Hint-Based SMT Proof Reconstruction

Joshua Clune[1[0000−0003−4047−6196]], Haniel Barbosa[2[0000−0003−0188−2300]], and Jeremy Avigad[1[0000−0003−1275−315X]]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
[2] Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

Abstract. There are several paradigms for integrating interactive and automated theorem provers, combining the convenience of powerful automation with strong soundness guarantees. We introduce a new approach for reconstructing proofs found by SMT solvers which we intend to be complementary with existing techniques. Rather than verifying or replaying a full proof produced by the SMT solver, or at the other extreme, rediscovering the solver's proof from just the set of premises it uses, we explore an approach which helps guide an interactive theorem prover's internal automation by leveraging derived facts during solving, which we call hints. This makes it possible to extract more information from the SMT solver's proof without the cost of retaining a dependency on the SMT solver itself. We implement a tactic in the Lean proof assistant, called QUERYSMT, which leverages hints from the `cvc5` SMT solver to improve existing Lean automation. We evaluate QUERYSMT's performance on relevant Lean benchmarks, compare it to other tools available in Lean relating to SMT solving, and show that the hints generated by `cvc5` produce a clear improvement in existing automation's performance.

Keywords: SMT Solving · Interactive Theorem Proving · Lean

1 Introduction

When it comes to formal verification, interactive and automatic theorem provers (ITPs or proof assistants, and ATPs, respectively) have complementary strengths. Proof assistants offer powerful languages for expressing arbitrary mathematical statements and the ability to verify claims down to domain-specific axioms and the rules of the underlying logical foundation, but doing so often requires considerable effort. Automatic provers offer push-button verification but often fail to scale to complex verification tasks and do not provide the same strong guarantees as interactive theorem provers. Proof assistants like Isabelle/HOL [9,10,19,31], Rocq [1,16], and Lean [13,23,25] aim for the best of both worlds by translating goals in a proof assistant to the language of a powerful external prover and then using information from the external prover to reconstruct a proof of the original result that is checked within the ITP. The challenge, then, is to bridge the gap and establish appropriate communication between the two.

There are several existing paradigms for effectively communicating between ATPs and proof assistants. Their approaches to reconstructing ATP proofs vary.

One approach, often used for superposition theorem provers such as `E` [33], `Vampire` [18] and `Zipperposition` [32], is to supply a large number of premises to the ATP, use the ATP's proof to identify a minimal subset of necessary premises, and then supply just those premises to internal ITP automation such as METIS [17]. This has the benefit of entirely removing the call to the external prover, but has the possibility of failure because it ultimately depends on internal automation independently discovering a proof.

SMT solvers [7], which combine generic first-order reasoning with theory-specific decision procedures, generally warrant reconstruction methods which more closely follow the external solver's original proof. Isabelle/HOL relies on internal tactics to replay each step in the certificates from the SMT solvers it supports (`z3` [24], `veriT` [12], and `cvc5` [3]). SMTCoq uses the SMT solver `veriT`, which can produce detailed certificates, and checks its certificates via a formally verified procedure within Rocq. LEAN-SMT uses a mixtures of these two approaches, but mostly proof replay, to reconstruct certificates from `cvc5`.

Our goal in this paper is to explore an alternative approach to proof reconstruction. Instead of reconstructing the SMT solver's proof exactly, retaining a dependency on it, or discarding all information about the solver's proof except the set of premises used, our approach uses the SMT solver's proof to help guide ITP automation by providing *hints*.

We develop a Lean tactic, called QUERYSMT, which uses LEAN-AUTO [28] to export problems from Lean's dependent type theory to the language of an SMT solver. We then instrument `cvc5` to report the preprocessing and theory reasoning performed while solving the translated problem. This is used by QUERYSMT to insert, in the Lean source file, a self-contained proof script for the goal, with Lean formulations of the theory-specific facts formalized as subgoals. The proof script uses GRIND, a built-in Lean tactic inspired by SMT solvers, to supply proofs of those subgoals, and uses a proof-producing superposition prover, Duper [14], to prove the original goal using those facts. The result is a structured proof that users can inspect, modify, and simplify, if they wish. Notably, the proof does not depend on calling an SMT solver anymore. As in Isabelle's Sledgehammer when not using the `smt` tactic, the call to the external prover disappears.

We believe this approach has complementary strengths to others. SMT solvers are notoriously unstable; small changes in context, even as minor as renaming variables, can cause proofs to break, as well as different solver versions running on the same problem [34]. Therefore, eliminating the SMT call in favor of a modular source-code proof results in a more stable artifact. Additionally, we consider it a benefit that users can inspect and modify the resulting proof script. Powerful automation may be good at finding a proof, but it rarely yields the nicest one; the ability to revise and improve proofs should also contribute to stability.

We demonstrate the method with arithmetic and inductive types, two of the most important and common theories in Lean and other proof assistants, though our approach is not specific to these. We evaluate the approach on relevant benchmark problems from Lean's Init, Batteries, and Mathlib [15] libraries, and compare it to other SMT-related tools available in Lean. We show that although proof reconstruction does not always succeed, incorporating SMT hints significantly improves internal automation's chances of successful reconstruction.

Our contributions are as follows:

– We augment `cvc5` with the ability to record data that will be useful for proof reconstruction and report it back to Lean.
– We develop a method of translating statements about natural numbers to SMT queries on the integers.
– We augment our back-end reconstruction, DUPER [14], to implement a set of support strategy [29], improving its ability to incorporate SMT hints.
– We evaluate QUERYSMT's performance on Lean benchmarks, showing that QUERYSMT compares favorably to existing SMT-related Lean automation and that SMT hints produce a clear improvement in DUPER's performance.

2 QUERYSMT Overview

To demonstrate and evaluate our approach, we develop QUERYSMT, a Lean tactic which utilizes hints from the `cvc5` SMT solver to suggest a self-contained proof script. The overall structure of QUERYSMT is given in Figure 1.

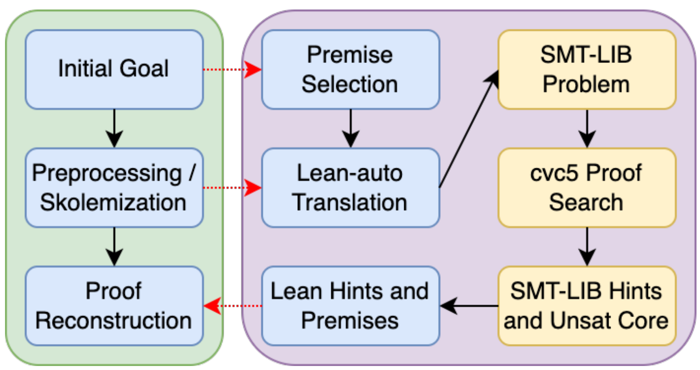

Fig. 1. Overview of the QUERYSMT tactic. Blue boxes indicate Lean stages. Yellow boxes indicate SMT stages. Stages in the green area directly transform the Lean goal and must be replayed in the final suggested proof script. Stages in the purple area do not transform the goal and therefore do not need to be replayed. Red dotted lines indicate information transfer between stages.

QUERYSMT consists of five primary components: preprocessing, translation, hint generation, hint interpretation, and proof reconstruction. Preprocessing transforms the goal into a form that reduces the likelihood of `cvc5` producing hints that can't be interpreted or proven. Translation from Lean's dependent type theory [2] to the many-sorted first-order logic used by SMT solvers [5] is handled by LEAN-AUTO [28], with extensions made to LEAN-AUTO's SMT-LIB translation described in Section 4. Hint generation consists of recording facts generated over the course of `cvc5`'s proof search and identifying an unsat core from the given goal and set of premises provided by premise selection. Hint interpretation involves translating `cvc5`'s generated hints into usable Lean expressions. Finally, proof reconstruction accepts `cvc5`'s unsat core and the output of hint interpretation and uses them to suggest a self-contained proof script that the user can examine and modify. Figure 2 contains an example of one such proof script. There, 5 hints are extracted from SMT lemmas and used by DUPER to discharge the goal. GRIND is used to justify the correctness of the hints themselves.

```
example (Even Odd : Int → Prop)
  (h1 : ∀ x : Int, ∀ y : Int, Odd (x) → Odd (y) → Even (x + y))
  (h2 : ∀ x : Int, ∀ y : Int, Odd (x) → Even (y) → Odd (x + y))
  (h3 : ∀ x : Int, Even (x) ↔ ¬ Odd (x))
  (h4 : Odd (1)) : Even (10) := by
  apply @Classical.byContradiction
  intro negGoal
  have smtLemma0 : Int.ofNat 5 + Int.ofNat 5 = Int.ofNat 10 := by
    grind
  have smtLemma1 : Int.ofNat 1 + Int.ofNat 4 = Int.ofNat 5 := by
    grind
  have smtLemma2 : Int.ofNat 1 + Int.ofNat 2 = Int.ofNat 3 := by
    grind
  have smtLemma3 : Int.ofNat 1 + Int.ofNat 1 = Int.ofNat 2 := by
    grind
  have smtLemma4 : Int.ofNat 1 + Int.ofNat 3 = Int.ofNat 4 := by
    grind
  duper [h1, h2, h3, h4, negGoal, smtLemma0, smtLemma1, smtLemma2,
    smtLemma3, smtLemma4] []
```

Fig. 2. A proof script suggested by QUERYSMT.

3 Preprocessing

Preprocessing takes a Lean goal of the form $\Gamma \vdash p : \mathsf{Prop}$ and transforms it into a goal of the form $\Gamma' \vdash \mathsf{False} : \mathsf{Prop}$ where all hypotheses in Γ' are Skolemized and Γ' entails Γ and $\neg p$. This transformation is necessary for two reasons.

First, the hints that `cvc5` generates may depend on the falsity of the initial target p. When this occurs, the generated hints are not entailed by Γ, meaning that any proof automation that attempts to derive the hint from Γ is doomed to fail. By preprocessing the goal into a state where the local context Γ' entails both Γ and $\neg p$, QUERYSMT ensures that when it comes time to prove the hints provided by `cvc5`, all pertinent information is accessible in the local context.

Second, the hints that `cvc5` generates may include constants that do not appear in the original SMT-LIB problem produced by LEAN-AUTO. This occurs when `cvc5` internally Skolemizes existential quantifiers, generating constants of the form `@QUANTIFIERS_SKOLEMIZE_X`.[3] While it is possible to recover the meaning of these constants by following the chain of inferences that were taken to reach `cvc5`'s Skolemization inferences, this approach essentially amounts to partial proof replay, which goes against the intention of QUERYSMT's design. To avoid this issue, QUERYSMT handles Skolemization in Lean prior to calling LEAN-AUTO's translation procedure. We define a tactic called `skolemizeAll` which iterates through every hypothesis in the local context and attempts to remove existential quantifiers and negated universal quantifiers, replacing them with fresh free variables. When no Skolemization is necessary because neither the original context Γ nor the negated target $\neg p$ contain existential quantifiers or negated universal quantifiers, QUERYSMT notes that `skolemizeAll` has no impact on the goal state and omits it from the final proof script suggestion. This is why the example in Figure 2 does not call `skolemizeAll`.

We note that because Lean's dependent type theory includes empty types, `skolemizeAll` may fail in cases where it is unable to verify type inhabitation. For example, Skolemizing the hypothesis $h : (\exists x : \alpha, P\ x) \vee \texttt{True}$ requires generating a free variable $y : \alpha$ and replacing h with $h' : P\ y \vee \texttt{True}$. However, h alone does not entail that α is inhabited. Unless α is already known to be inhabited, `skolemizeAll` cannot soundly add $y : \alpha$ to the local context. Consequently, there are some theorems that QUERYSMT is fundamentally unable to tackle due to the presence of empty types or possibly empty polymorphic types. This limitation is inherent to approaches translating from Lean's logic to SMT-LIB, and also affects e.g. LEAN-SMT [23, Sec. 3.2], where proof replay may fail when the solver relied on the non-emptiness of a type and that cannot be established by Lean.

4 Translation to SMT-LIB

4.1 LEAN-AUTO

For translation to the many-sorted first-order logic of SMT-LIB, we rely on LEAN-AUTO [28]. LEAN-AUTO normalizes universe levels, monomorphizes definitions, handles definitional equalities, and broadly takes care of the many features that make Lean's type theory complex. When targeting SMT-LIB, LEAN-AUTO attempts to translate Lean types to their closest SMT-LIB analogues. For example, Lean's `Prop` and `Bool` types are both translated to SMT-LIB's `Bool` sort, and Lean's `Int` and `Nat` types are both translated to SMT-LIB's `Int` sort. This ensures that the translation takes full advantage of SMT solvers' theories, but sometimes creates complications for interpreting the SMT hints (see Section 6).

[3] As defined in `https://cvc5.github.io/docs/cvc5-1.3.1/skolem-ids.html`, this constant corresponds to a term resulting from Skolemization, which could be defined e.g. via Hilbert's choice operator if the Skolemized quantifier were given as well. However, this is only done when proofs are generated.

LEAN-AUTO primarily translates essentially higher-order problems in Lean to monomorphic higher-order logic, but also can translate essentially first-order problems in Lean to first-order logic for the purpose of targeting SMT-LIB [14,28]. Although the most recent version of the SMT-LIB standard (Version 2.7) defines an SMT-LIB logic which extends beyond the many-sorted first-order logic adopted by the previous version [6], this SMT-LIB logic is not specifically targeted by LEAN-AUTO. When the upcoming Version 3 of the SMT-LIB standard is released with a higher-order base logic, we expect it will be fruitful to modify LEAN-AUTO so that it uses its primary translation procedure even when targeting SMT-LIB. This would expand the fragment of Lean that can be effectively translated to SMT-LIB, but is beyond the scope of this paper.

4.2 Translating Natural Numbers

Unlike the translation procedures used by SMTCoq [1] or Isabelle's Sledgehammer [21,22,26,27], LEAN-AUTO does not adopt an encoding-based approach to translating natural numbers. Instead, LEAN-AUTO directly translates Lean terms of type `Nat` to SMT-LIB terms of sort `Int` along with assertions of the form `(assert (>= n 0))`. Although this approach is sufficient for many common use cases, it is incomplete and can lead to unsound translations when natural numbers are embedded in larger structures or inductive types. For example, LEAN-AUTO is able to soundly translate `example` $(n : \texttt{Nat}) : 0 \leq n := \ldots$ into an unsatisfiable SMT-LIB problem, but the SMT-LIB problem generated from `example` $(x : \texttt{Nat} \times \texttt{Nat}) : 0 \leq x.\texttt{fst} := \ldots$ is satisfiable because LEAN-AUTO fails to assert that both projections of x must be nonnegative.

We extend LEAN-AUTO's procedure by preserving the direct translation from Lean `Nat` terms to SMT-LIB `Int` terms while expanding the set of circumstances in which nonnegativity assertions are made. For every Lean type α which appears in the problem, we define an SMT-LIB predicate $\texttt{wf}_\alpha : \hat{\alpha} \to \texttt{Bool}$ where $\hat{\alpha}$ is the SMT-LIB sort corresponding to α. This predicate is meant to encode the fact that the SMT-LIB term it applies to is *well-formed*, meaning it satisfies all of the nonnegativity constraints imposed by the `Nat` type on the Lean term from which it was derived. As a concrete example, if $\alpha = \texttt{Nat} \times \texttt{Int}$, then $\texttt{wf}_\alpha$ asserts that the first projection of the term it applies to is nonnegative.

Definition 1. *The predicate* $\texttt{wf}_\alpha\ \texttt{x}$ *is defined inductively on* α *as follows:*

- *If* $\alpha = \texttt{Nat}$ *then:*
  - $\texttt{wf}_\alpha\ \texttt{x} = \texttt{(>= x 0)}$
- *If* $\alpha = \alpha_1 \to \alpha_2$ *then:*
  - $\texttt{wf}_\alpha\ \texttt{x} = \texttt{(forall ((y } \hat{\alpha_1}\texttt{)) (=> (wf}_{\alpha_1}\texttt{ y) (wf}_{\alpha_2}\texttt{ (x y))))}$
- *If* α *is a structure*[4] *with projections* $p_1 : \alpha \to \alpha_1, \ldots p_n : \alpha \to \alpha_n$ *then:*

[4] In Lean's type theory, all structures are inductive datatypes, meaning there is no need to distinguish between them [2]. Definition 1's treatment of structures is logically equivalent to its treatment of inductive datatypes with one constructor, so it would be straightforward to eliminate the distinction. We nonetheless distinguish between them because only structures are guaranteed to have projections already defined in Lean. This has consequences for hint interpretation, which are discussed in Section 6.

  - $\texttt{wf}_{\alpha}\ \texttt{x} = \bigwedge_{i=1}^{n} \texttt{wf}_{\alpha_i}\ (\hat{p_i}\ \texttt{x})$

- *If α is an inductive datatype with constructors $(c_1 : \beta_{1,1} \to \ldots \to \beta_{1,m_1} \to \alpha), \ldots (c_n : \beta_{n,1} \to \ldots \beta_{n,m_n} \to \alpha)$ and is translated to a datatype with constructors $\hat{c_1}, \ldots \hat{c_n}$ and selectors $s_{i,j} : \hat{\alpha} \to \hat{\beta_{i,j}}$ then:*
  - $\texttt{wf}_{\alpha}\ \texttt{x} = \bigwedge_{i=1}^{n} \texttt{(=> (x is } \hat{c_i}\texttt{) (} \bigwedge_{j=1}^{m_i} \texttt{wf}_{\beta_{i,j}}\ (s_{i,j}\ \texttt{x})\texttt{)))}$
- *Otherwise:*
  - $\texttt{wf}_{\alpha}\ \texttt{x} = \texttt{True}$

To ensure that the semantics of the SMT-LIB problem coincide with the semantics of the original Lean goal, `wf` constraints are inserted such that almost all terms which appear in the SMT-LIB problem are provably well-formed. Whenever an SMT-LIB function or constant is declared, an assertion is added to guarantee that it is well-formed. Additionally, whenever an SMT-LIB formula is translated from a Lean proposition, the translation of quantifiers is modified to assert the well-formedness of the introduced variable. Universal quantification over α is translated to $\texttt{(forall ((x } \hat{\alpha}\texttt{)) (=> (wf}_{\alpha}\texttt{ x) (}\ldots\texttt{)))}$ and existential quantification over α is translated to $\texttt{(exists ((x } \hat{\alpha}\texttt{)) (and (wf}_{\alpha}\texttt{ x) (}\ldots\texttt{)))}$.

We note that this approach to translating natural numbers appears to coincide with `Trakt`'s methodology for handling partial embeddings on problems that do not involve inductive datatypes [11]. Our approach diverges from `Trakt`'s when inductive datatypes are involved because `Trakt` is intended to provide preprocessing transformations that are independent of the targeted backend while we explicitly aim to take advantage of SMT solvers' built-in datatype support.

Theorem 1. *All terms except datatype selectors[5] which appear in an SMT-LIB problem generated by our procedure are provably well-formed.*

Proof (sketch). Let $t : \hat{\alpha}$ be some term which appears in the generated SMT-LIB problem. The proof proceeds by induction on t. Here, we show one nontrivial case. For the full proof, see Appendix A.

- If t is an application of of the form $\texttt{(}t_1\ \ t_2\texttt{)}$, then the Lean term corresponding to t_1 has type $\beta \to \alpha$ and the Lean term corresponding to t_2 has type β. By the inductive hypothesis, t_2 is well-formed and t_1 is either well-formed or a selector function.
  - If t_1 is well-formed, then from the definition of well-formedness on functions, $\texttt{wf}_{\beta\to\alpha}\ t_1 = \texttt{(forall ((y } \hat{\beta}\texttt{)) (=> (wf}_{\beta}\texttt{ y) (wf}_{\alpha}\texttt{ (}t_1\texttt{ y))))}$. From this and $\texttt{wf}_{\beta}\ t_2$, it follows that $\texttt{wf}_{\alpha}\ \texttt{(}t_1\ \ t_2\texttt{)}$ as desired.

[5] Theorem 1 does not assert that datatype selectors are well-formed because in general they aren't. When a selector is passed a well-formed datatype built from the wrong constructor, the resulting application's output is only constrained by its sort (see remark 20 of the SMT-LIB standard [6]). Therefore, the output may fail to satisfy the nonnegativity constraints required by well-formedness.

- If t_1 is a selector function, then β is a structure or inductive datatype and t_1 has some associated constructor $\hat{c}$. From the definition of well-formedness on structures and inductive datatypes, $\texttt{wf}_\beta\ t_2$ entails `(=> (`t_2 `is` $\hat{c}$`) (wf`$_\alpha$ `(`$t_1\ t_2$`)))`. LEAN-AUTO's translation procedure guarantees that if `(`$t_1\ t_2$`)` appears in the generated SMT-LIB problem, then t_2 satisfies the tester for t_1's constructor, meaning `(`t_2 `is` $\hat{c}$`)` holds. From this and the implication entailed by $\texttt{wf}_\beta\ t_2$, it follows that $\texttt{wf}_\alpha$ `(`$t_1\ t_2$`)`.

5 Hint Generation

We have instrumented the `cvc5` SMT solver to report hints for external tools (such as QUERYSMT) based on logical consequences derived during proof search from the input formula and the theories supported by the solver.

The hints are collected from the internal proof produced by `cvc5` [4], which is then discarded. We consider three kinds of hints: *preprocessing lemmas*, *theory lemmas*, and *rewrite steps*. We consider them because they each contain theory reasoning performed by the solver while proving the given goal. We restrict ourselves to hints that were *useful* to the solver, i.e., they were used in the final proof. A useful by-product of the search for hints is to also collect an *unsat core* of the input, i.e., the elements of the input that were actually present in the proof.

Preprocessing is a key element of SMT solving where an input formula is simplified according to a series of preprocessing passes, each potentially modifying the input, be it by replacing the formula with a simplified version or generating new formulas entailed by it. The hints collected are entailments between the input formula and the preprocessed ones. An example of a key preprocessing step performed by `cvc5` is the inference and application of a substitution over part of the input to eliminate terms that are definable by others, as per the rest of the input (e.g. if the input contains the equality $x = 5$, a substitution corresponding to $x \mapsto 5$ is applied to the rest of the input to remove x [4, Sect. 5.1]).

Theory lemmas are valid disjunctions of literals from one or more theories. They generally correspond to explanations of why a given assignment of truth values to literals is inconsistent (e.g. assigning `True` to both $a = b$ and $f(a) \neq f(b)$), and allow the pruning of search space relative to that wrong assignment.

Finally, we also collect intermediate theory reasoning steps applied during preprocessing and during theory lemma generation that correspond to rewrite steps. These are important because they encapsulate key theory reasoning that would be difficult to gauge from just the preprocessing or theory lemmas themselves. An example is how `cvc5` reduces all arithmetic terms to sums of monomials. Since the hints will contain only the fully reduced terms, including the intermediate rewrite steps increases the information made available to QUERYSMT. In Figure 2, all hints shown are rewrite steps employed during proof search, but not actual theory lemmas.

Normalization of AC operators. Internally `cvc5`, as most SMT solvers, represent associative operators not as binary but as n-ary operators, so it is common to apply a "flattening" simplification so that terms such as `(* (*` x_1 x_2`)` x_3`)` and `(*` x_2 `(*` x_3 x_1`)))` are represented as the same term `(*` x_i x_j x_k`)`. Not only are applications of the operator flattened, but arguments are rearranged according to a canonical order. This normalization is useful for solving, but it complicates proof reconstruction in systems not representing AC operators in a normalized form. Since hints generated by `cvc5` refer to the normalized version of a term, and this normalization is not present in the hints (it would only be present in a proper proof), the applicability of the hints by QUERYSMT can be limited unless QUERYSMT can infer the relation between original terms and their normalized versions. A potential solution is to integrate facts pertaining to AC reasoning in QUERYSMT's proof reconstruction, but this has the disadvantage that the extra facts may lead to proof instability and loss of performance. QUERYSMT does not make use of such facts by default, but has an option to enable their inclusion. We discuss the impact of this option in Appendix B.

6 Hint Interpretation

For the most part, interpreting `cvc5`'s SMT-LIB hints and translating them into Lean expressions is straightforward. At each stage of LEAN-AUTO's translation pipeline, LEAN-AUTO creates mappings from terms and types in the source language to terms and types in the target language. This is to ensure that when the same term or type appears multiple times in the source problem, each instance of said term or type is translated in the same way. We modified LEAN-AUTO's translation procedure to make each of these mappings reversible, allowing QUERYSMT to translate SMT-LIB symbols and identifiers by simply passing them to the composition of LEAN-AUTO's reversed mappings. Although this is sufficient in most cases, there are two complications which merit further discussion.

Non-injectivity. LEAN-AUTO's mapping from Lean types to SMT-LIB sorts is not injective. LEAN-AUTO translates Lean's `Prop` and `Bool` types to the same SMT-LIB `Bool` sort, and as discussed in Section 4, it also translates Lean's `Int` and `Nat` types into the same SMT-LIB `Int` sort. Consequently, it is possible for `cvc5`'s hints to contain applications which typecheck according to SMT-LIB's semantics but fail to typecheck when naively translated into Lean. For example, if the original Lean goal contains n : `Nat` and x : `Int`, `cvc5` may create a hint which involves subtracting $\hat{n}$ from $\hat{x}$, an operation that is unproblematic in SMT-LIB but would fail to typecheck when translated back into Lean.

QUERYSMT's approach to interpreting such hints consists of defaulting to `Prop` and `Int` interpretations of SMT-LIB's `Bool` and `Int` sorts, inserting coercions as needed. There is only one circumstance in which expressions must be coerced to `Bool` or `Nat`, namely, when supplying an argument to a function that takes `Bool` or `Nat` inputs. To give a concrete example, if f : `Nat` $\to$ `Nat`, n : `Nat`, and m : `Nat`, then the SMT-LIB hint `(<` $\hat{n}$ `(`$\hat{f}$ `(-` $\hat{n}$ $\hat{m}$`)))` would be translated to `Int.ofNat` $n <$ `Int.ofNat` $(f$ `(Int.ofNat` $n -$ `Int.ofNat` m`).natAbs`$)$. This is more verbose than the seemingly more natural translation $n < f\ (n - m)$, but a faithful interpretation of `cvc5`'s hints requires that all built-in mathematical operations occur on integers rather than naturals[6].

Note that in the previous example, f is not supplied with the (possibly negative) result of subtracting `Int.ofNat` m from `Int.ofNat` n. Instead, f is supplied with the absolute value of said result. This coercion is needed to make the Lean expression typecheck, but one might reasonably question whether it compromises the faithfulness of the hint's interpretation. To answer this concern, we observe a subtle consequence of Theorem 1.

Corollary 1. *For any Lean function* $f : \texttt{Nat} \to \alpha$ *and SMT-LIB formula* F*, if* F *is entailed by the translated problem output by* LEAN-AUTO*, then* $F[(\hat{f} \circ \texttt{abs})/\hat{f}]$ *is also entailed by the translated problem.*

Proof. From Theorem 1, wherever $(\hat{f}\ \texttt{t})$ appears in the problem generated by LEAN-AUTO, the term `t` is provably well-formed. Therefore, any nontrivial assertions about the output of $\hat{f}$ which can be derived from the generated problem are conditioned on $\hat{f}$'s input being nonnegative. Since the output of $\hat{f}$ on negative inputs is unconstrained, any derivable fact about $\hat{f}$ also applies to all functions which agree with $\hat{f}$ on nonnegative inputs. $\hat{f} \circ \texttt{abs}$ agrees with $\hat{f}$ on nonnegative inputs, so any fact that can be derived about $\hat{f}$ also applies to $\hat{f} \circ \texttt{abs}$.

Non-surjectivity. There are two ways SMT-LIB terms without direct Lean equivalents may appear in `cvc5`'s hints. First, during the course of `cvc5`'s proof search, `cvc5` may apply Skolemization rules to generate constants of the form `@QUANTIFIERS_SKOLEMIZE_X`. These constants lack direct analogues among the expressions that appear in the input goal, and short of tracing `cvc5`'s proof up to the Skolemization rule application, it is difficult to determine how to interpret them. Fortunately, the generation of such constants can be avoided by preprocessing the goal into a form that does not trigger such rule applications. This preprocessing is discussed in Section 3.

The second way SMT-LIB terms without direct Lean equivalents may appear in `cvc5`'s hints relates to the translation of inductive datatypes. In Lean, a constructor's arguments can be accessed via pattern matching or by invoking a recursor that every inductive type is automatically equipped with. But in SMT-LIB, constructors' arguments are accessed via selector functions whose symbols are given as part of the datatype's declaration. In the special case that the inductive datatype being translated is also a structure, these selector functions can be identified with the projection functions that come with all Lean structures. But when the inductive datatype being translated is not a structure, there is no guarantee that Lean has ready-made analogues for SMT-LIB's selector functions.

In order to interpret hints which refer to selector functions, QUERYSMT adds fresh functions to the local context along with proofs that they satisfy the property that characterizes SMT-LIB's selector functions. The functions themselves

[6] This is particularly relevant when subtraction is involved, as the semantics of subtraction on the naturals differs from the semantics of subtraction on the integers. For example, the SMT-LIB term `(+ (- x y) y)` always evaluates to x, but the Lean expression $(x - y) + y$ is equal to `Nat.max` x y if x and y have type `Nat`.

are constructed from the inductive datatype's recursor, and the proofs they are paired with assert that if the function is passed the correct constructor, then the resulting application returns the appropriate argument of said constructor.

```
example {α : Type} [Inhabited α] (x y : α) : [x] = [y] ↔ x = y := by
  apply @Classical.byContradiction
  intro negGoal
  obtain ⟨_List.cons_sel0, _List.cons_sel0Fact⟩ :
    ∃ (_List.cons_sel0 : List α → α),
      ∀ (arg0 : α) (arg1 : List α),
        _List.cons_sel0 (arg0 :: arg1) = arg0 := by
    apply
      Exists.intro (List.rec (motive := fun (_ : List α) => α)
        default fun (arg0 : α) (arg1 : List α) (_ : α) => arg0)
    intros
    rfl
  duper [negGoal, _List.cons_sel0Fact] []
```

Fig. 3. A proof script suggested by QUERYSMT showcasing how Lean analogues for SMT-LIB's selector functions are constructed. Only one of the selector functions for lists is reproduced in the proof script because the other selector function (which retrieves the tail of a nonempty list) is not needed for the proof DUPER finds.

Figure 3 provides an example showcasing the construction on a goal with lists. When given the wrong constructor, the function returns `default`, an arbitrary element of the appropriate type which is only accessible if said type is `Inhabited`. If typeclass inference cannot prove that the type is `Inhabited`, then it instead uses `sorry`, leaving the proof of type inhabitation as a subgoal for the user.

7 Proof Reconstruction

The primary goal of QUERYSMT's proof reconstruction is not to produce a complete proof term for the given goal. Instead, the goal of QUERYSMT's proof reconstruction is to suggest a self-contained proof script for the user to examine and potentially modify. All proof scripts suggested by QUERYSMT consist of:

1. A tactic sequence designed to reproduce the effects of QUERYSMT's preprocessing and Skolemization, described in Section 3.
2. A sequence of `obtain` statements which create functions satisfying the properties of SMT-LIB's selectors. The construction of these functions is described in Section 6.
3. A sequence of `have` statements which assert hints output by `cvc5`. These `have` statements are proven with GRIND, a built-in Lean tactic.
4. A final call to DUPER [14], a superposition theorem prover intended to reconstruct the logical component of `cvc5`'s top-level proof.

To minimize the suggested proof script, some sections are omitted if deemed unnecessary. As mentioned in Section 3, `skolemizeAll` is only added to the suggested proof script if Skolemization will change the goal. Additionally, the sequence of `obtain` and `have` statements from steps 2 and 3 are minimized by only including those that are necessary for the proof DUPER finds in step 4.

QUERYSMT's ability to perform this minimization, and therefore suggest a usable proof script, depends on DUPER successfully finding a proof. Although DUPER has been shown to be effective in solving problems previously minimized by other superposition theorem provers [14], its performance degrades significantly when given too many unnecessary or irrelevant premises, and it lacks the theory-specific knowledge leveraged by SMT solvers.

To increase DUPER's effectiveness in reconstructing the logical component of `cvc5`'s proofs, we augment its given clause procedure to implement a variant of the set of support strategy used by Vampire [29]. This set of support strategy is designed to enable reasoning about theory axioms while mitigating the negative impact of their explosive properties. Facts that are included in DUPER's set of support are treated normally, but facts that are excluded from DUPER's set of support are only considered when they can be applied to facts in the set of support. The core idea is to limit theory axioms' explosive behavior by only applying them to facts that directly relate to the original goal.

We initially implemented this set of support strategy to exclude `cvc5`'s hints from the set of support, thinking that `cvc5`'s hints might behave similarly to more general theory axioms. However, experiments described in Appendix B revealed that this was actually detrimental to performance. Instead, the set of support strategy is used to exclude a small set of theory lemmas detailing properties of integers and natural numbers that `cvc5`'s hints aren't expected to capture. The set of theory lemmas used by QUERYSMT is provided in Appendix C.

8 Evaluation

We evaluate QUERYSMT and existing tools on 9,904 theorems related to integers, natural numbers, and lists taken from Lean's Init, Batteries, and Mathlib [15] libraries. Our evaluation focuses on these domains in particular, rather than randomly selected theorems, because we specifically seek to evaluate whether QUERYSMT benefits from `cvc5`'s domain-specific knowledge. Int theorems are chosen to test QUERYSMT's ability to benefit from hints related to SMT-LIB's LIA logic. Nat theorems are chosen to test whether QUERYSMT can make use of these hints even when it requires encoding Nat goals into Int problems and inferring facts about natural numbers from hints about integers. List theorems are chosen as a proxy for testing QUERYSMT's ability to make use of SMT solvers' built-in support for reasoning about algebraic datatypes.

8.1 Methodology

We perform all experiments on version `leanprover/lean4:v4.22.0` of Lean. The 9,904 theorems used as benchmark problems are obtained by scraping user-defined theorems from `Init.Data.X`, `Batteries.Data.X`, and `Mathlib.Data.X`, where `X` is any file prefixed with `Int`, `Nat`, or `List`. A constant is considered a user-defined theorem if it is marked as a theorem, has an explicit declaration in source code, and is not a projection function. All experiments are performed on an Amazon EC2 `ami-04f167a56786e4b09` instance with 4 virtual CPUs and 16 GiB memory. Each theorem is given a wall clock timeout of 30 seconds and the default Lean heartbeat limit of 200,000. The short timeout is used to reflect the expectation of proof assistant users of having quick results from tactics.

For premise selection, we approximate an ideal premise selector by inspecting the existing proofs of the benchmark theorems and extracting the set of premises $\mathcal{P}$ used to prove them. We also gather the set of constants $\mathcal{C}$ that are not theorems which appear in `rw` or `simp` calls of tactic proofs. These constants are used to indicate to the relevant automation that said constant should be unfolded, enabling the automation to invoke definitional equalities not otherwise captured by $\mathcal{P}$. Both QUERYSMT and the tools we compare with benefit from receiving these constants, so in the experiments, all tools are given $\mathcal{P} \cup \mathcal{C}$ as input.

Our testing script implements the following procedure:

1. Identify the (theorem, tool) pair to be tested.
2. Create a temporary Lean file which imports the benchmark theorem's original file as well as any files needed to run the tool being evaluated.
3. In the temporary Lean file, define `alias fakeThm := originalThm`.
4. Compile the temporary Lean file and extract the `ConstantInfo` associated with `fakeThm` along with the environment immediately prior to executing the `alias` command.
5. In the environment extracted from the previous step, create a fresh metavariable whose type is determined by the extracted `ConstantInfo` and attempt to instantiate this metavariable with the tool being tested.[7]

We note that the environment in which the tools are run does not perfectly match the original proof's environment. It is infeasible to exactly mimic this environment because among the tools being evaluated, only GRIND (which is built into Lean and requires no special imports) would be callable. The differences between the original environments and the environments used in our experiments are largely benign, but in Section 8.2 we discuss one circumstance where the difference in environments is more impactful.

[7] The tool is considered to have succeeded if the variable is instantiated, regardless of whether the instantiation contains `sorry`. When LEAN-AUTO + `cvc5` is evaluated, the only proof it produces is `sorry` (indicating that `cvc5` found a proof and LEAN-AUTO trusts the result). QUERYSMT also closes goals with `sorry` because QUERYSMT is meant to be replaced with the suggested proof script. Since QUERYSMT only suggests a proof script if DUPER finds a proof that follows from the hints, the suggested script is expected to succeed up to proof reconstruction for the individual hint assertions, for which we have a high success rate.

8.2 Results

Tool Comparison We compare QUERYSMT's performance against other tools available in Lean which either interface with external SMT solvers or implement techniques used by SMT solvers. Descriptions of these tools are included in Figure 4 and their respective performances are shown in Table 1.

1. LEAN-AUTO + `cvc5`: A tactic which uses LEAN-AUTO to translate input problems into the SMT-LIB format and trusts any proofs produced by `cvc5`. This serves as a theoretical upper bound for both QUERYSMT and LEAN-SMT.
2. QUERYSMT: The default implementation of QUERYSMT. QUERYSMT is considered to succeed if DUPER finds a top level proof of the original goals assuming the hints given by `cvc5`. This does not necessarily entail that GRIND alone is sufficient to prove all the hints DUPER depends on. GRIND's success rate at proving the hints output by `cvc5` is evaluated separately.
3. QUERYSMT−: A modified implementation of QUERYSMT in which DUPER is not provided the hints output by `cvc5`. QUERYSMT− retains the preprocessing described in Section 3 and still uses `cvc5`'s unsat core to minimize the set of premises provided to DUPER, but does not translate `cvc5`'s hints into Lean subgoals or pass the resulting assertions into DUPER.
4. LEAN-SMT: A tactic that interfaces with `cvc5` and performs proof reconstruction via proof replay [23]. At the advice of one of its authors, LEAN-SMT is run with the `+ mono` option which instructs LEAN-SMT to use LEAN-AUTO as a component of its preprocessing.
5. GRIND: A built-in Lean tactic inspired by modern SMT solvers. GRIND does not interface with external SMT solvers, but the inspiration for its underlying design and widespread use make it a helpful point of comparison.

Fig. 4. Descriptions of SMT-related methods

Table 1. Benchmark theorems solved by SMT-related methods

	Int Theorems	Nat Theorems	List Theorems
Total	2058	3270	4576
LEAN-AUTO + `cvc5`	1137	1486	891
QUERYSMT	839	869	743
QUERYSMT−	472	627	708
LEAN-SMT	333	35	891
GRIND	541	812	-

In all categories, QUERYSMT performs noticeably better with SMT hints than without. On Int and Nat benchmarks, QUERYSMT only outperforms GRIND with these hints. The impact of hints on QUERYSMT's performance appears to be more significant on Int and Nat benchmarks than on List benchmarks, and not coincidentally, LEAN-AUTO + `cvc5` solves a much smaller fraction of List theorems than Int or Nat theorems. From manual inspection of the theorems involved, we suspect that a significant factor contributing to this discrepancy is that in the List category, there is a significant overlap between the set of theorems for which built-in datatype reasoning would be helpful and the set of theorems for which induction is necessary. `cvc5` is not able to solve problems requiring induction by default [30], or to produce proofs for it, so it cannot be used by QUERYSMT in this scenario. We therefore suspect that a smaller fraction of the List theorems being solved by LEAN-AUTO + `cvc5` genuinely require theory reasoning.

On problems relating to Ints and Nats, QUERYSMT performs best, followed by GRIND, followed by LEAN-SMT. We note that GRIND has stricter conditions on its input lemmas than QUERYSMT or LEAN-SMT, and also that GRIND benefits from additional hints about how to use its input lemmas (in the form of custom attributes). It may be possible to achieve better performance with GRIND either by tailoring the set of provided premises to better suit GRIND or by manually providing additional hints about how to use the premises it receives.

LEAN-SMT's performance in both the Int and Nat categories is severely diminished by the fact that LEAN-SMT lacks special support for natural numbers. This says more about temporary limitations resulting from LEAN-SMT's current coverage than the theoretical limit of LEAN-SMT's approach. Still, we note that one of the benefits of our method is relative ease of extensionality. It requires much less effort to add support for parsing hints in a new theory than to add full-fledged proof replay for the same theory.

On problems relating to Lists, LEAN-SMT outperforms QUERYSMT, with LEAN-SMT successfully reconstructing all proofs found by LEAN-AUTO + `cvc5`. This indicates that LEAN-SMT's approach to proof reconstruction (proof replay) is more reliable than QUERYSMT's on problems that are fully supported by LEAN-SMT.

We omit GRIND's performance on List problems in Table 1 because it is significantly impacted by our evaluation methodology. When evaluating GRIND's performance on List problems with the same script used in the other experiments, GRIND succeeds at finding proofs for 1,458 problems. However, upon investigating why GRIND performs so much better than the other tools, we discovered that a nontrivial number of problems are solved by GRIND accessing lemmas it wouldn't have access to in the original environment. In particular, several theorems tagged with GRIND attributes yield benchmark problems that GRIND solves by invoking the original theorem. When we modify the evaluation script to test GRIND in the original theorem's proof environment[8], GRIND only solves 439 problems. The interpretation of these results depends on whether one views the theorems tagged with GRIND attributes as part of GRIND's implementation.

Proof Reconstruction for Hints As noted in Figure 4, QUERYSMT's success criteria depends on DUPER deriving a proof of the original goal from the hints

[8] Note that this test is only feasible because GRIND is built into Lean and is therefore accessible in the original proof environment.

output by cvc5, but does not depend on GRIND succeeding at proving all of the subgoals generated by assuming these hints. If QUERYSMT finds a proof and successfully outputs a proof script in which GRIND fails to prove one or more of the generated hints, QUERYSMT has still done something valuable in reducing the original goal to a smaller subgoal. Still, QUERYSMT's usefulness is significantly impacted by the frequency with which cvc5's hints can be proven automatically, so we perform an additional evaluation to test how frequently cvc5 produces hints that GRIND can't solve. This evaluation includes not just hints that appear in QUERYSMT's suggestions, but all hints output by cvc5 regardless of whether DUPER succeeds in its proof search, and regardless of whether they would actually appear in the final proof script suggestion.

Of the 9,904 problems tested, 499 produce a set of hints that GRIND fails to certify, approximately 5% of the total. 57 of these failures come from Int problems, 168 of these failures come from Nat problems, and 274 of these failures come from List problems. From manual inspection, we know that many of these failures are false negatives owed to cvc5 producing large sets of hints that can be solved individually by GRIND but collectively cause GRIND to time out[9]. We still register such cases as failures because it is difficult to distinguish this behavior from tests in which GRIND genuinely times out on a single hint. Not all failures are false negatives, but anecdotally, the hints GRIND genuinely fails on tend to be easy to discharge manually using some combination of AESOP [20] and DUPER.

9 Conclusion

We explored a new hint-based approach to leveraging SMT solvers for ITP automation. We implemented this approach in the Lean proof assistant to create QUERYSMT, a tactic that translates Lean goals to SMT-LIB, extracts the preprocessing and theory reasoning used by cvc5 to solve the translated problem, and uses that information to produce a self-contained proof script for the original goal which does not depend on cvc5. We evaluated QUERYSMT on problems related to its supported theories, showing that QUERYSMT compares favorably to existing SMT-related Lean automation and that the hints extracted from cvc5 produce a clear improvement in the underlying proof automation.

We see several possible directions for future work. One possibility is to implement our approach in other proof assistants or SMT solvers to see whether it can be applied to more than just Lean and cvc5. Another is to add support for more SMT theories to see how our approach generalizes beyond hints relating to integers and algebraic datatypes. A third possibility is to explore ways to gather even more information from cvc5's proofs. For example, collecting instances generated for quantified formulas could lead to DUPER finding proofs more quickly, as was recently done for METIS in Isabelle's Sledgehammer [8]. We also expect that QUERYSMT's reconstruction success rate could be increased with better instrumentation for tracking how cvc5 normalizes AC operators.

[9] As in other experiments, GRIND gets 200,000 heartbeats and 30 seconds per theorem.

References

1. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses, p. 135–150. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_12
2. Avigad, J., de Moura, L., Kong, S., Ullrich, S.: Theorem proving in Lean 4, https://leanprover.github.io/theorem_proving_in_lean4/
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 Computer 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., 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
6. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.7. Tech. rep., Department of Computer Science, The University of Iowa (2025), available at www.SMT-LIB.org
7. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1267–1329. IOS Press (2021). https://doi.org/10.3233/FAIA201017, https://doi.org/10.3233/FAIA201017
8. Bartl, L., Blanchette, J., Nipkow, T.: Exploiting instantiations from paramodulation proofs in isabelle/hol. In: Barrett, C.W., Waldmann, U. (eds.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 15943, pp. 573–593. Springer (2025). https://doi.org/10.1007/978-3-031-99984-0_30, https://doi.org/10.1007/978-3-031-99984-0_30
9. 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
10. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formalized Reasoning **9**(1), 101–148 (2016)
11. Blot, V., Cousineau, D., Crance, E., de Prisque, L.D., Keller, C., Mahboubi, A., Vial, P.: Compositional pre-processing for automated reasoning in dependent type theory. In: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 63–77. CPP '23, ACM (Jan 2023). https://doi.org/10.1145/3573105.3575676, http://dx.doi.org/10.1145/3573105.3575676

12. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An Open, Trustable and Efficient SMT-Solver. In: Schmidt, R.A. (ed.) Proc. 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
13. Böving, H., Bhat, S., Cicolini, L., Keizer, A., Frenot, L., Mohamed, A., Stefanesco, L., Khan, H., Clune, J., Barrett, C., Grosser, T.: Interactive bitvector reasoning using verified bit-blasting. Proceedings of the ACM on Programming Languages **9**(OOPSLA2), 3259–3285 (Oct 2025). https://doi.org/10.1145/3763167, http://dx.doi.org/10.1145/3763167
14. 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.) Interactive Theorem Proving (ITP). 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
15. mathlib Community, T.: The Lean mathematical library. In: Blanchette, J., Hritcu, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020. pp. 367–381. ACM (2020). https://doi.org/10.1145/3372885.3373824
16. Czajka, L., Kaliszyk, C.: Hammer for coq: Automation for dependent type theory. Journal of Automated Reasoning **61**(1–4), 423–453 (Feb 2018). https://doi.org/10.1007/s10817-018-9458-4
17. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003). pp. 56–68 (2003), http://www.gilith.com/papers
18. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 1–35. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_1
19. Lachnitt, H., Fleury, M., Barbosa, H., Jakpor, J., Andreotti, B., Reynolds, A., Schurr, H., Barrett, C.W., Tinelli, C.: Improving the SMT proof reconstruction pipeline in isabelle/hol. In: Forster, Y., Keller, C. (eds.) Interactive Theorem Proving (ITP). LIPIcs, vol. 352, pp. 26:1–26:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025). https://doi.org/10.4230/LIPICS.ITP.2025.26, https://doi.org/10.4230/LIPIcs.ITP.2025.26
20. Limperg, J., From, A.H.: Aesop: White-box best-first proof search for lean. In: Krebbers, R., Traytel, D., Pientka, B., Zdancewic, S. (eds.) Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023. pp. 253–266. ACM (2023). https://doi.org/10.1145/3573105.3575671
21. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Autom. Reason. **40**(1), 35–60 (2008). https://doi.org/10.1007/S10817-007-9085-Y
22. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Appl. Log. **7**(1), 41–57 (2009). https://doi.org/10.1016/J.JAL.2007.07.004
23. 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, p. 197–212. Springer Nature Switzerland (2025). https://doi.org/10.1007/978-3-031-98682-6_11, http://dx.doi.org/10.1007/978-3-031-98682-6_11

24. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24
25. Norman, C., Avigad, J.: Canonical for automated theorem proving in lean. In: Forster, Y., Keller, C. (eds.) 16th International Conference on Interactive Theorem Proving, ITP 2025, September 28 to October 1, 2025, Reykjavik, Iceland. LIPIcs, vol. 352, pp. 14:1–14:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025). https://doi.org/10.4230/LIPICS.ITP.2025.14, https://doi.org/10.4230/LIPIcs.ITP.2025.14
26. Paulson, L.C., Blanchette, J.C.: Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Schulz, S., Ternovska, E. (eds.) The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011. EPiC Series in Computing, vol. 2, pp. 1–11. EasyChair (2010). https://doi.org/10.29007/36DT
27. Paulson, L.C., Susanto, K.W.: Source-level proof reconstruction for interactive theorem proving. In: Schneider, K., Brandt, J. (eds.) Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4732, pp. 232–245. Springer (2007). https://doi.org/10.1007/978-3-540-74591-4_18
28. Qian, Y., Clune, J., Barrett, C., Avigad, J.: Lean-Auto: An Interface Between Lean 4 and Automated Theorem Provers, p. 175–196. Springer Nature Switzerland (2025). https://doi.org/10.1007/978-3-031-98682-6_10
29. Reger, G., Suda, M.: Set of support for theory reasoning. In: Eiter, T., Sands, D., Sutcliffe, G., Voronkov, A. (eds.) IWIL Workshop and LPAR Short Presentations. Kalpa Publications in Computing, vol. 1, pp. 124–134. EasyChair (2017). https://doi.org/10.29007/ndjg, /publications/paper/4Sd
30. Reynolds, A., Kuncak, V.: Induction for smt solvers. In: D'Souza, D., Lal, A., Larsen, K. (eds.) Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science, vol. 8931, pp. 80–98. Springer Berlin Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_5, http://dx.doi.org/10.1007/978-3-662-46081-8_5
31. Schurr, H., Fleury, M., Desharnais, M.: Reliable reconstruction of fine-grained proofs in a proof assistant. In: Platzer, A., Sutcliffe, G. (eds.) Proc. Conference on Automated Deduction (CADE). 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
32. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making Higher-Order Superposition Work, p. 415–432. Springer International Publishing (2021). https://doi.org/10.1007/978-3-030-79876-5_24
33. Vukmirović, P., Blanchette, J., Schulz, S.: Extending a High-Performance Prover to Higher-Order Logic, p. 111–129. Springer Nature Switzerland (2023). https://doi.org/10.1007/978-3-031-30820-8_10, http://dx.doi.org/10.1007/978-3-031-30820-8_10
34. Zhou, Y., Bosamiya, J., Takashima, Y., Li, J., Heule, M., Parno, B.: Mariposa: Measuring SMT instability in automated program verification. In: Nadel, A., Rozier, K.Y. (eds.) Formal Methods In Computer-Aided Design (FMCAD). pp. 178–188. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_26, https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_26

A Proof of Theorem 1

Theorem 1. *All terms except datatype selectors which appear in an SMT-LIB problem generated by our procedure are provably well-formed.*

Proof. Let $t : \hat{\alpha}$ be some term which appears in the generated SMT-LIB problem. We proceed by induction on t:

- If t is a special constant (i.e. a numeral, decimal, hexadecimal, binary, or string), then either t is interpreted to be of sort `Int` and is nonnegative, or t is interpreted to be of a sort such that $\texttt{wf}_\alpha\ t = \texttt{True}$.
- If t is an identifier corresponding to a declared function or constant, then when said function or constant was declared, an assertion was added to guarantee that t is well-formed.
- If t is a symbol corresponding to a bound variable introduced by `forall` or `exists`, then when the original Lean quantifier was translated, $\texttt{wf}_\alpha\ t$ was added to the translated body (either as a hypothesis for a universal quantifier's body or in conjunction with an existential quantifier's body).
- If t is a symbol corresponding to a datatype's constructor, then α must have the form $\beta_1 \to \ldots \to \beta_n \to \gamma$ where $\beta_1, \ldots \beta_n$ are the types of the Lean constructor's inputs and γ is either an inductive datatype or a structure. From the definition of well-formedness on function types, $\texttt{wf}_\alpha\ t$ is equivalent to $\texttt{(=> (wf}_{\beta_1}\texttt{ y1) ... (wf}_{\beta_n}\texttt{ yn) (wf}_\gamma\texttt{ (t y1 ... yn)))}$ where `y1 ... yn` are arbitrary terms of sorts $\hat{\beta_1}, \ldots \hat{\beta_n}$.
  - If γ is a structure, then $\texttt{wf}_\gamma\texttt{ (}t\texttt{ y1 ... yn)}$ asserts that all projections of $\texttt{(}t\texttt{ y1 ... yn)}$ are well-formed. This follows from the implication's hypotheses $\texttt{(wf}_{\beta_1}\texttt{ y1) ... (wf}_{\beta_n}\texttt{ yn)}$, so $\texttt{wf}_\alpha\ t$ holds.
  - If γ is an inductive datatype, then $\texttt{wf}_\gamma\texttt{ (}t\texttt{ y1 ... yn)}$ asserts that for each of γ's constructors c_i, if $\texttt{(}t\texttt{ y1 ... yn)}$ satisfies $\texttt{(\_ is }\hat{c_i}\texttt{)}$, then for each selector $s_{i,j}$ of $\hat{c_i}$, $\texttt{(}s_{i,j}\texttt{ (}t\texttt{ y1 ... yn))}$ must be well-formed. Since $\texttt{(}t\texttt{ y1 ... yn)}$ satisfies exactly one tester, namely $\texttt{(\_ is }t\texttt{)}$, it follows that $\texttt{wf}_\gamma\texttt{ (}t\texttt{ y1 ... yn)}$ asserts that applying any of t's selectors to $\texttt{(}t\texttt{ y1 ... yn)}$ yields a well-formed term. This follows from the implication's hypotheses $\texttt{(wf}_{\beta_1}\texttt{ y1) ... (wf}_{\beta_n}\texttt{ yn)}$, so $\texttt{wf}_\alpha\ t$ holds.
- If t is a symbol corresponding to a datatype's selector, then Theorem 1 does not require that t be well-formed.
- If t is the `ite` function from SMT-LIB's `Core` theory, then α must have the form $\texttt{Prop} \to \beta \to \beta \to \beta$ for some type β. From the definition of well-formedness on function types, $\texttt{wf}_\alpha\texttt{ t}$ is equivalent to $\texttt{(=> (wf}_{\texttt{Prop}}\texttt{ y1) (wf}_\beta\texttt{ y2) (wf}_\beta\texttt{ y3) (wf}_\beta\texttt{ (ite y1 y2 y3)))}$ where `y1` is an arbitrary term of sort `Bool` and `y2` and `y3` are arbitrary terms of sort $\hat{\beta}$. `(ite y1 y2 y3)` evaluates to `y2` or `y3` depending on `y1`, and the hypotheses of the implication give $\texttt{(wf}_\beta\texttt{ y2)}$ and $\texttt{(wf}_\beta\texttt{ y3)}$, so the conclusion $\texttt{(wf}_\beta\texttt{ (ite y1 y2 y3))}$ follows from the implication's hypotheses. This entails $\texttt{wf}_\alpha\ t$ as desired.
- If $t \in \{\texttt{+}, \texttt{*}, \texttt{div}, \texttt{mod}, \texttt{abs}\}$, then the semantics of SMT-LIB's `Int` theory guarantee that t preserves nonnegativity and is therefore well-formed.

- If t is the "`-`" symbol and corresponds to negation, then $\alpha = \texttt{Int} \to \texttt{Int}$ so $\texttt{wf}_\alpha\ t$ is equivalent to `True`.
- If t is the "`-`" symbol and corresponds to subtraction, then $\alpha = \texttt{Int} \to \texttt{Int} \to \texttt{Int}$, entailing that $\texttt{wf}_\alpha\ t$ is equivalent to `True`. Note that $\alpha \neq \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$ even though subtraction on the naturals exists in Lean because LEAN-AUTO does not translate Lean subtraction on the naturals directly into SMT-LIB subtraction on the integers. LEAN-AUTO translates $(e_1 : \texttt{Nat}) - (e_2 : \texttt{Nat})$ into the term $\texttt{(ite (>= } \hat{e_1}\ \hat{e_2}\texttt{) (- } \hat{e_1}\ \hat{e_2}\texttt{) 0)}$, and in this term, $\texttt{(- } \hat{e_1}\ \hat{e_2}\texttt{)}$ can be understood as corresponding to $(\uparrow e_1 : \texttt{Int}) - (\uparrow e_2 : \texttt{Int})$, meaning that even when subtraction on the naturals is being translated to SMT-LIB, the "`-`" symbol that appears in the resulting translation still corresponds to a function of type $\alpha = \texttt{Int} \to \texttt{Int} \to \texttt{Int}$.
- If t is a symbol corresponding to a built-in function from SMT-LIB's `Core` or `Int` theories and $t \notin \{\texttt{-}, \texttt{+}, \texttt{*}, \texttt{div}, \texttt{mod}, \texttt{abs}, \texttt{ite}\}$, then t is a function whose output sort is `Bool`. This entails that $\texttt{wf}_\alpha\ t$ is equivalent to `True`.
- If t is an application of of the form $\texttt{(}t_1\ t_2\texttt{)}$, then the Lean term corresponding to t_1 has type $\beta \to \alpha$ and the Lean term corresponding to t_2 has type β. By the inductive hypothesis, t_2 is well-formed and t_1 is either well-formed or a selector function.
  - If t_1 is well-formed, then from the definition of well-formedness on functions, $\texttt{wf}_{\beta\to\alpha}\ t_1 = \texttt{(forall ((y } \hat{\beta}\texttt{)) (=> (wf}_\beta\texttt{ y) (wf}_\alpha\texttt{ (}t_1\texttt{ y))))}$. From this and $\texttt{wf}_\beta\ t_2$, it follows that $\texttt{wf}_\alpha\ \texttt{(}t_1\ t_2\texttt{)}$ as desired.
  - If t_1 is a selector function, then β is a structure or inductive datatype and t_1 has some associated constructor $\hat{c}$. From the definition of well-formedness on structures and inductive datatypes, $\texttt{wf}_\beta\ t_2$ entails $\texttt{(=> (}t_2\texttt{ is } \hat{c}\texttt{) (wf}_\alpha\texttt{ (}t_1\ t_2\texttt{)))}$. LEAN-AUTO's translation procedure guarantees that if $\texttt{(}t_1\ t_2\texttt{)}$ appears in the generated SMT-LIB problem, then t_2 satisfies the tester for t_1's constructor, meaning $\texttt{(}t_2\texttt{ is } \hat{c}\texttt{)}$ holds. From this and the implication entailed by $\texttt{wf}_\beta\ t_2$, it follows that $\texttt{wf}_\alpha\ \texttt{(}t_1\ t_2\texttt{)}$.
- If t is a `forall` or `exists` binder, then $\alpha = \texttt{Prop}$ and $\texttt{wf}_\alpha\ t = \texttt{True}$.
- If t is a `match` binder, then t evaluates to one of the top-level terms among its match cases. None of these top-level terms can be unapplied selectors, so by the inductive hypothesis, they are all well-formed. Since t evaluates to a well-formed term, t itself is well-formed.
- If t is a wrapper expression of the form $\texttt{(! } t' \texttt{ ...)}$, then by the inductive hypothesis, t' is well-formed or a selector function. LEAN-AUTO's translation procedure only produces wrapper expressions of the form $\texttt{(! } t' \texttt{ ...)}$ to indicate named formulas, so t' cannot be a selector function. Therefore, t' is well-formed. Since t is semantically equivalent to t', t is also well-formed.
- The only remaining term forms to consider are `let` binders and `lambda` binders. t is guaranteed to be neither of these because LEAN-AUTO's translation procedure never produces them. `let` binders are never produced because it is never necessary to. `lambda` binders are never produced because they are part of a recent extension to SMT-LIB's logic which LEAN-AUTO does not specifically target.

B Component Evaluation

We evaluate the impact of several design decisions on QUERYSMT's performance by rerunning QUERYSMT's evaluation subject to various modifications. Each row in Table 2 corresponds to a re-evaluation of QUERYSMT with one component of QUERYSMT's default implementation disabled or altered. "Premises" refer to facts recommended by premise selection (these are tailored to the current problem), while "theory lemmas" refer to the small set of properties about integers and natural numbers that `cvc5`'s hints aren't expected to capture (these are fixed across all problems). For more information on the set of theory lemmas used by QUERYSMT, either in the default configuration or in one of the modified configurations, see Appendix C.

Table 2. Benchmark theorems solved by QUERYSMT subject to modifications

	Int Theorems	Nat Theorems	List Theorems
Default	839	869	743
No Hints	472	627	708
No Theory Lemmas	697	625	736
No Set of Support Strategy	721	613	434
Hints not in Set of Support	762	659	733
Premises not in Set of Support	807	838	480
Only Unit Theory Lemmas	821	847	735
With AC Theory Lemmas	828	883	736

In Section 8.2, we discuss the impact of removing hints from QUERYSMT (the "No Hints" row of Table 2 corresponds with the QUERYSMT− row of Table 1). Here, we comment on other design decisions less central to the overall narrative.

The inclusion of theory lemmas appears to play an important role in allowing DUPER to effectively use `cvc5`'s hints. From the fact that more Int theorems are solved with hints but no theory lemmas than with theory lemmas but no hints, we infer that hints retain some utility even in the absence of theory lemmas. But from the fact that QUERYSMT performs nearly identically on Nat theorems in the "No Hints" and "No Theory Lemmas" configurations, we infer that the theory lemmas are essential in practice for applying `cvc5`'s integer theory hints to problems about natural numbers.

The set of support strategy added to DUPER appears to play an important role in limiting the theory lemmas' explosive behavior. We see this most clearly in the performance of the "No Set of Support Strategy" configuration on List theorems. In QUERYSMT's default setting, the only facts excluded from the set of support are theory lemmas, so the only facts impacted by disabling DUPER's set of support strategy are these theory lemmas. From the significant drop in performance from QUERYSMT's default setting to "No Set of Support Strategy" configuration on List theorems, we infer that the theory lemmas exhibit explosive behavior that can effectively be reigned in by excluding them from the set of support. Conversely, from the fact that performance improves when hints and premises are included in the set of support, we infer that hints and premises do not generally exhibit this explosive behavior (or if they do, their relevance to the target goal outweighs the performance hit caused by any explosive behavior).

The exact set of theory lemmas included do not appear to have a large impact on querySMT's performance. This is somewhat surprising given that `cvc5`'s normalization of AC operators is not captured by the hints it produces (see Section 5). From the fact that AC theory lemmas marginally help querySMT's performance on Nat theorems and marginally hurt querySMT's performance on Int theorems, we infer that including AC lemmas does not universally help or hurt querySMT's performance. querySMT seems to do better on some problems with AC lemmas, and worse on others. Most likely, this is not indicative of a real qualitative difference between these two sets of problems. Rather, we suspect that adding or removing AC lemmas is introducing noise that by coincidence causes Duper to do better or worse on some problems.

C Theory Lemmas Used by querySMT

Table 3 contains a list of theory lemmas that are always provided to Duper (but excluded from Duper's set of support). These lemmas are intended to capture properties about integers and natural numbers that `cvc5`'s hints are not expected to cover. Most either relate to casting between Lean's `Nat` and `Int` types or equating terms that `cvc5` views as equivalent. The "D", "Unit", and "AC" columns indicate whether the lemma is included in querySMT's default implementation, "Only Unit Theory Lemmas" implementation, and "With AC Theory Lemmas" implementation respectively. Unit theory lemmas are theory lemmas that resolve to unit clauses when processed by Duper, and AC theory lemmas are lemmas that assert associativity or commutativity properties.

Table 3. Theory Lemmas used by QUERYSMT

Theory Lemma	D	Unit	AC
Nat.zero_le	Y	Y	Y
Int.natCast_nonneg	Y	Y	Y
ge_iff_le	Y	Y	Y
gt_iff_lt	Y	Y	Y
lt_iff_not_ge	Y	Y	Y
le_iff_lt_or_eq	Y	N	Y
Int.ofNat_inj	Y	Y	Y
Int.natAbs_natCast	Y	Y	Y
Int.natAbs_eq	Y	N	Y
Int.natAbs_eq_natAbs_iff	Y	N	Y
Int.ofNat_le	Y	Y	Y
Int.ofNat_lt	Y	Y	Y
Int.ofNat_eq_coe	Y	Y	Y
Int.zero_sub	Y	Y	Y
Int.natAbs_of_nonneg	Y	N	Y
Int.ofNat_natAbs_of_nonpos	Y	N	Y
Int.nonpos_of_neg_nonneg	Y	N	Y
Int.nonneg_of_neg_nonpos	Y	N	Y

Theory Lemma	D	Unit	AC
Int.natCast_add	Y	Y	Y
Int.natCast_mul	Y	Y	Y
Int.natAbs_mul	Y	Y	Y
Int.natCast_one	Y	Y	Y
Int.natCast_zero	Y	Y	Y
Int.natAbs_zero	Y	Y	Y
Int.natAbs_one	Y	Y	Y
Int.ofNat_zero	Y	Y	Y
Int.ofNat_one	Y	Y	Y
Int.mul_assoc	N	N	Y
Int.mul_comm	N	N	Y
Int.add_assoc	N	N	Y
Int.add_comm	N	N	Y
Nat.mul_assoc	N	N	Y
Nat.mul_comm	N	N	Y
Nat.add_assoc	N	N	Y
Nat.add_comm	N	N	Y