lean-smt: An SMT tactic for discharging proof goals in Lean

Abdalrhman Mohamed[1], Tomaz Mascarenhas[4], Harun Khan[1], Haniel Barbosa[4], Andrew Reynolds[2,3], Yicheng Qian[1], Cesare Tinelli[2], and Clark Barrett[1]

[1] Stanford University, Stanford, USA
[2] The University of Iowa, Iowa City, USA
[3] Amazon Web Services, Seattle, WA, USA
[4] Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

Abstract. Lean is an increasingly popular proof assistant based on dependent type theory. Despite its success, it still lacks important automation features present in more seasoned proof assistants, such as the Sledgehammer tactic in Isabelle/HOL. A key aspect of Sledgehammer is the use of proof-producing SMT solvers to prove a translated proof goal and the reconstruction of the resulting proof into valid justifications for the original goal. We present lean-smt, a tactic providing this functionality in Lean. We detail how the tactic converts Lean goals into SMT problems and, more importantly, how it reconstructs SMT proofs into native Lean proofs. We evaluate the tactic on established benchmarks used to evaluate Sledgehammer's SMT integration, with promising results. We also evaluate lean-smt as a standalone proof checker for proofs of SMT-LIB problems. We show that lean-smt offers a smaller trusted core without sacrificing too much performance.

1 Introduction

Proof assistants, also known as interactive theorem provers (ITPs), allow users to write mechanized proofs of statements written in a formal language, whose validity can be verified by a small, trusted kernel. They help users construct trustworthy, formal, machine-checkable proofs of theorems, and have been increasingly used to mechanize proofs of various mathematical results [18, 19]. This process has significantly accelerated in recent years with the adoption of the Lean 4 proof assistant [28] by leading members of the mathematical community [12, 14, 37]. Proof assistants are also commonly used in certain areas of computer science to model and formally verify systems, thanks to the high expressiveness of their underlying language and logic [23, 29].

The trustworthiness of proof assistants relies on the kernel correctly verifying every proof step. For this reason, ITP kernels are designed to be simple and small, implementing just straightforward logical operations from the logical framework underlying the proof assistant. This means that, in principle, each proof step must be explicitly formulated by the user, with the consequence that a naive use of ITPs may require a prodigious amount of expertise and effort. A major challenge, then, is the extension of the kernel with trustworthy facilities for automating the writing of mechanized proofs as much as possible, thereby reducing the burden on users.[5] Automation is generally achieved via *tactics*, proof-producing algorithms, traditionally written in a special-purpose language, that discharge proof obligations for certain classes of subgoals, often by simulating common proof techniques, such as case analysis or induction.

An alternative is to use external automatic theorem provers (ATPs) to solve subgoals when possible. Tools such as HOLYHammer [22], Miz$\mathbb{AR}$ [21], Sledgehammer [26], and Why3 [10], provide a one-click connection from proof assistants to first-order provers and have led to considerable improvements in proof-assistant automation [9]. Sledgehammer, a particularly successful tactic in Isabelle/HOL [30], includes an integration with proof-producing SMT solvers [8, 36]. Sledgehammer translates certain proof goals into SMT-LIB [6], a standard format for SMT problems, and sends them to a supported SMT solver. The proof returned by the solver is reconstructed by essentially reproducing each step within the proof assistant. The integration of SMT solvers in Sledgehammer has been especially useful for proof goals arising from formal verification efforts carried out in Isabelle/HOL [8]. We conjecture that any tactic in Lean with the same ambition and expected impact as Sledgehammer will require a similar level of SMT solver integration.

In this paper, we present LEAN-SMT,[6] which provides an initial implementation of a similar integration in Lean and can be seen as a stepping stone towards a full Sledgehammer-like tactic for Lean. LEAN-SMT operates by translating Lean proof goals expressible in the first-order logic fragment of dependent type theory (Section 3.2) into SMT problems, leveraging SMT theories to model corresponding elements from Lean, such as uninterpreted functions, propositional equality, first-order quantifiers, and arithmetic operators. To bridge the gap between this restricted fragment and the goals that arise in practice in Lean formalizations, we rely on LEAN-AUTO, a tactic developed by Qian et al. [34], to reduce Lean proof goals to first-order logic, together with dedicated preprocessing in LEAN-SMT itself to express them in the language of selected SMT theories (Section 3.1). LEAN-SMT currently supports the state-of-the-art proof-producing SMT solver CVC5 [3]. CVC5's extensive proof production capabilities [4] and strong performance on the fragment of interest make it well suited for such an integration. CVC5's proofs are reconstructed into native Lean proofs (Section 3.3) by using either a Lean theorem, a Lean tactic, or a formally verified Lean program.

We evaluate LEAN-SMT (Section 4) on proof goals from a standard benchmark set used to evaluate Sledgehammer, showing that LEAN-SMT performs comparably with Sledgehammer's SMT integration. We also evaluate it as a verified proof checker for SMT proofs. Although, as expected, LEAN-SMT is less performant

[5] In mathematics, the cost of mechanizing a proof is currently estimated to be $\sim$ 20x the original cost of writing the proof [37]. Using ITPs to formally verify large systems is well known to be very costly, in the range of multiple person-years [23].

[6] LEAN-SMT is available online at https://github.com/ufmg-smite/lean-smt

than standalone, unverified SMT proof checkers, its performance is generally within an order of magnitude of theirs. Additionally, it offers comparable performance to SMTCoq [16], a similarly verified checker for Rocq, while supporting a larger logical fragment.

2 Related Work

While our main inspiration for LEAN-SMT has been the SMT integration in Sledgehammer, there are other ways in which Sledgehammer leverages the power of automatic theorem provers. Sledgehammer includes a premise selection module, which filters lemmas from Isabelle's libraries that are potentially useful for proving the goal. These lemmas are added to the problem to be sent to the solver. An alternative strategy to directly reconstructing a proof returned by the SMT solver is to simply identify the input lemmas actually used in the proof and pass them, together with the original goal, to *metis* [20], a proof-producing superposition-based theorem prover written as an Isabelle tactic. While there is no equivalent for Sledgehammer in Lean, there is a growing ecosystem of tools that eventually can be combined into a comparable tool. Recently Aesop [25], a tableaux-based prover, and Duper [13] which is, like *metis*, a superposition-based prover, were introduced as proof-producing Lean tactics. No equivalent for the premise selection mechanism is readily available in Lean yet, although there has been initial work in this direction [32].

Another integration between a proof assistant and SMT solvers is offered by SMTCoq [16], a plug-in for the Rocq proof assistant [7]. It supports proof reconstruction for the SMT solvers veriT [11] and CVC4 [5], but rather than replaying each individual proof step within Rocq, as Sledgehammer and LEAN-SMT do, it applies a formally verified checker that, if successful, confirms the original proof goal as a theorem. This approach relies heavily on the efficiency of the proof assistant itself, since the proof checker runs within the proof assistant and may have to analyze and simplify huge proof terms. The Rocq proof assistant is designed to be very fast at this; on the other hand, Lean is not [2]. Moreover, the verified checker approach can be rather rigid, since any modification to the supported proof format requires the checker's correctness theorem to be proven again. This can require a significant effort, even for tools with a highly modular architecture like SMT-Coq. In contrast, in the *proof replay* approach, one needs to change the reconstruction tactic by modifying the step corresponding to the changed element of the format. These observations motivated our decision to use proof replay in LEAN-SMT. The decision has been crucial for its development so far since CVC5's proof calculus and infrastructure are still evolving [4, 24, 31].

3 System Overview

Figure 1 depicts the LEAN-SMT architecture, which takes as input a Lean goal, whose type is represented as the formula F, and generates a proof for it by solving a corresponding unsatisfiability problem in SMT and reconstructing its proof into a Lean proof for F. The arrows illustrate the tactic's pipeline, through which the

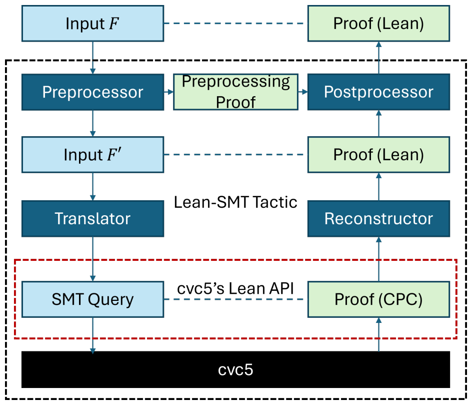


Fig. 1: Architecture of the LEAN-SMT tactic.

input formula is simplified, translated into an SMT query, and sent to CVC5. The solver's proof is then reconstructed as a proof in Lean for the input formula. The light green boxes represent the proofs used during the reconstruction stage, which correspond directly to the formulas (light blue) processed during the translation stage. The dashed lines emphasize this correspondence.

The *preprocessor* module converts the initial input F into an intermediate form F', simplifying or restructuring the input to make it more suitable for translation into an SMT query. During this phase, a preprocessing proof is generated, capturing the transformations applied. The *translator* module generates a formula in SMT-LIB format whose unsatisfiability corresponds to the validity of F'. The formula is passed to the CVC5 solver which produces a proof for it if it determines it to be unsatisfiable. The proof is expressed in the Cooperating Proof Calculus (CPC).[7] The tactic interfaces with CVC5 through Lean's Foreign Function Interface (FFI) and the solver's Lean API, which we added to the solver to facilitate the integration. The *reconstructor* module translates the CPC proof into a Lean proof of F', mapping the CPC proof structure to corresponding Lean constructs and ensuring logical equivalence. The *postprocessor* combines the preprocessing proof and the reconstruction proof into a single Lean proof for the original formula F, which is then checked by Lean's kernel.

3.1 Preprocessing Original Lean Goal

Lean's type system, rooted in dependent type theory (DTT) [1], is far more expressive than the many-sorted first-order logic (FOL) [17] used by SMT solvers. To bridge this gap, we employ proof-producing preprocessing steps that simplify Lean goals into a form more amenable to translation into SMT-LIB.

[7] See https://cvc5.github.io/docs/cvc5-1.2.1/proofs/output_cpc.html. A complete list of the proof rules in CPC can be found at https://cvc5.github.io/docs/cvc5-1.2.1/api/cpp/enums/proofrule.html. The semantics of the rules is also defined in the Eunoia logical framework, described in the user manual of the Ethos proof checker: https://github.com/cvc5/ethos/blob/main/user_manual.md.

We first use LEAN-AUTO[8] to reduce the goal to FOL. LEAN-AUTO normalizes universe levels, monomorphizes definitions, adds lemmas related to inductive types, and replaces type class instances by their corresponding values. All of these transformations are implemented in Lean and are proof producing, so their soundness is guaranteed by Lean's kernel. However, they are inherently incomplete, given the expressivity gap between DTT and FOL.

The preprocessing applied by LEAN-AUTO is not specific to SMT solvers, so while the resulting goal is in FOL, it is not aligned with the SMT-LIB standard. We apply a further preprocessing step so that certain Lean types and constructs (e.g., `Prop`, `Nat`, `Rat`, and `Iff`) that don't have direct counterparts in SMT-LIB can be transformed into types and constructs that do.

Example 1. Consider the following Lean goal, which asserts the uniqueness of the identity element in a group:

```
⊢ ∀ (G : Type u) [Group G] (e : G), (∀ (a : G), e * a = a) ↔ e = 1
```

This goal cannot be directly translated into SMT-LIB due to the presence of the type class `Group` and the logical operator ↔. During preprocessing, the goal is transformed and expanded to:

```
G: Type u
inst: Group G
e e': G
op: G → G → G
inv: G → G
one_mul: ∀ (a : G), op e a = a
inv_mul_cancel: ∀ (a : G), op (inv a) a = e
mul_assoc: ∀ (a b c : G),
op (op a b) c = op a (op b c)
⊢ (∀ (a : G), (op e' a = a)) = (e' = e)
```

Here, LEAN-AUTO replaces the type class `Group` with explicit assumptions about the group operations (e.g., associativity, identity, and inverse axioms), and LEAN-SMT's preprocessing transforms the bidirectional logical operator ↔ into a suitable equality comparison. These transformations make the goal compatible with SMT-LIB's logic while preserving its original meaning.

3.2 Translation to SMT-LIB

After preprocessing, translating Lean goals into SMT-LIB is relatively straightforward, as the fragments mostly overlap. However, one key challenge stems from differing assumptions about sorts: SMT-LIB assumes sorts are non-empty, while Lean allows types to be empty. This discrepancy can make the translation unsound. The reconstruction stage ensures soundness by failing if a proof step depends on a type being non-empty and Lean cannot establish that the type is an instance of the type class of non-empty types. As long as the instances are found, this discrepancy between the logic systems is successfully addressed.

[8] https://github.com/leanprover-community/lean-auto

Example 2. Continuing from our previous example, the preprocessed goal is translated into SMT-LIB as follows:

```
(declare-sort G 0)
(declare-const e G)
(declare-const |e'| G)
(declare-fun op (G G) G)
(declare-fun inv (G) G)
(assert (forall ((a G)) (= (op e a) a)))
(assert (forall ((a G)) (= (op (inv a) a) e)))
(assert (forall ((a G) (b G) (c G)) (= (op (op a b) c) (op a (op b c)))))
(assert (distinct (forall ((a G)) (= (op |e'| a) a) (= |e'| e))))
(check-sat)
```

Note that the translation is sound in this example because any group G is guaranteed to be non-empty due to the existence of the identity element e.

3.3 cvc5's Proof Format and Reconstruction

When cvc5 establishes that an SMT query is unsatisfiable, it can optionally generate a proof in the CPC format that accurately mirrors its internal reasoning. In CPC, each proof rule can be represented as follows:

$$\frac{\varphi_1 \ \cdots \ \varphi_m \ \mid \ t_1 \ \cdots \ t_n}{\psi} \text{ if } C$$

where $\varphi_1, \ldots, \varphi_m$ are the premises, $t_1, \ldots, t_n$ are the arguments provided to the proof rule, and C (which is optional) denotes a decidable side-condition. The proof format currently specifies over 662 proof rules in various domains, including arithmetic, strings, quantifiers, and higher-order logic. lean-smt supports around 200 of these proof rules, which currently amounts to approximately 30% of cvc5's proof rules. We prioritized this subset because it suffices to support the most common proof goals in Lean.[9] The remaining rules are required for less common reasoning steps used by specific theory solvers. To reconstruct the cvc5 proofs in Lean, lean-smt processes the CPC proof step by step, translating each proof step into an equivalent one in Lean. After lean-smt completes proof reconstruction, the entire proof is submitted to the Lean kernel for verification. In cases where a proof step cannot be reconstructed, it is presented to the user as a subgoal to be proved manually, ensuring that the reconstruction process remains sound. Note that since the logic of SMT-LIB is classical, certain parts of the proof rely heavily on the axiom of choice. Below we detail each of the reconstruction techniques we apply.

Reconstruction via theorems. Proof rules without side conditions generally correspond directly to theorems in Lean. We proved an extensive library of such theorems, which cover 163 proof rules, to use for proof reconstruction.

[9] For comparison, it corresponds to the same logical fragment supported initially by Sledgehammer and SMTCoq.

Example 3. Consider the `ARITH_MULT_TANGENT` proof rule from CPC.

$$\frac{- \mid x, y, a, b, \sigma}{(xy \leq \mathit{tplane}) = ((x \leq a \wedge y \geq b) \vee (x \geq a \wedge y \leq b))} \text{ if } \sigma = \top$$
$$\frac{- \mid x, y, a, b, \sigma}{(xy \leq \mathit{tplane}) = ((x \leq a \wedge y \leq b) \vee (x \geq a \wedge y \geq b))} \text{ if } \sigma = \bot$$

where x, y are real terms, a, b are real constants, $\sigma \in \{\top, \bot\}$ and $\mathit{tplane} := b \cdot x + a \cdot y - a \cdot b$ is the tangent plane of $x \cdot y$ at (a, b). Formalizing this proof rule into a Lean theorem is straightforward:

```
theorem arithMulTangentLowerEq :
(x * y ≤ b * x + a * y - a * b) = ((x ≤ a ∧ y ≥ b) ∨ (x ≥ a ∧ y ≤ b))
theorem arithMulTangentUpperEq :
(x * y ≤ b * x + a * y - a * b) = ((x ≤ a ∧ y ≤ b) ∨ (x ≥ a ∧ y ≥ b))
```

There are several CPC rules, however, which are more complex and require careful consideration in order to correctly capture their semantics when stating the corresponding Lean theorems.

Example 4. Consider for example the `RESOLUTION` proof rule:

$$\frac{C_1 \quad C_2 \mid \mathit{pol}, L}{C},$$

where C_1, C_2 are disjunctions, L is a disjunct occurring positively (respectively, negatively) in C_1 and negatively (resp., positively) in C_2 if *pol* is the Boolean constant *true* (resp., *false*). The result C is a disjunction consisting of the disjuncts from C_1 minus L (resp., $\neg L$) and the disjuncts from C_2 minus $\neg L$ (resp., L) if *pol* is *true* (resp., *false*). Moreover, C is a *flat* disjunction of disjuncts from C_1 and C_2 as opposed to a nested disjunction, reflecting cvc5's treatment of logical disjunction as a variadic operator. To capture the semantics of this rule precisely in Lean, where logical disjunction is expressed by a binary operator, one needs to carefully reason about the associativity and commutativity of that operator. We encapsulate the semantics in a Lean theorem as follows:

```
theorem orN_resolution (hps : orN ps) (hqs : orN qs)
    (hi : i < ps.length) (hj : j < qs.length)
    (hij : ps[i] = ¬qs[j]) :
    orN (ps.eraseIdx i ++ qs.eraseIdx j)
```

The premises, `hps : orN ps` and `hqs : orN qs`, are disjunctions but are built with the `orN` function, which takes a `List` of literals, each represented as a `Prop`. This formulation avoids representing `ps` and `qs` as, for example, inductively defined instances of `Prop` instead of `List` or encoding the literals as `Bool` instead of `Prop`. Using `List` also permits leveraging general theorems about `List` that we proved for eliminating tedious corner cases that would otherwise arise in a `Prop` implementation. Additionally, the choice to encode Boolean literals as `Prop` is a deliberate choice due to the type-theoretic foundations of Lean.

Reconstruction via tactics. Some proof rules without side conditions require the application of multiple lemmas or more complex reasoning. We implement specialized tactics to encapsulate these steps. This streamlines the reconstruction process by automating repetitive or intricate reasoning steps. We cover 37 proof rules this way, which required implementing a library with around 400 theorems.

Example 5. A proof rule that is very general, making it hard to state and prove as a theorem, is the ARITH_SUM_UB proof rule:

$$\frac{\bigwedge_{i=1}^{n} a_i \bowtie_i b_i \mid a_i, b_i}{\sum_{i=1}^{n} a_i \bowtie^* \sum_{i=1}^{n} b_i}$$

where $\bowtie_i$ can be either $<$, $\leq$ or $=$, and $\bowtie^*$ is either $<$, if at least one of the $\bowtie_i$ is $<$, or $\leq$, otherwise. Moreover, while each pair of variables a_i and b_i always have the same type, it is possible that different pairs have different types, some being integers and some being reals. It is possible to encode this proof rule as a single theorem in Lean, but the statement of the theorem would be quite intricate, due to the necessity of lifting the integer variables to reals and of combining the inequalities statically. Also, it is likely that this would make it very hard to prove. In this case, it is easier to write a tactic that considers the different cases of the rule and applies an appropriate, simpler theorem for each case. The implementation of this tactic requires 9 variations of the following general theorem:

```
sumBoundsThm {α : Type} [LinearOrderedRing α] {a b c d : α} :
  a < b → c < d → a + c < b + d
```

Each variation corresponds to one possible combination of the inequality symbols in the hypothesis. The relation symbol in the conclusion is adapted accordingly in each theorem. Since the proof rule accepts mixing of integer and real variables, we need a variation of each one of those 9 theorems for each combination of the types of the variables. Instead of stating all the combinations explicitly, which would result in a total of 36 theorems and a long branch in the implementation of the tactic, we state only one polymorphic version of each, as indicated by the type parameter α in theorem `sumBoundsThm`. Obviously, the theorem does not hold for just any type α. In fact, it cannot even be stated if there are no comparison or addition operators defined over α. We solve this issue by adding a restriction, stating that α satisfies the axioms of a *Linear Ordered Ring* (a class of types that contains both `Int` and `Real` defined in Lean's `mathlib` library). With this restriction, we can prove each of the 36 theorems. The full tactic can be seen in the appendix of a longer version of this paper [27].

Reconstruction via reflection. For proof rules involving complex side conditions or computations such as arithmetic simplifications, we encode the side condition into a reflective decision procedure, which we have formally verified in Lean. The proof rule is then translated into a theorem with the side condition as an additional premise. Applications of such rules are verified by the Lean kernel using definitional equality. We cover 5 proof rules this way. We reused one program from Lean's library, `ac_rfl`, which applies associative and commutative properties of addition and multiplication to normalize arithmetic expressions; and we implemented a new program, `poly_norm`, which normalizes polynomials up to associativity, commutativity, and distributivity by expanding polynomials.

Example 6. Consider the example:

```
example (x y : Int) (z : Real) :
  1 * ↑(x + y) * z / 4 = 1 / (2 * 2) * (z * ↑y + ↑x * z) := by poly_norm
```

Proving the correctness of `poly_norm` required proving around 70 theorems, amounting to 620 lines of Lean code. We define a monomial as an ordered list of natural numbers representing variable indices, so that equality of two monomials is immediate. Polynomials are defined as lists of monomials, and theorems about monomials generalize to polynomials using induction. We define `denote`, which essentially evaluates polynomial expressions. Using lemmas we proved about lists and the objects we defined, we prove a theorem pushing `denote` into each operator.

```
theorem denote_mul {p q : Polynomial} :
  (p.mul q).denote ctx = p.denote ctx * q.denote ctx
```

Proving similar theorems for each operator (addition, multiplication, division by a constant, and negation) yields a correctness theorem.

```
theorem denote_eq_from_toPoly_eq {e₁ e₂ : RealExpr}
  e₁.toPoly = e₂.toPoly → e₁.denote ictx rctx = e₂.denote ictx rctx
```

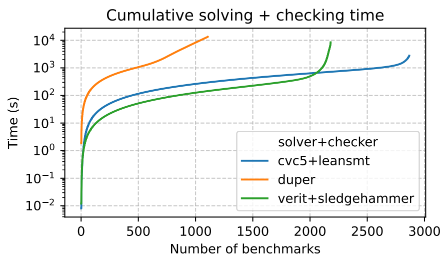
Since variables are represented as natural numbers, the premise of the theorem does not contain actual variables. Therefore, Lean can establish the premise through definitional equality. Moreover, the premise is decidable, allowing us to compile it into machine code for enhanced performance. In our experiments, this approach achieved speedups of up to 25x on very large arithmetic expressions compared to using definitional equality.

4 Evaluation and Results

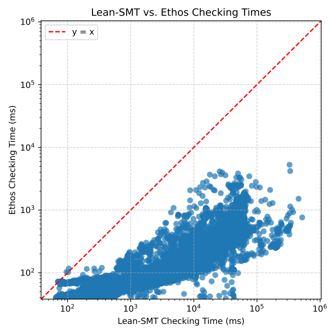
The LEAN-SMT tactic is mainly designed to prove Lean goals provided by users, but can also act as a proof checker for CPC proofs in supported fragments. We evaluate it[10] in these scenarios with two sets of benchmarks: a set of 5000 SMT-LIB benchmarks generated by Sledgehammer from Isabelle/HOL, taken from *Seventeen Provers Under the Hammer* [15]; and a set of 24,817 unsatisfiable SMT-LIB benchmarks used in SMT-COMP 2024[11] that fit the supported fragments of LEAN-SMT: UF, IDL, RDL, LIA, LRA, LIRA, and their quantifier-free subfragments. The Sledgehammer benchmarks allow us to assess LEAN-SMT's performance in both mathematics and formal verification domains and include

[10] All benchmarks were executed on a cluster with nodes equipped with 48 AMD Ryzen 9 7950X processors running at 4.50GHz and 128GB of RAM each.

[11] https://smt-comp.github.io/2024/



(a)



(b)

Fig. 2: (a) shows the performance of LEAN-SMT on Sledgehammer benchmarks, while (b) compares proof checking performance of LEAN-SMT with Ethos.

lemmas selected by Sledgehammer with its premise selection mechanism. The problems in SMT-COMP are used together with proof-producing SMT solvers to generate proofs that are passed on to a set of proof checkers, including LEAN-SMT.

4.1 Isabelle Sledgehammer Benchmarks

We chose these benchmarks over other options (e.g., Lean's MathLib) due to Lean's lack of a premise selection mechanism, which is crucial for reducing false positives (i.e., goals found to be invalid by the SMT solver due to missing premises). These benchmarks also stress test solvers, as they contain many (up to 512) lemmas. We compare the performance of LEAN-SMT against Sledgehammer with the veriT back end, which supports similar proof reconstruction techniques, and Duper. We do not include CVC4 as a back end for Sledgehammer because its proof production is unstable and does not provide sufficient detail for reliable reconstruction. The results in Figure 2a show that LEAN-SMT effectively solves a large variety of Sledgehammer benchmarks, underscoring its potential for integration into general-purpose proof environments. LEAN-SMT outperforms veriT+Sledgehammer mainly because CVC5 outperforms veriT on this set of benchmarks. LEAN-SMT takes less than a second to replay proofs for 98% of the benchmarks, with the remaining 2% taking less than 5 seconds. Sledgehammer, by comparison, is faster at reconstructing shorter proofs, but does not scale as well for larger proofs.

4.2 SMT-COMP Benchmarks

We evaluate LEAN-SMT on the selected SMT-COMP benchmarks against the proof checkers Ethos[12] v0.1.0 and SMTCoq v2.2. Ethos is a proof checker implemented in C++ for the Eunoia logical framework, in which CPC has been formalized. SMTCoq is a proof checker in OCaml extracted from a formally

[12] https://github.com/cvc5/ethos/

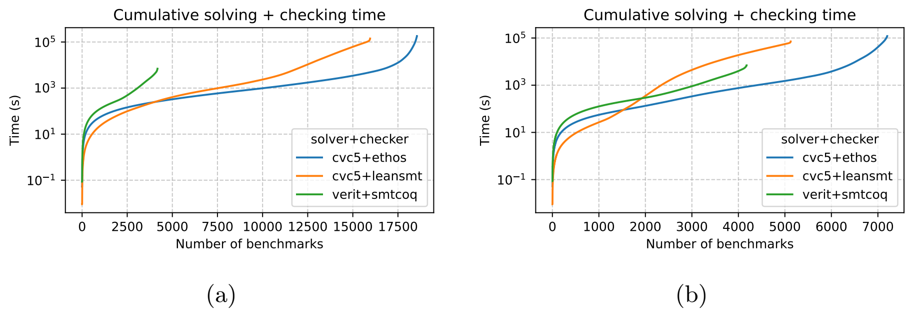


Fig. 3: Figure (a) shows the performance of LEAN-SMT on supported SMT-LIB fragments, while Figure (b) shows the performance on the quantifier-free subset.

verified Rocq program, and supports proofs in a subset of the Alethe proof format [35]. It can check proofs produced by versions of the veriT SMT solver up to 2016. Since the different approaches use different SMT solvers, our evaluation includes both proof-checking and SMT solving times. All benchmarks were run with a standard 20-minute timeout.

Note that both LEAN-SMT and SMTCoq are highly trustworthy, since they both rely on small kernels. However, SMTCoq's code extraction mechanism, which extracts OCaml code from verified Rocq code, has to be trusted as well. The trusted base for Ethos, besides its kernel, also includes the Eunoia signature formalizing CPC. Moreover, its kernel includes native support for arithmetic (via GMP) and arrays for efficiency, while Lean's kernel only natively supports arithmetic.

Supported SMT-LIB Benchmarks Figure 3a compares the cumulative solving and checking times for all SMT-LIB benchmarks. Out of the 21,769[13] benchmarks for which proofs can be generated[14] by CVC5, LEAN-SMT successfully verified 16,583 proofs (76%), despite relying solely on the Lean kernel for soundness. Ethos verifies 98% of the proofs. Figure 2b compares the performance of LEAN-SMT to Ethos on proof checking times. LEAN-SMT stays within an order of magnitude of Ethos for most benchmarks. One reason for Ethos' superior performance is the lack of specialized support for arrays in Lean's kernel. This difference could be mitigated by switching to a more efficient array representation in Lean.

Quantifier-Free Fragment Figure 3b focuses on the quantifier-free subset of SMT-LIB benchmarks, where SMTCoq's verified approach shines in terms of speed. However, SMTCoq falls short in the total number of proofs verified, primarily because, due to its fully verified architecture, it has not kept pace with the rapidly

[13] This number is for CVC5 +LEAN-SMT; for CVC5 +Ethos, 21,660 proofs are produced. The difference is due to the overhead of proof printing and piping for the latter combo, while in the former, the proof is passed directly via the API to LEAN-SMT.

[14] On occasion, CVC5 will produce proofs containing holes. Both Ethos and LEAN-SMT verify every non-hole proof step.

evolving features of modern SMT solvers. In contrast, LEAN-SMT benefits from the flexibility of proof replay, which has allowed us to adapt it more easily to updates in CVC5's proof production capabilities, resulting in broader coverage with respect to SMTCoq.

Overall, LEAN-SMT balances flexibility and performance, achieving promising results for a proof checker deeply integrated into Lean. While it trails Ethos in raw speed, its ability to verify a wide range of benchmarks with a small trusted base makes it an attractive option for checking SMT proofs in critical domains.

5 Conclusion

LEAN-SMT, a trustworthy integration of the CVC5 SMT solver into the Lean 4 proof assistant, is a significant step toward building a Lean hammer that enhances automation and verifies proofs generated by CVC5. LEAN-SMT shows promising results when compared to the state-of-the-art proof checker Ethos, both in terms of performance and effectiveness. It is already being used by other Lean-based projects [33] and is capable of verifying a diverse range of SMT-LIB benchmarks. Future work includes creating a dedicated Lean benchmark set for more targeted evaluation and expanding LEAN-SMT to support additional SMT-LIB theories, such as bit-vectors, floats, datatypes, and strings. We also plan to extend its proof coverage beyond the current 200 rules and incorporate more preprocessing steps to boost performance. Finally, we plan to improve integration with LEAN-AUTO to support higher-order logic and extend support for common Lean datatypes (e.g., tuples, structures, and modular arithmetic). Our ultimate objective is to develop a Lean hammer that brings unprecedented automation and verification capabilities to Lean.

Acknowledgments. This work was partially supported by a gift from Amazon Web Services, the Stanford Center for Automated Reasoning, the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-24-2-1001. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of DARPA.

Disclosure of Interest Clark Barrett is an Amazon Scholar.

References

1. Avigad, J., de Moura, L., Kong, S., Ullrich, S.: Theorem proving in Lean 4, uRL: https://leanprover.github.io/theorem_proving_in_lean4/
2. Baanen, A.: Formalizing Fundamental Algebraic Number Theory. Phd-thesis - research and graduation internal, Vrije Universiteit Amsterdam (Jan 2024). https://doi.org/10.5463/thesis.541
3. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part I. Lecture Notes in Com-

puter Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24
4. Barbosa, H., Reynolds, A., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Viswanathan, A., Viteri, S., Zohar, Y., Tinelli, C., Barrett, C.W.: Flexible proof production in an industrial-strength SMT solver. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) International Joint Conference on Automated Reasoning (IJCAR). Lecture Notes in Computer Science, vol. 13385, pp. 15–35. Springer (2022). https://doi.org/10.1007/978-3-031-10769-6_3, https://doi.org/10.1007/978-3-031-10769-6_3
5. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification (CAV). pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14, http://dx.doi.org/10.1007/978-3-642-22110-1_14
6. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at `www.SMT-LIB.org`
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). https://doi.org/10.1007/978-3-662-07964-5
8. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. In: Bjørner, N.S., Sofronie-Stokkermans, V. (eds.) Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6803, pp. 116–130. Springer (2011). https://doi.org/10.1007/978-3-642-22438-6_11, https://doi.org/10.1007/978-3-642-22438-6_11
9. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formalized Reasoning **9**(1), 101–148 (2016)
10. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64 (2011)
11. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An Open, Trustable and Efficient SMT-Solver. In: Schmidt, R.A. (ed.) Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 5663, pp. 151–156. Springer (2009). https://doi.org/10.1007/978-3-642-02959-2_12, http://dx.doi.org/10.1007/978-3-642-02959-2_12
12. Castelvecchi, D.: Mathematicians welcome computer-assisted proof in "grand unification" theory. Nature **595** (06 2021). https://doi.org/10.1038/d41586-021-01627-2
13. 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.) 15th International Conference on Interactive Theorem Proving (ITP 2024). Leibniz International Proceedings in Informatics (LIPIcs), vol. 309, pp. 10:1–10:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024). https://doi.org/10.4230/LIPIcs.ITP.2024.10, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2024.10
14. mathlib Community, T.: The Lean mathematical library. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 367–381. CPP 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372885.3373824

15. Desharnais, M., Vukmirovic, P., Blanchette, J., Wenzel, M.: Seventeen provers under the hammer. In: Andronick, J., de Moura, L. (eds.) 13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel. LIPIcs, vol. 237, pp. 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPICS.ITP.2022.8, https://doi.org/10.4230/LIPIcs.ITP.2022.8
16. Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., Barrett, C.: SMTCoq: A plug-in for integrating SMT solvers into Coq. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 126–133. Springer International Publishing, Cham (2017)
17. Enderton, H.B.: A mathematical introduction to logic. Academic Press, 2 edn. (2001)
18. Gonthier, G.: The four colour theorem: Engineering of a formal proof. In: Kapur, D. (ed.) Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers. Lecture Notes in Computer Science, vol. 5081, p. 333. Springer (2007). https://doi.org/10.1007/978-3-540-87827-8_28, https://doi.org/10.1007/978-3-540-87827-8_28
19. Hales, T., Adams, M., Bauer, G., Dang, D.T., Harrison, J., Hoang, T.L., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., Nguyen, T.Q., Nipkow, T., Obua, S., Pleso, J., Rute, J., Solovyev, A., Ta, A.H.T., Tran, T.N., Trieu, D.T., Urban, J., Vu, K.K., Zumkeller, R.: A formal proof of the kepler conjecture (2015)
20. Hurd, J.: First-order proof tactics in higher-order logic theorem provers in proc (2003), https://api.semanticscholar.org/CorpusID:11201048
21. Jakubuv, J., Chvalovský, K., Goertzel, Z.A., Kaliszyk, C., Olšák, M., Piotrowski, B., Schulz, S., Suda, M., Urban, J.: Mizar 60 for mizar 50. In: Naumowicz, A., Thiemann, R. (eds.) Interactive Theorem Proving (ITP). LIPIcs, vol. 268, pp. 19:1–19:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ITP.2023.19, https://doi.org/10.4230/LIPIcs.ITP.2023.19
22. Kaliszyk, C., Urban, J.: Hol(y)hammer: Online ATP service for HOL light. Math. Comput. Sci. **9**(1), 5–22 (2015). https://doi.org/10.1007/S11786-014-0182-0, https://doi.org/10.1007/s11786-014-0182-0
23. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: formal verification of an operating-system kernel. Commun. ACM **53**(6), 107–115 (2010). https://doi.org/10.1145/1743546.1743574, https://doi.org/10.1145/1743546.1743574
24. Lachnitt, H., Fleury, M., Aniva, L., Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: Isarare: Automatic verification of SMT rewrites in isabelle/hol. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part I. Lecture Notes in Computer Science, vol. 14570, pp. 311–330. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_17, https://doi.org/10.1007/978-3-031-57246-3_17
25. Limperg, J., From, A.H.: Aesop: White-box best-first proof search for Lean. In: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 253–266. CPP 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3573105.3575671, https://doi.org/10.1145/3573105.3575671
26. Meng, J., Quigley, C., Paulson, L.C.: Automation for interactive proof: First prototype. Inf. Comput. **204**(10), 1575–1596 (2006). https://doi.org/10.1016/J.IC.2005.05.010, https://doi.org/10.1016/j.ic.2005.05.010

27. Mohamed, A., Mascarenhas, T., Khan, H., Barbosa, H., Reynolds, A., Qian, Y., Tinelli, C., Barrett, C.: Lean-SMT: An SMT tactic for discharging proof goals in Lean (2025), https://arxiv.org/abs/2505.15796
28. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 12699, pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37, https://doi.org/10.1007/978-3-030-79876-5_37
29. Nelson, L., Geffen, J.V., Torlak, E., Wang, X.: Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel. In: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020. pp. 41–61. USENIX Association (2020), https://www.usenix.org/conference/osdi20/presentation/nelson
30. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
31. Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Reconstructing fine-grained proofs of rewrites using a domain-specific language. In: Griggio, A., Rungta, N. (eds.) Formal Methods In Computer-Aided Design (FMCAD). pp. 65–74. IEEE (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_12, https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_12
32. Piotrowski, B., Mir, R.F., Ayers, E.: Machine-learned premise selection for Lean (2023), https://arxiv.org/abs/2304.00994
33. Pîrlea, G., Gladshtein, V., Kinsbruner, E., Zhao, Q., Sergey, I.: Veil: A Framework for Automated and Interactive Verification of Transition Systems. In: Piskac, R., Rakamaric, Z. (eds.) Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 21-25, 2025, Proceedings. Lecture Notes in Computer Science, Springer (2025), to appear.
34. Qian, Y., Clune, J., Barrett, C., Avigad, J.: Lean-auto: An interface between Lean 4 and automated theorem provers. In: Piskac, R., Rakamaric, Z. (eds.) Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 21-25, 2025, Proceedings. Lecture Notes in Computer Science, Springer (2025), to appear.
35. Schurr, H., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a generic SMT proof format (extended abstract). CoRR **abs/2107.02354** (2021), https://arxiv.org/abs/2107.02354
36. Schurr, H., Fleury, M., Desharnais, M.: Reliable reconstruction of fine-grained proofs in a proof assistant. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12699, pp. 450–467. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_26, https://doi.org/10.1007/978-3-030-79876-5_26
37. Tao, T.: Machine assisted proof. AMS Notices **72**(1), 86–95 (2025). https://doi.org/10.1090/noti3041, https://doi.org/10.1090/noti3041