof of C:

A	axiom
L_1	from A
L_2'	from A and L_1
L_3	from L_1 and L_2'
C	by a two-step equality chain using L_2' and L_3

Finally, other combinations of candidate departure and arrival lemmas are also considered, and the shortest proof is retained. ■

5.1 Construction of the Dependency Graph

We identify lemmas occurring close to the end of the derivation as candidate arrival lemmas. Different candidates typically depend on substantially different subsets of earlier lemmas. Each candidate therefore induces its own dependency chain, and different choices can lead to substantially different proof lengths. We consider six candidate arrival lemmas extracted from the initial proof, including the conjecture itself, since our approach may produce a shorter proof of the conjecture by reproving it directly from a minimized dependency set.

For every candidate, we build a dependency graph that captures the lemmas required to derive it. Dependencies are determined from the shortest Vampire or Twee proof obtained for each lemma. Given that we generate three problem variants and run two provers, up to six proofs per lemma are considered. A lemma ℓ is considered to directly depend on a lemma ℓ' if the shortest proof of ℓ uses ℓ' as an axiom. Thus, for big-step and abstracted problems, only the original axioms can be dependencies. For small-step problems, each intermediate step in a Vampire proof and each lemma in a Twee proof is considered a lemma.

The dependency graph associated with a candidate arrival lemma is a directed acyclic graph (DAG) whose nodes correspond to lemmas and whose edges express derivability between them. Formally, let V be a finite set of lemmas, each represented by an equation and a set of dependencies on other lemmas. We construct a DAG (V, E), where each vertex $\ell \in V$ corresponds to a lemma and each edge $(\ell, \ell') \in E$ indicates that lemma ℓ directly depends on lemma ℓ'. As an optimization, we merge lemmas that are identical up to the naming of variables, keeping the shortest proof.

5.2 Construction of the First Proof Segment

For each candidate arrival lemma, we investigate whether all lemmas included in its dependency graph are needed to derive it or whether a shorter proof can be obtained by choosing a departure lemma and recomputing parts of the derivation by combining proofs generated by the provers.

As candidate departure lemmas, we consider all lemmas in the DAG. Let ℓ be a candidate departure lemma. If ℓ depends only on the axioms, we take the shortest big-step, small-step, or abstracted proof previously found by Vampire or Twee. Otherwise, we build a problem that includes ℓ's dependencies in the DAG as axioms and the departure lemma as the conjecture, and we run Vampire and Twee. If at least one of them succeeds, we choose the shorter proof as ℓ's proof. This derivation, together with the shortest proofs of ℓ's dependencies generated for the big-step, small-step, or abstracted problems, forms the first segment of the final proof. However, if we found an even shorter proof for the big-step, small-step, or abstracted problem, we use that proof instead. For small-step proofs, we must also include the proofs of the intermediate lemmas encoded as axioms.

5.3 Construction of the Remaining Proof Segments

To construct the second segment, we generate a problem with the departure lemma and its dependencies as axioms and the arrival lemma as the conjecture, and run both provers. If at least one of them succeeds, we choose the shorter proof as the proof of the arrival lemma. As above, we fall back on the proof of a big-step, small-step, or abstracted problem if it is even shorter.

Finally, to construct the third segment, we generate a problem with the departure lemma, the arrival lemma, and their dependencies as axioms and the original conjecture as the conjecture, and invoke both provers. If at least one of them succeeds, we choose the shorter proof as the proof of the original conjecture. As above, we fall back on a previously derived proof if it is even shorter.

The final proof is obtained by concatenating the three segments. The proof might contain unreferenced lemmas; these are pruned.

5.4 Proof Output

Our tool generates the minimized proof in a native format, from which two Lean outputs are produced. The first Lean output is a step-by-step formalization using the `calc` tactic to reconstruct chains of equalities. It applies the `duper` tactic to fill in the subproofs. For example, a proof of $t_1 = t_2 = t_3 = t_4$ would be represented by

```
calc
  t₁ = t₂ := by duper ...
  _ = t₃ := by duper ...
  _ = t₄ := by duper ...
```

where the ellipses stand for `duper`'s arguments. The second Lean output is a more compact Lean formalization in which each lemma is proved directly using Lean's automation without including the intermediate steps in chains of equalities.

6 Application to Tao's Challenge

We implemented our approach and tried the resulting tool, Krympa, on Tao's challenge theorem 650 $\Longrightarrow$ 448:

$$(\forall x, y, z.\ x = x \diamond (y \diamond ((z \diamond x) \diamond y))) \Longrightarrow \forall x, y, z.\ x = x \diamond (y \diamond (z \diamond (x \diamond z))).$$

Our tool first ran Vampire to obtain an initial 62-step superposition proof. Then it constructed 62 problems of each variant (big-step, small-step, and abstracted) and tried to prove them using Vampire and Twee. Among the six candidate arrival lemmas, the shortest proof was found by selecting

$$\forall x, y, z.\ x = x \diamond ((y \diamond ((z \diamond y) \diamond y)) \diamond x). \qquad \text{(lemma 9)}$$

The coloring highlights repeating patterns. Next, our tool constructed the dependency graph for this lemma. The DAG contained 37 lemmas. It was based on big- and small-step proofs.

Among the 37 candidate departure lemmas, our tool found the shortest proof by selecting

$$\begin{aligned}&\forall x, y, z, w.\ (x \diamond ((y \diamond x) \diamond x)) \diamond z = \\ &\quad ((x \diamond ((y \diamond x) \diamond x)) \diamond z) \diamond (w \diamond ((x \diamond ((y \diamond x) \diamond x)) \diamond w)).\end{aligned} \qquad \text{(lemma 7)}$$

According to the DAG, the shortest proof of this lemma was found by running Vampire on the small-step problem consisting of the axiom and the following lemma dependencies:

$$\forall x,y,z,w.\ x \diamond ((y \diamond z) \diamond x) = (x \diamond ((y \diamond z) \diamond x)) \diamond (w \diamond (z \diamond w)) \quad \text{(lemma 1)}$$

$$\forall x,y,z,w,v,u.\ x \diamond ((y \diamond ((z \diamond w) \diamond y)) \diamond x) = (x \diamond ((y \diamond ((z \diamond w) \diamond y)) \diamond x)) \diamond (v \diamond ((u \diamond (w \diamond u)) \diamond v)) \quad \text{(lemma 2)}$$

$$\forall x,y,z,w,v.\ x \diamond (y \diamond x) = (x \diamond (y \diamond x)) \diamond (z \diamond ((w \diamond ((v \diamond y) \diamond w)) \diamond z)) \quad \text{(lemma 3)}$$

$$\forall x,y,z,w,v.\ x \diamond (y \diamond x) = (x \diamond (y \diamond x)) \diamond ((z \diamond (y \diamond z)) \diamond (w \diamond ((v \diamond y) \diamond w))) \quad \text{(lemma 4)}$$

$$\forall x,y,z,w.\ x \diamond ((y \diamond ((z \diamond y) \diamond y)) \diamond x) = (x \diamond ((y \diamond ((z \diamond y) \diamond y)) \diamond x)) \diamond (w \diamond ((y \diamond ((z \diamond y) \diamond y)) \diamond w)) \quad \text{(lemma 5)}$$

$$\forall x,y,z,w.\ (x \diamond ((y \diamond x) \diamond x)) \diamond z = ((x \diamond ((y \diamond x) \diamond x)) \diamond z) \diamond ((w \diamond ((x \diamond ((y \diamond x) \diamond x)) \diamond w)) \diamond (z \diamond ((x \diamond ((y \diamond x) \diamond x)) \diamond z))). \quad \text{(lemma 6)}$$

Following the inference steps of the initial Vampire proof, our tool derived lemma 1 by applying a superposition inference with the axiom $x = x \diamond (y \diamond ((z \diamond x) \diamond y))$ as the first premise and a renamed copy $x' = x' \diamond (y' \diamond ((z' \diamond x') \diamond y'))$ as the second premise. The most general unifier of the first premise's right-hand side and the subterm $z' \diamond x'$ of the second premise is $\{x' \mapsto y \diamond ((z \diamond x) \diamond y),\ z' \mapsto x\}$. Applying the unifier to both premises yields the equations $x = x \diamond (y \diamond ((z \diamond x) \diamond y))$ and $y \diamond ((z \diamond x) \diamond y) = (y \diamond ((z \diamond x) \diamond y)) \diamond (y' \diamond ((x \diamond (y \diamond ((z \diamond x) \diamond y))) \diamond y'))$. The superposition inference replaced the subterm $x \diamond (y \diamond ((z \diamond x) \diamond y))$ in the second equation with x using the first equation as a right-to-left rewrite rule, and thus derived lemma 1, up to the naming of variables. Lemmas 2 to 7 were derived similarly following the steps of the initial Vampire proof.

Next, from the axiom and lemma 7, our tool proved the arrival lemma (lemma 9) using Twee. For this proof, Twee introduced the auxiliary lemma

$$\forall x,y,z,w.\ (y \diamond ((z \diamond y) \diamond y)) \diamond w = ((y \diamond ((z \diamond y) \diamond y)) \diamond w) \diamond ((y \diamond ((z \diamond y) \diamond y)) \diamond x). \quad \text{(lemma 8)}$$

Finally, assuming all the lemmas derived so far, our tool proved the conjecture from lemmas 5 and 9 using Twee. The resulting proof has 20 steps, including three Twee-generated chains of equalities.

Below we present the final proof adapted from our tool's detailed Lean output. Instead of relying on proof automation, we use the `nth_rw` tactic, which performs a single rewrite step, where the numeric argument indicates which matching occurrence should be rewritten. In one case, two numbers are supplied, corresponding to a parallel rewrite.

```
class Magma (α : Type _) where
  op : α → α → α

infix:65 " ◇ " => Magma.op
```

```
theorem Equation650_implies_Equation448 (G : Type _) [Magma G]
      (op_law : ∀ x y z : G, x = x ◇ (y ◇ ((z ◇ x) ◇ y))) :
    ∀ x y z : G, x = x ◇ (y ◇ (z ◇ (x ◇ z))) :=
  have lemma1 (x y z w : G) :
      x ◇ ((y ◇ z) ◇ x) = (x ◇ ((y ◇ z) ◇ x)) ◇ (w ◇ (z ◇ w)) := by
    nth_rw 3 [op_law z x y]
    exact op_law (x ◇ ((y ◇ z) ◇ x)) w z

  have lemma2 (x y z w v u : G) :
      x ◇ ((y ◇ ((z ◇ w) ◇ y)) ◇ x) =
      (x ◇ ((y ◇ ((z ◇ w) ◇ y)) ◇ x)) ◇ (v ◇ ((u ◇ (w ◇ u)) ◇ v)) := by
    nth_rw 1 2 [lemma1 y z w u]
    exact lemma1 x (y ◇ ((z ◇ w) ◇ y)) (u ◇ (w ◇ u)) v

  have lemma3 (x y z w v : G) :
      x ◇ (y ◇ x) = (x ◇ (y ◇ x)) ◇ (z ◇ ((w ◇ ((v ◇ y) ◇ w)) ◇ z)) := by
    nth_rw 1 [lemma1 w v y x]
    exact op_law (x ◇ (y ◇ x)) z (w ◇ ((v ◇ y) ◇ w))

  have lemma4 (x y z w v : G) :
      x ◇ (y ◇ x) = (x ◇ (y ◇ x)) ◇ ((z ◇ (y ◇ z)) ◇ (w ◇ ((v ◇ y) ◇ w))) := by
    nth_rw 1 [lemma1 w v y z]
    exact lemma3 x y (z ◇ (y ◇ z)) w v

  have lemma5 (x y z w : G) :
      x ◇ ((y ◇ ((z ◇ y) ◇ y)) ◇ x) =
      (x ◇ ((y ◇ ((z ◇ y) ◇ y)) ◇ x)) ◇ (w ◇ ((y ◇ ((z ◇ y) ◇ y)) ◇ w)) := by
    nth_rw 1 [lemma2 w y z y x ((z ◇ y) ◇ y)]
    exact lemma4 x (y ◇ ((z ◇ y) ◇ y)) w x ((z ◇ y) ◇ y)

  have lemma6 (x y z w : G) :
      (x ◇ ((y ◇ x) ◇ x)) ◇ z =
      ((x ◇ ((y ◇ x) ◇ x)) ◇ z) ◇ ((w ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ w)) ◇
        (z ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ z))) := by
    nth_rw 1 [lemma5 z x y w]
    exact op_law ((x ◇ ((y ◇ x) ◇ x)) ◇ z) (w ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ w)) z

  have lemma7 (x y z w : G) :
      (x ◇ ((y ◇ x) ◇ x)) ◇ z =
      ((x ◇ ((y ◇ x) ◇ x)) ◇ z) ◇ (w ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ w)) := by
    nth_rw 1 [lemma5 w x y z]
    exact lemma6 x y z w

  have lemma8 (x y z w : G) :
    ((x ◇ ((y ◇ x) ◇ x)) ◇ z) ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ w) =
    (x ◇ ((y ◇ x) ◇ x)) ◇ z := by
    let T := x ◇ ((y ◇ x) ◇ x)
    calc
      (T ◇ z) ◇ (T ◇ w) =
      ((T ◇ z) ◇ ((T ◇ w) ◇ ((T ◇ (T ◇ w)) ◇ ((w ◇ (T ◇ w)) ◇ (T ◇ (T ◇ w)))))) := by
```

```
          nth_rw 1 [←op_law]
        _ = ((T ◇ z) ◇ ((T ◇ w) ◇ ((T ◇ (T ◇ w)) ◇ ((w ◇ (T ◇ w)) ◇
              (T ◇ ((T ◇ w) ◇ (w ◇ (T ◇ w)))))))) := by
          nth_rw 1 [←lemma7]
        _ = ((T ◇ z) ◇ ((T ◇ w) ◇ ((T ◇ (T ◇ w)) ◇ ((w ◇ (T ◇ w)) ◇
              ((T ◇ ((T ◇ w) ◇ (w ◇ (T ◇ w)))) ◇ (((T ◇ w) ◇ (w ◇ (T ◇ w))) ◇
                (T ◇ ((T ◇ w) ◇ (w ◇ (T ◇ w)))))))))) := by
          nth_rw 2 [←lemma7]
        _ = ((T ◇ z) ◇ ((T ◇ w) ◇ ((T ◇ (T ◇ w)) ◇ (w ◇ (T ◇ w))))) := by
          nth_rw 1 [←op_law]
        _ = ((T ◇ z) ◇ ((T ◇ w) ◇ (T ◇ (T ◇ w)))) := by
          nth_rw 1 [←lemma7]
        _ = ((x ◇ ((y ◇ x) ◇ x)) ◇ z) := by
          nth_rw 1 [←lemma7]

  have lemma9 (x y z : G) :
      (x ◇ ((y ◇ ((z ◇ y) ◇ y)) ◇ x)) = x := by
    calc
      (x ◇ ((y ◇ ((z ◇ y) ◇ y)) ◇ x)) =
      (x ◇ (((y ◇ ((z ◇ y) ◇ y)) ◇ x) ◇ ((y ◇ ((z ◇ y) ◇ y)) ◇ x))) := by
        nth_rw 1 [lemma8]
      _ = (x ◇ (((y ◇ ((z ◇ y) ◇ y)) ◇ x) ◇ (((y ◇ ((z ◇ y) ◇ y)) ◇ x) ◇
            ((y ◇ ((z ◇ y) ◇ y)) ◇ x)))) := by
        nth_rw 2 [lemma8]
      _ = x := by
        nth_rw 1 [←op_law]

  show _ by
    intros x y z
    calc
      x = x ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ x) := by
        nth_rw 1 [lemma9]
      _ = (x ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇ x)) ◇ ((y ◇ (z ◇ (x ◇ z))) ◇
            ((x ◇ ((y ◇ x) ◇ x)) ◇ (y ◇ (z ◇ (x ◇ z))))) := by
        nth_rw 1 [←lemma5]
      _ = x ◇ ((y ◇ (z ◇ (x ◇ z))) ◇ ((x ◇ ((y ◇ x) ◇ x)) ◇
            (y ◇ (z ◇ (x ◇ z))))) := by
        nth_rw 1 [lemma9]
      _ = x ◇ (y ◇ (z ◇ (x ◇ z))) := by
        nth_rw 1 [lemma9]
```

7 Experiments on Other Equational Proofs

To assess the general potential of our approach, we evaluated our tool on a set of equational theorems obtained from the Equational Theories Project repository [7]. We selected all problems in the 13 Lean files `Proofs1` to `Proofs13` that have a proof and translated them to TPTP problem files, yielding 1431 benchmarks.

For each file, we invoked our tool's TPTP problem generator, which parses the Lean theorems and produces corresponding TPTP problem files. For each problem, our tool was given 2700 seconds to produce a minimized proof using Vampire to find the initial proof and Vampire and Twee to find subproofs; on failure, the initial Vampire proof was output. A time limit of 10 seconds was used for each prover invocation. The experiments were conducted on a server equipped with a dual-socket AMD EPYC 9965 system (384 cores, 768 threads) running at 2.25–3.70 GHz with 3 TiB of DDR5 ECC RAM, and running Debian GNU/Linux 13 (kernel 6.17.13+deb13-amd64).

Overall, proofs for the 13 Lean files have an average length of 6.6 steps before minimization and 4.5 steps after minimization using the combination of small-step and abstracted problems and both provers. This corresponds to a 31.5% decrease, showing that even short proofs can often be made shorter.

Since longer proofs present more opportunities for minimization, we now focus on problems whose initial proofs have at least 15 steps. Table 1 compares proof lengths before and after minimization. The "Avg. before" column shows the average number of inference steps in the initial Vampire proofs. The remaining columns report the average proof length after minimization under four configurations, which differ in which problem variants are used to generate candidate lemmas: "BA" denotes the combination of the big-step and abstracted variants; "SA" denotes the combination of the small-step and abstracted variants; "BS" denotes the combination of the big- and small-step variants; and "BSA" denotes the combination of all three variants.

The results show an often substantial reduction in proof length. SA generally yielded the shortest proofs. Across all problems for the 13 Lean files, the average reduction with SA is 56.7%. BS and BSA also produced substantial reductions, whereas BA generally yielded the least improvements.

Table 1. Comparison of proof lengths before and after minimization for problems with initial proofs of at least 15 steps

File	Num. problems	Avg. before	Avg. after			
			BA	SA	BS	BSA
`Proofs1`	135	17.3	16.0	**13.1**	13.3	13.3
`Proofs2`	117	16.9	14.4	**11.5**	**11.5**	**11.5**
`Proofs3`	108	19.3	15.6	10.9	**10.7**	10.9
`Proofs4`	125	19.1	14.3	**10.9**	11.4	11.4
`Proofs5`	116	20.1	17.6	12.8	**11.9**	**11.9**
`Proofs6`	115	25.6	18.9	**12.5**	12.6	12.6
`Proofs7`	117	37.2	19.7	**11.8**	11.9	11.9
`Proofs8`	114	24.4	15.6	**12.3**	13.1	13.1
`Proofs9`	112	39.8	29.0	**13.1**	14.1	14.1
`Proofs10`	101	21.5	16.0	**8.0**	11.0	11.0
`Proofs11`	110	25.4	22.4	**13.0**	14.0	14.0
`Proofs12`	123	24.6	16.5	**8.0**	8.5	8.5
`Proofs13`	38	35.3	27.7	**9.1**	10.1	10.1

It might seem counterintuitive that SA, which does not consider big-step problems, outperforms BSA. However, the nonmonotonicity is to be expected. Provers are nondeterministic, especially when invoked with a time limit. More importantly, our approach makes different heuristic choices when constructing the three proof segments depending on which problem variants are used. As a result, SA might find a short proof that escapes BSA.

The reduction in proof length is especially noticeable in individual cases. The problem $2666 \Longrightarrow 3460$ has a Vampire proof with 51 inference steps, which our tool reduces to only 12 single rewrite steps, and $2923 \Longrightarrow 2628$ is reduced from 180 steps to only 34. The problem $3569 \Longrightarrow 3957$ is reduced from 92 to 23 steps and, even more dramatically, $3957 \Longrightarrow 3971$ is reduced from 141 steps to only 23. Furthermore, $2860 \Longrightarrow 2660$ is reduced from 44 to 14 steps, and $723 \Longrightarrow 872$ goes from 57 to 13 steps. Finally, $947 \Longrightarrow 3897$ underwent the largest reduction, from 151 to 10 steps. Overall, these results demonstrate that our approach produces shorter proofs across a diverse set of equational theorems.

8 Related Work

At least two other researchers took on Tao's challenge. Kinyon [17] found a 24-step proof (excluding preprocessing) of $650 \Longrightarrow \forall x, y.\ x = x \diamond y$ using Prover9 [21], from which $650 \Longrightarrow 448$ follows by instantiation. Later, Le Floch [11] developed a pen-and-paper proof and translated it to Lean. The Lean proof relies on only 14 rewrite steps but includes additional reasoning as proof terms, and two of the rewrite steps are parallel, so the overall length is similar to ours. The proof idea is "loosely based" on the output of multiple Prover9 runs "with intermediate results thrown in as assumptions or as goals"—in essence, a manual approximation of our approach. Also in the context of the Equational Theories Project, Janota [15] evaluated Vampire on the project's problems and showed that combining superposition with finite model finding can solve almost all problems.

We are aware of little work on automated proof minimization. Stachniak [25] designed an algorithm for constructing resolution proofs in propositional logics known as strongly finite logics. Amjad [2] and Cotton [10] introduced techniques for minimizing propositional resolution proofs. Vyskočil et al. [29] proposed to compress proofs by inventing new definitions using a heuristics based on substitution trees. Gu et al. [14] developed ProofOptimizer, which uses large language models to simplify Lean proofs.

Some SAT (satisfiability) and SMT (satisfiability modulo theories) solvers can minimize the number of axioms needed for a proof, but the result can be a longer proof. SAT solvers commonly interleave search, which can be expressed as resolution steps, with formula-rewriting techniques that go beyond resolution. This interleaving, known as inprocessing [16], is highly effective and often yields both faster solving times and shorter proofs than either approach in isolation.

The idea of automatically mixing and matching proofs is not new. Sutcliffe et al. [27] introduced a method for combining automatically generated proofs to generate new ones. Their proofs are represented as DAGs, enabling the identification and replacement of subproofs across different proofs. Proof combination is guided by heuristics that measure structural similarity, and a greedy search strategy is used to explore alternative combinations that yield proofs differing from the originals. In contrast to our approach, the main objective is to increase proof diversity rather than minimize proof length.

9 Conclusion

Historically, more research has gone into finding proofs automatically than into improving and presenting them. We introduced an approach for minimizing equational proofs by mixing and matching the output of separate runs of Vampire and Twee, and implemented it in a new tool, Krympa. We used the tool to minimize the proof of problem $650 \Longrightarrow 448$ from the Equational Theories Project from 62 to 20 steps, thereby providing a fully automatic solution to a challenge posed by Tao. We also obtained remarkable reductions on other problems originating from the project. The shorter proofs are arguably easier to understand by humans and sometimes more general. Our work shows that proof automation and readability can go hand in hand.

Our approach could be extended in several ways. First, it could be generalized to support full first- or higher-order logic. Second, alternative lemma abstraction strategies could be explored. Third, proofs with more than three segments could be synthesized. Fourth, we might want to consider not only the number of steps but also term size when measuring proofs, as suggested by Le Floch [12]. Fifth, we could try to translate Vampire's superposition steps to Twee's structured equality chain format.

Some possible extensions specifically concern the implementation. First, we could, following a private suggestion by Martin Suda, explore whether nondefault Vampire strategies can produce shorter proofs. Second, since proof generation relies heavily on external provers, performance could benefit from better scheduling of prover invocations, using adaptive time limits. Finally, as the number of possible lemma combinations grows rapidly, exploiting parallelism at multiple levels—such as lemma re-proving, dependency graph construction, and proof construction—would be a natural extension of the current architecture.

Acknowledgments. We thank Bruno Le Floch, Andrew Reynolds, Stephan Schulz, and Uwe Waldmann for fruitful discussions. We thank Martin Suda and Mark Summerfield for helpful textual suggestions.

Blanchette's research was cofunded by the European Union (ERC, Nekoka, 101083038). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Heule's research is supported by the NSF under grant DMS-2434625 and funding from AFRL and DARPA under Agreement FA8750-24-9-1000.

References

1. The Lean Language Reference (2025), https://lean-lang.org/doc/reference/latest/
2. Amjad, H.: Compressing propositional refutations. In: Merz, S., Nipkow, T. (eds.) AVoCS 2006. Electronic Notes in Theoretical Computer Science, vol. 185, pp. 3–15. Elsevier (2006)
3. Bachmair, L., Dershowitz, N., Plaisted, D.A.: Completion without failure. In: Aït-Kaci, H., Nivat, M. (eds.) Rewriting Techniques, pp. 1–30. Academic Press (1989)
4. Bachmair, L., Ganzinger, H.: Strict basic superposition. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS, vol. 1421, pp. 160–174. Springer (1998)
5. Bártek, F., Bhayat, A., Coutelier, R., Hajdú, M., Hetzenberger, M., Hozzová, P., Kovács, L., Rath, J., Rawson, M., Reger, G., Suda, M., Schoisswohl, J., Voronkov, A.: The Vampire diary. In: Piskac, R., Rakamaric, Z. (eds.) CAV 2025. LNCS, vol. 15933, pp. 57–71. Springer (2025)
6. Blanchette, J.C., Böhme, S., Fleury, M., Smolka, S.J., Steckermeier, A.: Semi-intelligible Isar proofs from machine-generated proofs. J. Automated Reas. **56**, 155–200 (2016)
7. Bolan, M., Breitner, J., Brox, J., Carlini, N., Carneiro, M., van Doorn, F., Dvorak, M., Goens, A., Hill, A., Husum, H., Mejia, H.I., Kocsis, Z.A., Floch, B.L., Bar-on, A.L., Luccioli, L., McNeil, D., Meiburg, A., Monticone, P., Nielsen, P., Osazuwa, E.O., Paolini, G., Petracci, M., Reinke, B., Renshaw, D., Rossel, M., Roux, C., Scanvic, J., Srinivas, S., Tadipatri, A.R., Tao, T., Tsyrklevich, V., Vaquerizo-Villar, F., Weber, D., Zheng, F.: The Equational Theories Project (2025)
8. Buch, A., Hillenbrand, T.: WALDMEISTER: Development of a high performance completion-based theorem prover (1996)
9. Clune, J., Qian, Y., Bentkamp, A., Avigad, J.: Duper: A proof-producing superposition theorem prover for dependent type theory. In: Bertot, Y., Kutsia, T., Norrish, M. (eds.) ITP 2024. LIPIcs, vol. 309, pp. 1–20. Leibniz-Zentrum für Informatik (2024)
10. Cotton, S.: Two techniques for minimizing resolution proofs. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6175, pp. 306–312. Springer (2010)
11. Floch, B.L.: Zulip post in “Machine Learning for Theorem Proving: A (Semi)-Autoformalization Challenge (650 $\rightarrow$ 448)”. Zulip (2025), available at https://leanprover.zulipchat.com/#narrow/channel/219941-Machine-Learning-for-Theorem-Proving/topic/A.20.28semi.29-autoformalization.20challenge.3A.20650.3D.3E448/near/518970125
12. Floch, B.L.: Zulip post in “Machine Learning for Theorem Proving: A (Semi)-Autoformalization Challenge (650 $\rightarrow$ 448)”. Zulip (2025), available at https://leanprover.zulipchat.com/#narrow/channel/219941-Machine-Learning-for-Theorem-Proving/topic/A.20.28semi.29-autoformalization.20challenge.3A.20650.3D.3E448/near/568313777
13. Geoff Sutcliffe: Stepping stones in the TPTP World. In: Benzmüller, C., Heule, M., Schmidt, R. (eds.) IJCAR 2024. pp. 30–50. LNCS (2024)
14. Gu, A., Piotrowski, B., Gloeckle, F., Yang, K., Markosyan, A.H.: ProofOptimizer: Training language models to simplify proofs without human demonstrations. CoRR **abs/2510.15700** (2025)

15. Janota, M.: Experimental results for Vampire on the Equational Theories Project (2025)
16. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 355–370. Springer (2012)
17. Kinyon, M.: Zulip post in "Machine Learning for Theorem Proving: A (Semi)-Autoformalization Challenge (650 → 448)". Zulip (2025), available at `https://leanprover.zulipchat.com/#narrow/channel/219941-Machine-Learning-for-Theorem-Proving/topic/A.20.28semi.29-autoformalization.20challenge.3A.20650.3D.3E448/near/518961204`
18. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press (1970)
19. Kondylidou, L., Blanchette, J., Heule, M.: Tao's equational proof challenge accepted. Zenodo (2026), `https://doi.org/10.5281/zenodo.18624123`
20. Limperg, J., From, A.H.: Aesop: White-box best-first proof search for Lean. In: CPP 2023. pp. 253–266. Association for Computing Machinery (2023)
21. McCune, W.: Prover9 and Mace4 (2005–2010)
22. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Bertot, Y., Kutsia, T., Norrish, M. (eds.) CADE 2021. LNCS, vol. 12699, pp. 625–635. Springer (2021)
23. Norman, C., Avigad, J.: Canonical for automated theorem proving in Lean. In: ITP 2025. LIPIcs, vol. 352, pp. 1–20. Leibniz-Zentrum für Informatik (2025)
24. Smallbone, N.: Twee: An equational theorem prover. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS, vol. 12699, pp. 602–613. Springer (2021)
25. Stachniak, Z.: Minimization of resolution proof systems. Fundam. Informaticae **14**(1), 129–146 (1991)
26. Sutcliffe, G.: The 12th IJCAR Automated Theorem Proving System Competition—CASC-J12. AI Communications **38**, 3–20 (2025)
27. Sutcliffe, G., Chang, C., McGuinness, D., Lebo, T., Ding, L., da Silva, P.P.: Combining proofs to form different proofs. In: Fontaine, P., Stump, A. (eds.) PxTP 2011. pp. 60–73. LNCS (2011)
28. Tao, T.: Machine learning for theorem proving: A (semi)-autoformalization challenge (650 → 448). `https://leanprover-community.github.io/archive/stream/219941-Machine-Learning-for-Theorem-Proving/`, Lean Zulip thread, created May 16, 2025
29. Vyskočil, J., Stanovský, D., Urban, J.: Automated proof compression by invention of new definitions. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNCS, vol. 6355, pp. 447–462. Springer (2010)
30. Zhu, T., Clune, J., Avigad, J., Jiang, A.Q., Welleck, S.: Premise selection for a Lean hammer. arXiv preprint arXiv:2506.07477 (2025)