crush: Extensible Proof Automation for Lean

Mike He, Dexin Zhang, and Aarti Gupta

Princeton University

Abstract. Lean's rich expressiveness supports higher-order functions, inductive datatypes, and user-defined abstractions, making it challenging to translate proof obligations into first-order Satisfiability Modulo Theory (SMT) queries and reconstruct kernel-checked proofs. We present **crush**, an extensible SMT-backed proof automation framework for Lean. To bridge the gap between Lean's higher-order constructs and the solver's first-order logic, **crush** uses defunctionalization to encode function values and their applications. We formalize this translation for a supported higher-order fragment and prove its soundness. Moreover, **crush** provides metaprogramming interfaces for extending both Lean-to-SMT translation and the reconstruction of Lean proofs from solver certificates without modifying the translation pipeline. We implement **crush** as both a tactic and a backend for other third-party automation systems. We show that across four such use cases that target verification and general proof automation, **crush** discharges more obligations than the competing backends and reconstructs more kernel-checked proofs than the baselines.

Keywords: Interactive Theorem Proving· SMT Solvers· Lean Hammer

1 Introduction

Theorem proving has been widely applied in verification of hardware designs [8, 16, 30], software codebases [1, 17, 19] and cloud services [7, 15, 33]. The success of theorem proving in these domains can be attributed to rigorous improvements along two lines: interactive theorem proving and automated theorem provers (ATP). Interactive theorem proving involves manual formalization and proofs, checked by a proof assistant (PA) such as Rocq [34], Lean [23], Isabelle/HOL [24] and Agda [25]. These tools provide rich type systems in which users can formalize properties with abstract and higher-order constructs. On the other hand, automated theorem provers such as Z3 [11], CVC5 [3], and E-Prover [31] support less expressive logics but offer significant automation in proof search.

Integrating ATPs with proof assistants, as in CoqHammer [10] and SMTCoq [2], combines automated search with kernel-checked proofs. While offering merits from both PA and ATP, these integrations face two main challenges, shown in Table 1: (1) translating expressive proof obligations into the solver's first-order logic and (2) reconstructing kernel-checkable proofs from solvers' outputs. The translation has to accommodate features like polymorphism and higher-order constructs admitted by PAs and provide extensible encodings that

Table 1: Qualitative comparison of goal translation and proof reconstruction. *Poly.* shows support of polymorphic premises; *HO* shows support of higher-order functions; *Ext. encoding* denotes interfaces for custom encodings. *Core-guided* and *Replay* shows whether the tool supports reconstructing ITP-checked proofs from solver certificates (unsat cores and solver-produced proofs, respectively).

		ITP → ATP			ATP → ITP	
ITP	**Tool**	**Poly.**	**HO**	**Ext. encoding**	**Core-guided**	**Replay**
Isabelle/HOL	`smt` [6]	✓	✓	✓	×	✓
Rocq	SMTCoq [2]	×	×	✓	×	✓
	CoqHammer [10]	✓	✓	×	✓	×
Lean	lean-auto [28]*	✓	×	×	×	×
	LeanHammer [36]**	✓	✓	×	✓	×
	lean-smt [21]	✓	×	✓	×	✓
	crush	✓	✓	✓	✓	✓

*SMT backend of lean-auto.
**Duper backend of LeanHammer; its replay support is provided separately by lean-smt.

allow users to customize how new definitions are translated. Proof reconstruction has to produce kernel-checkable derivations from certificates generated by the solvers, which may provide only selected premises (i.e., unsat cores), or rely on solver inference rules that have to be mapped into the PA's proof system.

In Lean specifically, proof automation that leverages power of ATPs follows two broad approaches. The first consists of tactics implemented natively in Lean, represented by **`grind`** [22] and **`aesop`** [20]. These tactics operate directly on Lean expressions and construct proof terms that are checked by the Lean kernel. **`grind`** combines congruence closure, E-matching, constraint propagation, case analysis, and specialized solvers for domains such as linear arithmetic and commutative rings. **`aesop`**, in contrast, constructs a best-first proof-search tree from an extensible set of rules. Rules may be built from theorem applications, constructor introduction, term rewriting, or arbitrary tactics. These tactics are native to Lean, but their automation is limited to the search procedures, registered rules, and decision procedures implemented within Lean.

The second category consists of hammer-style tactics, such as lean-auto [28] and lean-smt [21]. These tactics collect local hypotheses and the negated proof goal, translate the supported fragment of Lean into a solver language such as SMT-LIB [4], and invoke an external SMT solver. This architecture allows Lean to benefit from mature decision procedures and proof-search heuristics developed for ATPs. When the SMT solver reports unsatisfiability (UNSAT), this fact could either be trusted directly, or be witnessed by a reconstructed proof, using solver's certificates, that is subsequently and independently checked by the Lean kernel. The translation boundary of existing approaches suffers from a limitation: Lean constructs, including higher-order functions and user-defined abstractions, must be represented in the solver's supported logic. The applicability and soundness guarantees of these tactics are therefore limited by the coverage and extensibility of both the translation and proof reconstruction procedures.

To address these gaps, we have developed **crush**, an SMT-backed proof automation framework that combines translation support for higher-order functions with user-extensible encodings, and supports both unsat core-guided proof reconstruction and proof replay from solver inference steps. Users can customize how their definitions are encoded and how the resulting solver proofs are reconstructed in Lean. Our design has three key capabilities. First, **crush** supports nondependent higher-order functions, including function-valued arguments, partial applications, and lambda abstractions. Through defunctionalization, it represents these functions as values of uninterpreted SMT sorts and introduces application symbols and equations describing their behavior. We formalize this translation and prove its soundness. Second, we provide an extensible translation interface that allows users to define customized encodings of Lean types, functions, and operators without modifying the core translator. Third, we provide an extensible proof replay interface that allows users to define proof reconstruction procedures in the form of typed rewrite rules for terms introduced by their custom translation and for solver's inference rules, yielding kernel-checkable proofs.

We evaluate **crush** on four benchmark suites: a Curated set of proof obligations developed by us, Cashmere [14, 18], Velvet [13], and PLean [26]. These include 754 verification conditions (VCs) that contain mixed theories. **crush** discharges 709 (94.0%) of the 754 VCs, with 693 (91.9%) successfully reconstructed kernel-checked proofs. In comparison, `auto`, the strongest non-proof-producing tactic, closes only 622 (82.5%), and `grind`, the strongest proof-producing tactic, closes 641 (85.0%). With kernel-checked proofs, **crush** also exceeds lean-smt [21] and Duper [9] by 34.7 and 37.1 percentage points in overall coverage, respectively. In summary, our contributions are:

Extensible translation. We design a pattern-based interface for extending SMT encodings of Lean types and terms. These extensions combine with translation of higher-order constructs with defunctionalization without modifying the core translator. We also formalized soundness of the defunctionalization pass.

Extensible proof replay. We design interfaces for interpreting terms in SMT solver proofs and reconstructing their inference steps in Lean. Users can extend the procedure to support custom encodings and additional solver inference rules.

The crush framework. We implement these ideas as a proof automation framework **crush** in Lean. **crush** is publicly available on GitHub[1].

2 Overview

Fig. 1 summarizes the end-to-end **crush** pipeline. Given a Lean goal, its hypotheses, and lemmas, **crush** first prepares a set of propositions by negating the goal, collecting all local hypotheses, and adding user-supplied lemmas. The preprocess pass performs normalization, monomorphization and a bounded quantifier instantiation on the collected formulas. Appendix A describes these standard pre-processing steps in more detail. **crush** then translates processed for-

[1] https://github.com/AD1024/lean-crush

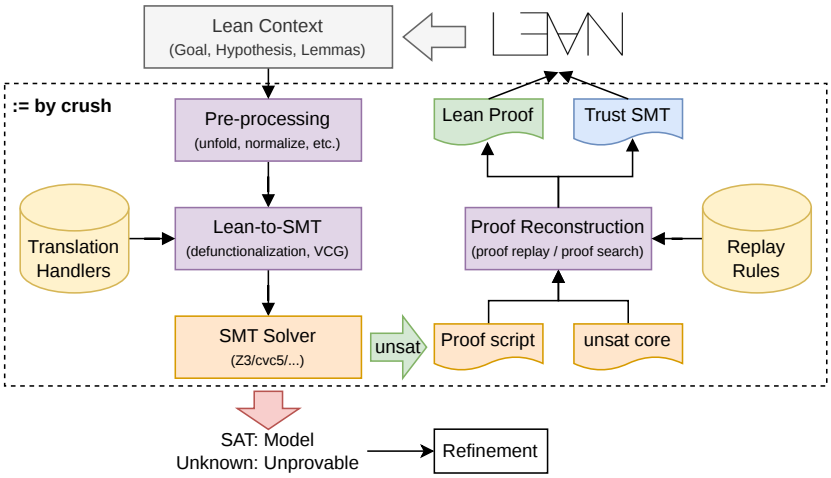

Fig. 1: The **crush** pipeline.

mulas into an SMT query, where hypotheses and lemmas are conjoined with the negated goal, and employs defunctionalization to encode higher-order functions and their applications in first-order logic. If the solver reports UNSAT, **crush** can either trust the solver or construct a Lean proof from an unsat core or by replaying the inference steps produced by the solver. If solver reports SAT (with a model) or Unknown, the user may leverage external tools to fix or refine the proof.

2.1 Running Example

Fig. 2 defines a function NAND over two bitvectors x, y (with width 4), and states a property for **crush** to prove. The hypothesis on line 7 specifies that applying `run` on function `f` with `x` gives the result of applying `f` on `x`. The function on line 9 applies a mask to a given input y. Its two inner NAND applications produce the same complemented conjunction; the outer NAND recovers the conjunction of the input and mask. Thus, the left-hand-side application computes the masked input.

```
1  def Nand (x y : BitVec 4) : BitVec 4 :=
2    ~~~(x &&& y)
3
4  theorem computes_masked
5      (run : (BitVec 4 → BitVec 4) →
6        BitVec 4 → BitVec 4)
7      (hrun : ∀ f x, run f x = f x)
8      (x mask : BitVec 4) :
9      run (fun y => Nand (Nand y mask)
10                        (Nand y mask)) x =
11       x &&& mask := by
12   crush
```

Fig. 2: The running example: a callback that applies a bitvector mask.

2.2 Extensible Translation

crush supports extensible translation via rewrite rules to define SMT encodings for Lean terms. Fig. 3 shows an example encoding for `Nand`: the pattern on line 2 matches applications of `Nand`. The sub-patterns `(term x)` and `(term y)` match the arguments and bind the translated results to `x` and `y`. The right hand side pattern (line 3) defines the emitted SMT encoding that applies the solver's `bvnand` operation with `x` and `y`, written in a DSL of an SMT-LIB subset embedded in Lean. §3 discusses this interface in more detail.

```
1 register_lowering term <<
2   (Nand (term x) (term y)) =>
3     (smt| (bvnand $x $y))
4 >>
```

Fig. 3: A custom encoding from `Nand` to solver's `bvnand`.

2.3 Higher-Order Functions

The translation is defined for bitvector operations, but the higher-order function must also be represented in first-order SMT. To encode function values in first-order SMT, **crush** introduces a fresh first-order sort for each function type and a fresh operator for its application. Then, it emits axioms that assign interpretations to the sort with respect to its definition in Lean. This defunctionalization combines with the extensible translation to encode higher-order functions and bitvector operations into theories supported by the solver. §4 presents the encoding in detail and proves its soundness.

2.4 Extensible Proof Reconstruction

The resulting query can now be handled by an SMT solver. If it is unsatisfiable, **crush** supports two paths to obtain a kernel-checked proof. Reconstruction from an unsat core selects the relevant Lean hypotheses and uses Lean automation tactics (e.g., **`grind`**, **`omega`**) to rederive a proof. Alternatively, **crush** can replay inference steps produced by the solver. **crush** currently supports the Alethe format [32]. As an inverse of Lean-to-SMT translation, **crush** also supports extensible proof reconstruction via rewrite rules that map solver inference steps to a Lean term or proof.

```
1 register_crush_replay term <<
2   (bvnand (term x : BitVec _)
3           (term y : BitVec _)) =>
4     ~~~(x &&& y)
5 >>
6
7 set_option crush.backend "cvc5"
8 set_option crush.trust "reconstruct"
9 set_option crush.reconstruct "alethe"
```

Fig. 4: Extending proof replay and selecting Alethe reconstruction.

Fig. 4 shows a rewrite rule for the running example: the pattern on lines 2–3 matches applications of `bvnand` operation and its arguments. The sub-patterns (e.g., `(term x : BitVec _)`) on the arguments match bitvectors of arbitrary known widths. On the right hand side (line 4), the pattern defines the translated Lean term. Finally, lines 7–9 configure **crush** to enable proof reconstruction from Alethe scripts. §5 discusses the reconstruction procedures and the replay extension interface in detail.

```
register_lowering result-type with ctx <<
  Bool => do
    match ctx.fn, ctx.args with
    | .const ``Bool.not _, #[p] =>
        return some (smt| (not $(← ctx.emitTerm p)))
    | .const ``Bool.xor _, #[p, q] =>
        return some (smt| (xor $(← ctx.emitTerm p) $(← ctx.emitTerm q)))
    | _, _ => return none
>>
```

Fig. 6: Grouping Boolean operations in a type-indexed lowering.

3 Extensible Lean-to-SMT Translation

crush provides extensible translation from Lean terms and types to SMT terms and sorts: unfolding a definition exposes the defining equation encoded in the SMT query. Additionally, users may define custom encoding of terms and types as rewrite rules that match head symbols of a term or type application.

3.1 Definition Unfolding

The `crush_unfold` attribute makes the defining equations of a function available whenever that function is called. Fig. 5 shows an example of an integer clamp: the annotation on line 1 directs **crush** to include the definition of `clamp` in the emitted query. Then, the SMT solver proves the goal by the hypothesis on line 7 and the definition of `clamp`.

```
@[crush_unfold]
def clamp (x lo hi : Int) : Int :=
  if x < lo then lo
  else if hi < x then hi else x

theorem clamp_bounds (x lo hi : Int)
    (h : lo ≤ hi) :
    lo ≤ clamp x lo hi ∧
      clamp x lo hi ≤ hi := by
  crush
```

Fig. 5: Definition unfolding on functions.

3.2 Head- and Type-Indexed Term Translation

Let $e^\flat$ denote the SMT term produced from a Lean expression e, and $\tau^\flat$ denote the SMT sort used to represent a Lean type τ.

Head-indexed translation. Translations on Lean terms can be defined as rewrite rules that match the head of an application. For example, in Fig. 3, the pattern on line 2 matches applications of `Nand` and translates its two operands, producing $(\texttt{Nand}\ x\ y)^\flat = \mathsf{bvnand}(x^\flat, y^\flat)$.

Type-indexed translation. Rewrites can also be grouped based on the type of Lean terms. For instance, in Fig. 6, the `Bool` pattern on line 2 matches both `Bool.not` and `Bool.xor`. The handler then distinguishes these operations by their application heads and arguments. Lines 4 and 6 recursively translate the operands of negation and exclusive-or and construct the corresponding SMT terms. Line 8 ignores other Boolean terms, and the translation procedure will try other rewrite rules if the triggered rule returns `none`.

3.3 Coordinating Sort and Term Translation

When users define custom types τ in Lean, **crush** has to handle translations for both terms with type τ and the encoding of τ. With the extensible interface, **crush** allows users to define rewrite rules over type applications to define encodings for user-defined Lean types.

For example, suppose the user defines a Lean type TotalMap κ ν, whose values map keys of type κ to values of type ν. Fig. 7 shows the custom encoding of TotalMap κ ν into the Array sort in SMT. The semantics of rewrite rules over Lean types is similar to rewrite rules over Lean terms. The left-hand-side pattern on line 2 matches the key and value types, and the right-hand-side pattern on line 3 defines the corresponding SMT encoding. Additionally, rewrite rules on lines 7 and 12 define encodings for applications of get and set functions associated with TotalMap, translating get to **select** and set to **store**.

```
1  register_lowering sort <<
2    (TotalMap (sort key) (sort value)) =>
3      (smt| (Array $key $value))
4  >>
5
6  register_lowering term <<
7    (TotalMap.get _ _ (term m) (term k)) =>
8      (smt| (select $m $k))
9  >>
10
11 register_lowering term <<
12   (TotalMap.set _ _ (term m)
13     (term k) (term v)) =>
14     (smt| (store $m $k $v))
15 >>
```

Fig. 7: Encoding of TotalMap.

crush performs sort-checking on queries emitted by handlers before invoking the solver. This light-weight checking catches incompatible sort and term translations, such as erroneously applying an array operation (e.g., **select**) to an integer. However, **crush** does not check that the SMT encoding is semantically correct: the user is responsible for ensuring the correctness of extended translations. For this example, since TotalMap is encoded as SMT Arrays, the user should ensure that TotalMap.get and TotalMap.set are consistent with the axioms of the array theory (i.e., $\mathsf{get}(\mathsf{set}(m, k, v), k) = v$).

4 Higher-Order Translation via Defunctionalization

crush uses defunctionalization to encode higher-order functions in first-order SMT, introducing uninterpreted sorts for function types and replacing function values and their applications with uninterpreted symbols constrained by axioms. Our translation procedure supports the encoding for a subset of Lean with *monomorphic*, *nondependent* higher-order constructs. We model this translation procedure in Lean and formalize its soundness.

4.1 Defunctionalization

Function values and application. For a nondependent function type $\theta = \sigma_1 \to \cdots \to \sigma_n \to \tau$, where τ is not a function type, **crush** introduces an

uninterpreted sort Fn_θ whose elements represent function values. We refer to these uninterpreted sorts as Fn sorts. Its application symbol has signature

$$\mathsf{app}_\theta : \mathsf{Fn}_\theta \times \sigma_1^\flat \times \cdots \times \sigma_n^\flat \to \tau^\flat.$$

An application $f\, a_1 \cdots a_n$ becomes $\mathsf{app}_\theta(f, a_1^\flat, \ldots, a_n^\flat)$, where f is a function symbol. Quantification over a function therefore becomes ordinary first-order quantification over the corresponding uninterpreted sort Fn.

Lambdas and captured variables. For each lambda $\ell = \lambda x_1 \cdots x_n.\, e$, **crush** introduces a fresh uninterpreted symbol clos_ℓ of sort $\mathsf{Fn}_{\theta_\ell}$. Then, **crush** generates an axiom about the closure application according to its definition, where $\bar{y}$ are variables captured by ℓ:

$$\forall \bar{y}, \bar{x}. \quad \mathsf{app}_\theta(\mathsf{clos}_\ell(\bar{y}), \bar{x}) = e^\flat.$$

Here the variables $\bar{x}, \bar{y}$ range over their translated sorts, and the body is translated under the same bindings. The closure symbol clos_ℓ is uninterpreted, and its semantics are axiomatized according to its definition.

Partial application and function equality. **crush** applies η-expansion[2] to partial applications, transforming them into closures. Then, it reuses the encoding for lambdas to encode the η-expanded terms. When the query compares function values for equality, **crush** additionally emits an extensionality axiom for the corresponding Fn sort.

Guarded Sub-types. For types represented by a subset of an SMT sort, such as Nat within Int, the encoding represents inhabitants of the Lean types and constrains the application results accordingly. For example, **crush** reuses SMT solver's integer theory to represent Nat, and for each query involving a Nat x, the assertion will be guarded by $x \geq 0$.

Encoding the running example. Let B denote the SMT sort of $\mathsf{BitVec}\ 4$ and $\theta = \mathsf{BitVec}\ 4 \to \mathsf{BitVec}\ 4$. The application symbol has signature $\mathsf{app}_\theta : \mathsf{Fn}_\theta \times \mathsf{B} \to \mathsf{B}$. For the fixed mask m in Fig. 2, the closure on line 9, denoted by ℓ, is represented by the nullary closure symbol $\mathsf{clos}_\ell : \mathsf{Fn}_\theta$. Combining the closure body with the custom translation in Fig. 3 gives

$$\forall y : \mathsf{B}. \quad \mathsf{app}_\theta(\mathsf{clos}_\ell, y) = \mathsf{bvnand}(\mathsf{bvnand}(y, m), \mathsf{bvnand}(y, m)).$$

The `run` function on line 5 becomes an uninterpreted symbol $\widehat{\mathsf{run}} : \mathsf{Fn}_\theta \times \mathsf{B} \to \mathsf{B}$. Its hypothesis and the negated goal become, respectively,

$$\begin{aligned} &\forall f : \mathsf{Fn}_\theta,\, y : \mathsf{B}. \quad \widehat{\mathsf{run}}(f, y) = \mathsf{app}_\theta(f, y), \\ &\widehat{\mathsf{run}}(\mathsf{clos}_\ell, x) \neq \mathsf{bvand}(x, m). \end{aligned}$$

Instantiating the hypothesis with clos_ℓ and x, and the closure equation with x completes the proof by contradiction.

Syntax of higher-order language (HO)

$$\begin{aligned}\sigma, \tau &::= \mathsf{Bool} \mid \beta \mid \sigma \to \tau,\\ e &::= x \mid c \mid \mathsf{true} \mid \mathsf{false}\\ &\quad \mid \lambda x : \tau.\, e \mid e\, e\\ &\quad \mid e = e \mid \neg e \mid e \circ e\\ &\quad \mid Q x : \tau.\, e.\end{aligned}$$

Syntax of first-order language (FO)

$$\begin{aligned}s &::= \mathsf{Bool} \mid \beta \mid \mathsf{Fn}_\theta,\\ k &::= \widehat{c} \mid \mathsf{app}_\theta \mid \mathsf{clos}_\ell,\\ t &::= x \mid \mathsf{true} \mid \mathsf{false}\\ &\quad \mid k(t_1, \ldots, t_n)\\ &\quad \mid t = t \mid \neg t \mid t \circ t\\ &\quad \mid Q x : s.\, t.\end{aligned}$$

$$\circ \in \{\wedge, \vee, \to, \leftrightarrow\} \qquad Q \in \{\forall, \exists\}$$

Denotational semantics ($[\![\cdot]\!]_I$)

HO Sorts

$$\begin{aligned}[\![\mathsf{Bool}]\!]_M &= \mathsf{Prop},\\ [\![\beta]\!]_M &= M(\beta) \text{ s.t. } M(\beta) \neq \varnothing,\\ [\![\sigma \to \tau]\!]_M &= [\![\sigma]\!]_M \to [\![\tau]\!]_M.\end{aligned}$$

FO Sorts

$$\begin{aligned}[\![\mathsf{Bool}]\!]_N &= \mathsf{Prop},\\ [\![\beta]\!]_N &= N(\beta) \text{ s.t. } N(\beta) \neq \varnothing,\\ [\![\mathsf{Fn}_\theta]\!]_N &= N(\mathsf{Fn}_\theta) \text{ s.t. } N(\mathsf{Fn}_\theta) \neq \varnothing.\end{aligned}$$

Constants, abstractions, and applications

$$\begin{aligned}[\![c]\!]_M(\rho) &= M(c),\\ [\![\lambda x : \tau.\, e]\!]_M(\rho) &= (\mathsf{fun}\ v \Rightarrow [\![e]\!]_M(\rho[x \mapsto v])),\\ [\![e_1\, e_2]\!]_M(\rho) &= [\![e_1]\!]_M(\rho)([\![e_2]\!]_M(\rho)),\\ [\![k(t_1, \ldots, t_n)]\!]_N(\rho) &= N(k)([\![t_1]\!]_N(\rho), \ldots, [\![t_n]\!]_N(\rho)).\end{aligned}$$

Variables and logical connectives (HO and FO)

$$\begin{aligned}[\![x]\!]_I(\rho) &= \rho(x), \qquad [\![\mathsf{false}]\!]_I(\rho) = \mathsf{False},\\ [\![\mathsf{true}]\!]_I(\rho) &= \mathsf{True}, \qquad [\![\neg u]\!]_I(\rho) = \neg [\![u]\!]_I(\rho).\end{aligned}$$

$$\begin{aligned}[\![u = v]\!]_I(\rho) &= ([\![u]\!]_I(\rho) = [\![v]\!]_I(\rho)),\\ [\![u \circ v]\!]_I(\rho) &= [\![u]\!]_I(\rho) \circ [\![v]\!]_I(\rho),\\ [\![Q x : \alpha.\, u]\!]_I(\rho) &= Q v : [\![\alpha]\!]_I.\ [\![u]\!]_I(\rho[x \mapsto v]).\end{aligned}$$

Fig. 8: HO syntax (left), FO syntax (right), and their denotational semantics.

4.2 Soundness of Defunctionalization

In this section, we describe the formalization of the defunctionalization pass and show the proof skeleton of its soundness. Full formalization in Lean is available in our artifact[3]. We first formalize the higher-order (HO) and first-order (FO) languages shown on the left and right of Fig. 8, respectively. Both languages have Boolean literals, equality, logical connectives, and quantifiers. HO models the fragment of Lean supported by this transformation. In HO, β ranges over base (non-arrow) sorts, σ, τ over HO sorts, e over terms, and c over typed constants. FO uses FO sorts s (including the same base sorts β as HO) and terms t, replacing each HO function sort θ by Fn_θ as described earlier. The symbols k are uncurried constant symbols $\widehat{c}$, application symbols app_θ, and closure symbols clos_ℓ. Each has type $s_1 \times \cdots \times s_n \to s$ and is fully applied, with $n = 0$ for nullary symbols.

[2] Converting a function f to its normal form, i.e., $f = \lambda x.\, f x$

[3] Crush/Metatheory

$$\theta = \sigma_1 \to \cdots \to \sigma_n \to \tau, \quad \tau \in \{\mathsf{Bool}, \beta\}.$$

Function application

$$\frac{f : \theta \qquad a_i : \sigma_i \qquad \mathcal{F}(a_i) = (a_i^\flat, C_{a_i}, E_{a_i}) \quad (1 \le i \le n)}{\mathcal{F}(f\, a_1 \cdots a_n) = \left(\mathsf{app}_\theta(f, \bar{a}^\flat),\ \bigcup_i C_{a_i},\ \bigcup_i E_{a_i}\right)}$$

Closure construction

$$\frac{\ell = \lambda \bar{x}.\, e : \theta \qquad e : \tau \qquad \bar{y} = \mathsf{FV}(\ell) \qquad \mathcal{F}(e) = (e^\flat, C_e, E_e)}{\mathcal{F}(\ell) = (\mathsf{clos}_\ell(\bar{y}),\ C_e \cup \{K_\ell\},\ E_e)}$$

$$K_\ell = (\forall \bar{y}, \bar{x}.\ \mathsf{app}_\theta(\mathsf{clos}_\ell(\bar{y}), \bar{x}) = e^\flat).$$

Function equality

$$\frac{u, v : \theta \qquad \mathcal{F}(u) = (u^\flat, C_u, E_u) \qquad \mathcal{F}(v) = (v^\flat, C_v, E_v)}{\mathcal{F}(u = v) = (u^\flat = v^\flat,\ C_u \cup C_v,\ E_u \cup E_v \cup \{\mathsf{ext}_\theta\})}$$

$$\mathsf{ext}_\theta = (\forall f, g : \mathsf{Fn}_\theta.\ (\forall \bar{x}.\ \mathsf{app}_\theta(f, \bar{x}) = \mathsf{app}_\theta(g, \bar{x})) \to f = g).$$

Fig. 9: Defunctionalization (selected rules).

Definition 1 (Denotation and satisfaction). *Let I be a model of either HO or FO. We write $[\![\alpha]\!]_I$ for the carrier of sort α, i.e., the set of values interpreting α in model I. We write $[\![u]\!]_I(\rho)$ for the denotation of term u in model I under valuation ρ. We use $\varnothing$ to denote an empty valuation. For a collection T of closed formulas in the corresponding language, define*

$$I \models T \quad \iff \quad \forall \varphi \in T.\ [\![\varphi]\!]_I(\varnothing).$$

A higher-order model M assigns nonempty carriers $M(\beta)$ to base sorts and values $M(c)$ to constants; arrow sorts denote full function spaces. A first-order model N assigns arbitrary nonempty carriers to base sorts and Fn sorts, and interpretations $N(k)$ to symbols. For denotational semantics in Fig. 8, we write I for a model of either language, u, v for its terms, and α for a sort in that language. A valuation ρ assigns values to variables. Boolean terms denote Lean propositions (Prop), and quantifiers extend the valuation with a value from the interpretation of their binder sort. The sort encoding $\tau^\flat$ leaves Boolean and base sorts unchanged, with $\theta^\flat = \mathsf{Fn}_\theta$ for an HO function sort θ.

Definition 2 (Defunctionalization). *Defunctionalization maps a well-typed higher-order term $e : \tau$ to $\mathcal{F}(e) = (e^\flat, C_e, E_e)$, where $e^\flat : \tau^\flat$ is a first-order term with first-order sort $\tau^\flat$ and C_e, E_e are sets of closed closure equations and extensionality axioms, respectively. Core definitions are given in Fig. 9. Complete definitions are shown in Appendix B.1.*

In Fig. 9, θ has a complete argument list $\bar{\sigma}$ and non-function result τ. Each $a_i^\flat$ is the first-order term returned by the recursive call $\mathcal{F}(a_i)$, including defunctionalization when a_i has function type. Application of a function variable $f : \theta$ passes these terms to app_θ and accumulates their generated axioms.

For a lambda $\ell = \lambda \bar{x}.\, e$, the closure symbol takes its free variables $\bar{y} = \mathsf{FV}(\ell)$, and K_ℓ specifies its behavior using the recursively translated non-function body e. Variables of Fn sorts, constants, and partial applications are η-expanded over their remaining arguments and use the same closure construction. Appendix B.1 gives the generalized rule, including lambdas whose bodies still have function type. Equality at function type θ additionally contributes ext_θ that encodes functional extensionality. Logical connectives and quantifiers preserve their operators, replacing quantifications to the corresponding first-order sorts.

Definition 3 (First-order encoding). *For a finite collection T of closed higher-order formulas, let $(\varphi^\flat, C_\varphi, E_\varphi) = \mathcal{F}(\varphi)$ and define*

$$\mathcal{D}(T) = \bigcup_{\varphi \in T} \big(C_\varphi \cup E_\varphi \cup \{\varphi^\flat\}\big).$$

Definition 4 (Canonical model extension). *For each lambda ℓ with free variables $\bar{y} = (y_1, \ldots, y_m)$, let $\rho_{\ell,\bar{v}}$ be the valuation defined by $\rho_{\ell,\bar{v}}(y_i) = v_i$ for $1 \le i \le m$. The extension $M^\star$ of M interprets the generated symbols by*

$$\begin{aligned} [\![\mathsf{Fn}_\theta]\!]_{M^\star} &= [\![\theta]\!]_M, \\ M^\star(\mathsf{app}_\theta)(f, \bar{a}) &= f\, a_1 \cdots a_n, \\ M^\star(\mathsf{clos}_\ell)(\bar{v}) &= [\![\ell]\!]_M(\rho_{\ell,\bar{v}}). \end{aligned}$$

Since corresponding HO and FO sorts have the same carriers in M and $M^\star$, we use ρ for both valuations.

Lemma 1 (Preservation of denotations). *For every well-typed higher-order term e, model M, and valuation ρ, with $M^\star$ as in Definition 4,*

$$[\![e^\flat]\!]_{M^\star}(\rho) = [\![e]\!]_M(\rho).$$

Proof sketch. We proceed by induction on typed terms. In the lambda case, we use the fact that denotation depends only on free variables, so reconstructing the captured values suffices. The full proof is shown in Appendix B.3.

Lemma 2 (Validity of generated axioms). *For every model M and well-typed term e,*

$$M^\star \models C_e \cup E_e.$$

Proof sketch. We establish closure equations using Definition 4 and Lemma 1. In $M^\star$, the Fn sorts are interpreted as function spaces and the application symbols as function application, so the extensionality axioms follow from function extensionality. The full proof appears in Appendix B.4.

We combine these properties to obtain the following result.

Theorem 1 (Soundness). *For every finite collection T of closed higher-order formulas, if $\mathcal{D}(T)$ is unsatisfiable, then T is unsatisfiable.*

Proof sketch. We argue by contradiction. If $M \models T$, Lemma 1 makes every translated formula true in $M^\star$, while Lemma 2 supplies all auxiliary axioms. Thus $M^\star \models \mathcal{D}(T)$, contradicting its unsatisfiability. The full proof is shown in Appendix B.5.

5 Proof Reconstruction

For a goal G in Lean, **crush** emits the translated hypotheses and $\neg G$ to the solver. When the solver reports SAT, it provides a model of the encoded first-order formula, possibly spurious if an opaque definition was not unfolded; when it reports UNKNOWN, the Lean goal is left unresolved. When it reports UNSAT, the solver may provide an UNSAT core, a proof script, or both. In the UNSAT case, **crush** attempts to reconstruct a kernel-checked proof through two approaches: use the UNSAT core to select premises and search the proof via tactics in Lean, or replay the solver's inference steps (proof scripts), translating them into Lean proof terms.

5.1 Core-Guided Reconstruction

An UNSAT core contains assertions sufficient for a contradiction, usually much smaller than the input. **crush** maps them back to Lean hypotheses and searches for a proof by contradiction from the hypotheses and the negated goal leveraging existing proof automation tactics (e.g., `omega` or `grind`). Appendix C provides details for the search procedure. To extend search procedure, users may mark a theorem as a reconstruction hint via `@[crush_reconstruct]`. When the current goal matches the conclusion of a marked theorem, it is applied to the goal, and **crush** proves the premises recursively.

5.2 Extensible Solver Inference Replay

Alternatively, **crush** can replay the proof script generated by SMT solvers to reconstruct kernel-checked Lean proofs. A proof script is a list of inference steps, where each inference step records a conclusion, a rule, and any premises or arguments used by the rule to derive the conclusion. **crush** relies on rewrite rules to translate terms and inference steps in proof scripts into Lean term and Lean proofs. Users may extend those rewrite rules via **crush**'s extensible and type-aware interfaces. Appendix C provides the details of the replay procedure.

Term decoding. Users may specify translations from SMT operators in proof scripts into Lean terms via `register_crush_replay term` command. The translated Lean terms are further type-checked by the Lean kernel. As an example, Fig. 4 defines a handler for the SMT operator bvnand, which requires

Alethe inference

```
(step s
  (cl (= (bvnot (bvnot a)) a))
  :rule rare_rewrite
  :args ("bv-not-idemp" a))
```

Lean inference handler

```
register_crush_replay rule <<
  (rare_rewrite "bv-not-idemp"
    (term x : BitVec 4)) =>
  by exact BitVec.not_not (b := x)
>>
```

Fig. 10: An inference from the running example's certificate and a replay handler.

the arguments to have Lean type BitVec and translates the expression into a complemented conjunction.

Solver inference replay. Users may define translations of inference steps into a Lean proof with `register_crush_replay rule` command. For example, the left of Fig. 10 shows a use of `"bv-not-idemp"` rewrite rule in the solver's inference step, which rewrites a SMT term $\mathsf{bvnot}(\mathsf{bvnot}(a))$ into a and asserts its equality. The right of Fig. 10 registers a rule to translate `"bv-not-idemp"` in the proof script into a Lean proof of $\sim(\sim x) = x$ (where $\sim$ denotes bitwise complement in Lean), using the existing lemma `BitVec.not_not`.

Current limitations. The proof replay procedures in **crush** rely on the SMT solver's ability to produce a proof script. For CVC5, its Alethe proof generation does not cover all theories it can solve. Moreover, the built-in rules in **crush**'s replay procedures may not cover every SMT encoding or inference step. In practice, these gaps can be filled by additional user-supplied rules.

6 Evaluation

We compare **crush** against related proof automation frameworks on metrics for verification coverage, runtime, and proof reconstruction. We first compare overall coverage and runtime (§6.1) for each benchmark suite, and then compare **crush** pair-wise with each baseline framework on the VCs solved by both (§6.2). Finally, we examine how core-guided reconstruction and solver inference replay contribute to obtaining kernel-checked proofs (§6.3).

Experimental setup. The benchmark comprises 754 VCs from four suites: Curated (20), Cashmere [18] (38), Velvet [13] (504), and PLean [26] (192). We compare **crush** with lean-auto (`auto`) [28], Duper [9], lean-smt [21], and Grind (`grind`) [22]. Curated is a standalone collection of Lean goals developed by us, covering higher-order functions, datatypes, arithmetic, bitvectors, and strings. Cashmere, Velvet and PLean use Loom [14] in their VC generation. Cashmere and Velvet are case studies of Loom, covering VCs from program verification. PLean covers verification of distributed protocols written in P [12] language embedded in Lean. We plug **crush** and the baselines into Loom as SMT backends to discharge the generated VCs. We evaluate **crush** both with SMT trusted (i.e., without proof reconstruction) and with kernel-checked proof reconstruction. The former trusts the solver's verdict and, like `auto`, uses an axiom to close the goal.

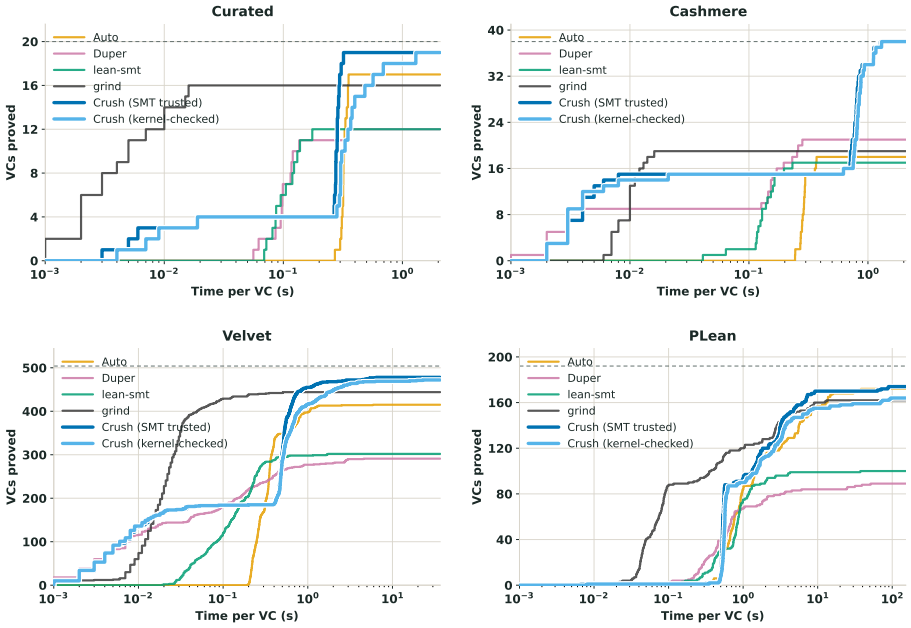

Fig. 11: Verification coverage against time per VC.

The latter requires a kernel-checked proof of the original goal and combines core-guided reconstruction with replay of the solver's inference steps recorded in the Alethe proof format (*Alethe replay*). In this case, **crush** is considered successful if it produces a kernel-checked proof for the original goal.

All experiments ran on an Apple M1 Pro CPU with 32 GB of memory. We use cvc5 1.3.4 as the SMT backend. We use Duper revision `ca7c5862`, lean-smt revision `e5025665`, and Lean 4.32.2 (except for **`auto`** on Cashmere, Velvet, and PLean, for which we retain Lean 4.24.0 to preserve the existing Loom–lean-auto integration). We set a limit of 5 seconds on each solver call and a limit of 1M Lean heartbeats on each tactic invocation. We measure the time for each tactic on each VC, including translation, solver execution, and proof reconstruction.

6.1 Verification Coverage and Runtime

Fig. 11 shows the number of VCs discharged within each time threshold per VC. Here, the x-axis shows the time threshold in seconds for each VC on a log scale; the y-axis shows the number of VCs discharged within that time threshold. The dashed horizontal line marks the total number of VCs in each benchmark suite.

Across the four benchmarks, **crush** discharges 709 VCs (94.0%) when trusting the solver's results, and 693 (91.9%) when enabling proof reconstruction. Among the compared baselines, **`grind`** achieves the highest coverage (641 VCs), followed by **`auto`** (622), lean-smt (431), and Duper (413). *Notably,* ***crush*** *achieves*

Table 2: Head-to-head comparison with **crush** using proof reconstruction. Times are mean ± standard deviation across VCs discharged by both tools, after averaging repetitions per VC.

		# VCs Discharged			Time on Both (ms)	
Benchmark	**Baseline**	**crush only**	**Baseline only**	**Both**	**crush**	**Baseline**
Curated (20 VCs)	`auto`	2	0	17	$365.0_{\pm 298.1}$	$322.2_{\pm 20.1}$
	Duper	8	1	11	$313.1_{\pm 191.1}$	$98.3_{\pm 24.1}$
	lean-smt	7	0	12	$404.2_{\pm 338.6}$	$106.8_{\pm 31.5}$
	`grind`	3	0	16	$371.4_{\pm 309.3}$	$5.5_{\pm 4.8}$
Cashmere [18] (38 VCs)	`auto`	20	0	18	$235.9_{\pm 395.8}$	$294.0_{\pm 36.0}$
	Duper	17	0	21	$252.9_{\pm 407.1}$	$109.3_{\pm 102.6}$
	lean-smt	21	0	17	$156.5_{\pm 352.4}$	$135.6_{\pm 44.2}$
	`grind`	19	0	19	$223.6_{\pm 388.4}$	$9.8_{\pm 2.7}$
Velvet [13] (504 VCs)	`auto`	70	13	402	$598.5_{\pm 1400.7}$	$372.5_{\pm 215.2}$
	Duper	184	3	288	$287.0_{\pm 483.2}$	$220.3_{\pm 517.4}$
	lean-smt	180	10	292	$611.4_{\pm 1584.3}$	$142.1_{\pm 95.9}$
	`grind`	37	9	435	$460.8_{\pm 595.6}$	$26.1_{\pm 26.8}$
PLean [26] (192 VCs)	`auto`	2	10	162	$4517.8_{\pm 14086.1}$	$3380.3_{\pm 5970.0}$
	Duper	75	0	89	$644.6_{\pm 284.3}$	$2755.4_{\pm 8169.2}$
	lean-smt	65	1	99	$1611.3_{\pm 9096.7}$	$1303.1_{\pm 3731.4}$
	`grind`	2	0	162	$4493.5_{\pm 14089.6}$	$1206.5_{\pm 2327.6}$

the highest overall coverage even when kernel-checked proofs are required. The runtime curves show a tradeoff: `grind` discharges many VCs at shorter time thresholds, while **crush** achieves higher overall coverage. The advantage also varies by benchmark suites. On PLean, `auto` discharges more VCs than kernel-checked **crush** (172 versus 164), although trusted **crush** discharges 174.

6.2 Pairwise Comparison with crush

Table 2 compares **crush** (with proof reconstruction) against each baseline on the same VCs. Runtimes are reported as mean ± standard deviation over VCs discharged by both tools. **crush** discharges more VCs in all pairs except against `auto` on PLean. The trusted-mode comparison is in Table 5, Appendix D.

The pairwise results show that **crush** can prove many obligations missed by the baselines. Across the four suites, **crush** proves 61 VCs that `grind` cannot close, while `grind` proves 9 that **crush** cannot close. On Velvet, **crush** proves 184 VCs missed by Duper and 180 missed by lean-smt; these baselines prove only 3 and 10 VCs, respectively, that **crush** misses. On VCs solved by both tools, **crush** has a higher mean runtime in most comparisons, and `grind` is faster on all four suites. The tradeoff varies by workload: on PLean, **crush** is slower on average than lean-smt (1611.3 versus 1303.1 ms) on the 99 VCs, but it also proves 65 VCs that lean-smt misses.

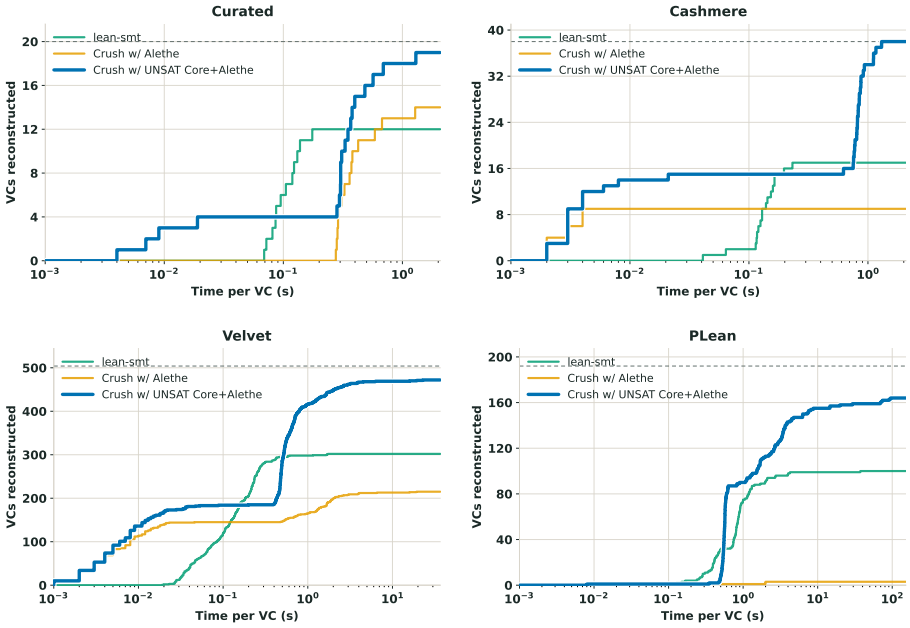

Fig. 12: Kernel-checked proof reconstruction coverage over time per VC.

Table 3: Proof reconstruction failures among VCs discharged by SMT.

Method	Failure mode	VCs
Alethe replay	Certificate generation	431
	Term decoding	1
	Total	**432 / 503**
Combined	Certificate generation and core-guided reconstruction	11
	Lean heartbeat limit	2
	Tactic failure during replay	3
	Total	**16 / 503**

We observe that reconstruction failures mainly arise when cvc5 cannot produce a proof script and core-guided reconstruction also fails to recover a proof. We examine these cases in more detail in the next section.

6.3 Proof Reconstruction

We evaluate how combining Alethe replay with core-guided reconstruction affects proof coverage, examine the remaining failure modes, and report the manual effort needed to define replay handlers.

Reconstruction coverage. Fig. 12 shows the number of VCs that have proofs reconstructed by **crush** and lean-smt in each case study, with the same x- and

Table 4: Implementation size of built-in Alethe replay extensions.

Category	Handlers	Handler LoC	Lemma LoC	Total
Assumption normalization	2	16	34	50
Logic and equality	10	47	10	57
Quantifiers	1	3	22	25
Arithmetic	17	68	311	379
Bitvectors	13	72	337	409
Strings	12	50	83	133
Total	**55**	**256**	**797**	**1,053**

y-axes as in Fig. 11. In total, **crush** closes 693 VCs when combining Alethe replay with core-guided reconstruction, compared with 241 using Alethe replay alone and 431 for lean-smt. Adding core-guided reconstruction therefore enables **crush** to close more VCs.

Failure analysis for Alethe replay. We further categorized the failure modes to identify specific limitations in the Alethe replay, and which limitations were overcome by core-guided reconstruction. Solver certificate generation caused 431 of the replay failures among the 503 VCs discharged by SMT. Although cvc5 concludes unsatisfiability, it does not fully support proof generation in Alethe. For example, it currently cannot generate a proof for the theory of arrays, which is used in the Cashmere and Velvet suites. Here, core-guided reconstruction often allows **crush** to obtain kernel-checked proofs even when the solver's proof-generation support is insufficient. The other replay failure stems from a `choice` expression in the certificate, which chooses an element from a set. **crush** does not currently support decoding this expression, since it would require synthesizing a proof of non-emptiness of the set from which the element is chosen.

In 11 VCs, certificate generation fails and core-guided reconstruction also fails to find a proof: 4 occur in Velvet and 7 in PLean. The remaining failures are due to exhausted heartbeat budgets and tactic failures, caused by insufficient lemmas for the built-in proof search procedures during replay.

Replay implementation effort. The built-in replay handler library covers assumption normalization, logic and equality, quantifiers, arithmetic, bitvectors, and strings through the extension interface (§5.2). The 55 manually defined handlers take 256 LoC, with another 797 LoC of supporting lemmas, for a total of 1,053 LoC (Table 4).

7 Related Work

ATP Integration in Proof Assistants. CoqHammer [10] translates Rocq goals through CIC_0 into untyped first-order logic. Although there is a soundness proof for CoqHammer's core encoding, its implementation uses approximations that can compromise soundness, including omitted type guards in lambda-lifting equations. It ensures soundness by reconstructing kernel-checked Rocq proofs

using ATP-selected premises. **crush** instead mechanizes the soundness of defunctionalization for its supported higher-order fragment.

SMTCoq [2] reifies Rocq goals into SMT formulas and validates solver certificates using a verified checker. Its modular architecture supports additional solvers and theories; it provides extensible preprocessing through Trakt [5]. It targets universal goals over supported SMT theories. In contrast, **crush** supports higher-order terms and existential quantifications, and natively provides extensible translation and proof replay. Its handlers construct kernel-checked Lean proofs without requiring separate soundness proofs for their implementations.

Automation Tactics in Lean. Several prior works attempted to close the gap between Lean and SMT solvers for better proof automation. lean-auto [28] uses a monomorphization pass to translate dependent type theory into higher-order logic, but its SMT backend supports a restricted first-order fragment without SMT proof reconstruction or custom encoding interfaces. lean-smt [21] uses lean-auto for monomorphization and provides extensible SMT translation, term decoding, and inference replay. Its translation similarly requires a supported first-order fragment. LeanHammer [36] combines premise selection, lean-auto's higher-order translation, Zipperposition [35] proof search, and Duper [9] reconstruction from selected premises. More recent versions also integrate lean-smt, hence inheriting its SMT translation and replay capabilities.

crush is also broadly related to tactics that are natively implemented in Lean. Grind [22] implements several SMT solving procedures in Lean to perform proof search; Duper [9] implements proof-producing superposition, similar to Isabelle's Metis [27]; Aesop [20] implements white-box proof search in Lean.

8 Conclusion and Future Work

We present **crush**, an extensible proof automation framework for Lean. It provides extensible translation interfaces for encoding Lean proof obligations into first-order SMT and leverages defunctionalization to support higher-order constructs. We have also provided a soundness proof of this translation procedure for the supported fragment of Lean. **crush** reconstructs proofs from solver certificates and supports proof search guided by unsat cores and replaying inference steps produced by proof-producing SMT solvers. On 4 benchmark suites with 754 verification conditions in Lean, **crush** solves 709 (94.0%) when trusting the SMT solver and 693 (91.9%) when proof reconstruction is required, outperforming all the baseline approaches.

Future work. We aim to support dependent function types and extend the soundness formalization beyond defunctionalization to Lean-to-SMT-LIB translation. For proof reconstruction, we plan to integrate tactics such as Duper [9], support CPC [29] certificates, and synthesize non-emptiness proofs for `choice` expressions in solver proof certificates. We also aim to improve the feedback to the user when the SMT solver reports SAT, for example, by distinguishing true and spurious counter-examples.

Data Availability

The implementation of **crush**, its mechanized proofs, benchmark data, and scripts for reproducing the experiments are publicly available at `https://github.com/AD1024/lean-crush/tree/tacas-ae`.

References

1. Appel, A.W.: Verified software toolchain: (invited talk). In: European Symposium on Programming. pp. 1–17. Springer (2011)
2. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of sat/smt solvers to coq through proof witnesses. In: Proceedings of the First International Conference on Certified Programs and Proofs. p. 135–150. CPP'11, Springer-Verlag, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_12, https://doi.org/10.1007/978-3-642-25379-9_12
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 the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, 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. 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
5. 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
6. Böhme, S., Weber, T.: Fast lcf-style proof reconstruction for z3. In: International Conference on Interactive Theorem Proving. pp. 179–194. Springer (2010)
7. Brooker, M., Desai, A.: Systems correctness practices at amazon web services. Commun. ACM **68**(6), 38–42 (Jun 2025). https://doi.org/10.1145/3729175, https://doi.org/10.1145/3729175
8. Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., Arvind: Kami: a platform for high-level parametric hardware specification and its modular verification. Proceedings of the ACM on Programming Languages **1**(ICFP), 1–30 (2017)
9. Clune, J., Qian, Y., Bentkamp, A., Avigad, J.: Duper: A proof-producing superposition theorem prover for dependent type theory. In: International Conference on Interactive Theorem Proving (2024), https://api.semanticscholar.org/CorpusID:272330518
10. Czajka, ź., Kaliszyk, C.: Hammer for coq: Automation for dependent type theory. J. Autom. Reason. **61**(1–4), 423–453 (Jun 2018). https://doi.org/10.1007/s10817-018-9458-4, https://doi.org/10.1007/s10817-018-9458-4

11. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
12. Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., Zufferey, D.: P: safe asynchronous event-driven programming. SIGPLAN Not. **48**(6), 321–332 (Jun 2013). https://doi.org/10.1145/2499370.2462184, https://doi.org/10.1145/2499370.2462184
13. Gladshtein, V., Kurin, V., Feng, Y., Kafle, D., Pîrlea, G., Zhao, Q., Sergey, I.: Velvet: A foundational multi-modal verifier for imperative programs in lean. In: Computer Aided Verification: 38th International Conference, CAV 2026, Lisbon, Portugal, July 26–29, 2026, Proceedings, Part II. p. 228–242. Springer-Verlag, Berlin, Heidelberg (2026). https://doi.org/10.1007/978-3-032-32526-6_11, https://doi.org/10.1007/978-3-032-32526-6_11
14. Gladshtein, V., Pîrlea, G., Zhao, Q., Kurin, V., Sergey, I.: Foundational multi-modal program verifiers. Proc. ACM Program. Lang. **10**(POPL) (Jan 2026). https://doi.org/10.1145/3776719, https://doi.org/10.1145/3776719
15. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: Ironfleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles. p. 1–17. SOSP '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2815400.2815428, https://doi.org/10.1145/2815400.2815428
16. Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodová, A., Taylor, C., Frolov, V., Reeber, E., et al.: Replacing testing with formal verification in intel coretm i7 processor execution engine validation. In: International Conference on Computer Aided Verification. pp. 414–429. Springer (2009)
17. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: formal verification of an os kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. p. 207–220. SOSP '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1629575.1629596, https://doi.org/10.1145/1629575.1629596
18. Lab, V.: Loom — github.com. https://github.com/verse-lab/loom/tree/master/CaseStudies/Cashmere, [Accessed 09-10-2026]
19. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: Compcert-a formally verified optimizing compiler. In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress (2016)
20. 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
21. 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. In: Computer Aided Verification: 37th International Conference, CAV 2025, Zagreb, Croatia, July 23–25, 2025, Proceedings, Part III. p. 197–212. Springer-Verlag, Berlin, Heidel-

berg (2025). https://doi.org/10.1007/978-3-031-98682-6_11, https://doi.org/10.1007/978-3-031-98682-6_11
22. Morrison, K., de Moura, L.: grind: An smt-inspired tactic for lean 4 (short paper – system description). In: Biere, A., Lutz, C., Negri, S. (eds.) Automated Reasoning. pp. 106–114. Springer Nature Switzerland, Cham (2026)
23. de Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings. p. 625–635. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-79876-5_37, https://doi.org/10.1007/978-3-030-79876-5_37
24. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer-Verlag, Berlin, Heidelberg (2002)
25. Norell, U.: Dependently typed programming in agda. In: Proceedings of the 4th International Workshop on Types in Language Design and Implementation. p. 1–2. TLDI '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1481861.1481862, https://doi.org/10.1145/1481861.1481862
26. P-Developers: Plean — github.com. https://github.com/p-org/P/tree/dev/p-lean/Src/PLean, [Accessed 09-10-2026]
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. pp. 232–245. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
28. Qian, Y., Clune, J., Barrett, C., Avigad, J.: Lean-auto: An interface between lean 4 and automated theorem provers. In: Computer Aided Verification: 37th International Conference, CAV 2025, Zagreb, Croatia, July 23–25, 2025, Proceedings, Part III. p. 175–196. Springer-Verlag, Berlin, Heidelberg (2025). https://doi.org/10.1007/978-3-031-98682-6_10, https://doi.org/10.1007/978-3-031-98682-6_10
29. Reynolds, A., Schurr, H.J., Barbosa, H., Israel, O., Jakpor, J.Z., Lachnitt, H., Mohamed, A., Niemetz, A., Preiner, M., Zohar, Y., Jones, R., Barrett, C., Tinelli, C.: The cooperating proof calculus: Comprehensive proofs for an smt solver. In: Darulova, E., Lin, A.W., Rümmer, P. (eds.) Computer Aided Verification. pp. 188–212. Springer Nature Switzerland, Cham (2026)
30. Russinoff, D.M.: A mechanically checked proof of ieee compliance of the floating point multiplication, division and square root algorithms of the amd-k7™ processor. LMS Journal of Computation and Mathematics **1**, 148–200 (1998)
31. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Automated Deduction – CADE 27: 27th International Conference on Automated Deduction, Natal, Brazil, August 27–30, 2019, Proceedings. p. 495–507. Springer-Verlag, Berlin, Heidelberg (2019). https://doi.org/10.1007/978-3-030-29436-6_29, https://doi.org/10.1007/978-3-030-29436-6_29
32. Schurr, H.J., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a generic smt proof format. arXiv preprint arXiv:2107.02354 (2021)
33. Sharma, U., Jung, R., Tassarotti, J., Kaashoek, F., Zeldovich, N.: Grove: a separation-logic library for verifying distributed systems. In: Proceedings of the 29th Symposium on Operating Systems Principles. p. 113–129. SOSP '23, ACM (Oct 2023). https://doi.org/10.1145/3600006.3613172, http://dx.doi.org/10.1145/3600006.3613172

34. Team, T.C.D.: The coq proof assistant (Sep 2024). https://doi.org/10.5281/zenodo.14542673, https://doi.org/10.5281/zenodo.14542673
35. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making Higher-Order Superposition Work. Journal of Automated Reasoning **66**(4), 541–564 (Nov 2022). https://doi.org/10.1007/s10817-021-09613-z, https://inria.hal.science/hal-03909997
36. Zhu, T., Clune, J., Avigad, J., Jiang, A.Q., Welleck, S.: Premise selection for a lean hammer (2026), https://arxiv.org/abs/2506.07477

Appendix

Table of Contents

A Details of Pre-processing passes 24
A.1 Fact Collection and Normalization 24
A.2 Lemma Monomorphization 24
A.3 Bounded Quantifier Instantiation 25
B Formalization of Defunctionalization 25
B.1 Languages and Translation 25
B.2 Model Extension 27
B.3 Proof of Preservation of Denotations 27
B.4 Proof of Validity of Generated Axioms 30
B.5 Proof of Soundness 32
B.6 Scope of Formalization 32
C Reconstruction Procedures 33
D Trusted SMT Comparison 34

A Details of Pre-processing passes

Given a Lean proof goal, **crush** first constructs a set of propositions and pre-processes them before translating (defunctionalizing) them.

A.1 Fact Collection and Normalization

crush collects the negated goal, local propositional hypotheses, and lemmas supplied explicitly or selected from the environment. It retains their origins so an unsat core can later be mapped back to the corresponding Lean facts.

crush further applies normalization on the collected propositions. If they use definitions that are marked with `crush_unfold`, **crush** will rewrite their applications using the definitions' equation lemmas and also expose these (quantified) equations as facts in future passes. We note that such rewriting is performed by tactics that produce equality proofs checked by Lean kernel. Moreover, **crush** will not expose equations of unused definitions, preventing introduction of quantifiers and instantiations that are unproductive.

Example. Suppose $\mathit{double}(x)$ is defined as $x+x$ and marked with `crush_unfold`. For a goal $0 \le \mathit{double}(x)$ under the hypothesis $0 \le x$, fact collection produces

$$0 \le x \qquad \text{and} \qquad \neg\big(0 \le \mathit{double}(x)\big).$$

Because the marked definition occurs in the negated goal, normalization rewrites the latter fact to $\neg(0 \le x + x)$. The recorded equality proof connects this normalized contradiction back to the original goal.

A.2 Lemma Monomorphization

Lean lemmas may be polymorphic, i.e., quantifying over types, whereas SMT symbols have concrete sorts. Translating the type variables of such a lemma into some uninterpreted sorts will make SMT solvers unable to use it for a concrete sort, e.g., Int. **crush** therefore collects the concrete data types in the proof obligation and specializes the type parameters of polymorphic lemmas at those types, maybe iteratively. Every instance is obtained by applying the original Lean proof. The numbers of instances and iterations are bounded by user-configurable options, which could reduce the completeness but still preserve soundness. We note that function types are handled separately by the defunctionalization procedure in §4.

Example. Consider a polymorphic function f and a hypothesis

$$h : \forall(\alpha : \mathsf{Type})\,(xs : \mathsf{List}\ \alpha), f\ \alpha\ xs = 0.$$

If the goal is $f\ \mathsf{Int}\ ys = 0$, where $ys : \mathsf{List\ Int}$, the occurrences of Int and $\mathsf{List\ Int}$ direct **crush** to apply h to Int. The resulting fact

$$h_{\mathsf{Int}} : \forall(xs : \mathsf{List\ Int}), f\ \mathsf{Int}\ xs = 0$$

uses the same concrete SMT symbols as the goal while retaining $h\ \mathsf{Int}$ in its Lean proof.

A.3 Bounded Quantifier Instantiation

Bounded quantifier instantiation supplies solver-side E-matching with a finite set of relevant ground instances. E-matching requires a suitable ground term before instantiating a quantified proposition, so it can miss a forward chain in which one instance produces the trigger for the next. Therefore, **crush** applies active bounded instantiation by matching quantified lemmas against ground terms and proposition occurrences already in the query, including newly generated instances. Note that every instance is constructed by applying its quantified Lean proof to the selected arguments.

crush takes a "two-shot" approach after quantifier instantiation. It first emits an SMT query without the quantified lemmas, avoiding unbounded solver-side instantiation. It closes the goal if this query is already UNSAT. If the query is SAT or unknown, **crush** retries with the complete set of propositions including quantified lemmas.

Example. Let the selected lemmas and goal be

$$\begin{aligned} &\mathit{step} : \forall x,\ R\ x\ (\mathit{next}\ x),\\ &\mathit{lift}\ \ : \forall x\, y,\ R\ x\ y \to P\ y,\\ &\mathit{goal} : \exists y,\ R\ a\ y \wedge P\ y. \end{aligned}$$

The ground term a first instantiates *step*, producing $R\ a\ (\mathit{next}\ a)$. That new occurrence fixes both arguments of *lift*, producing $P\ (\mathit{next}\ a)$ in the next round. The solver can then choose $\mathit{next}\ a$ as the existential witness. Both generated facts carry applications of the original lemmas as Lean proofs.

B Formalization of Defunctionalization

We formalize typed higher-order and first-order languages and a translation between them, and prove Theorem 1.

B.1 Languages and Translation

Fig. 8 defines both languages and their semantics. A higher-order signature assigns a type to each constant, and a context assigns types to local variables. The Lean representation is Intrinsically typed, i.e., indexing terms by their signature, context, and type, so only well-typed terms can be constructed. Formulas have type Bool; they are closed when their local context is empty.

First-order terms are likewise indexed by their context and result sort. Generated symbols are identified by the original constant, function type, or closure they represent, independently of concrete SMT names. Their argument and result sorts are part of their types. Valuations respect these type and sort assignments. In both languages, quantifiers range over the full interpretation of their binder types or sorts, and the valuation is extended with the bound value when evaluating the body.

Recursive translation. The following rules give the full definition of $\mathcal{F}$, including the cases summarized in Fig. 9. Throughout, $\tau \in \{\mathsf{Bool}, \beta\}$ is a non-function

sort and $\theta = \sigma_1 \to \cdots \to \sigma_n \to \tau$ with $n \geq 1$. Variables and constants of non-function sort, and Boolean literals, require no recursive translation:

$$\frac{x : \tau}{\mathcal{F}(x) = (x, \emptyset, \emptyset)} \qquad \frac{c : \tau}{\mathcal{F}(c) = (\widehat{c}, \emptyset, \emptyset)}$$

$$\frac{v \in \{\mathsf{true}, \mathsf{false}\}}{\mathcal{F}(v) = (v, \emptyset, \emptyset)}$$

Applications. The helper $\mathcal{A}_h(\bar{u}) = (t, C^h, E^h)$ applies a head $h : \theta$ to a complete list of first-order arguments $\bar{u}$, returning a term and the head's auxiliary axioms. Here c is a constant, f a bound variable, and ℓ a lambda; u_i has sort $\sigma_i^\flat$.

$$\frac{c : \theta}{\mathcal{A}_c(\bar{u}) = (\widehat{c}(\bar{u}), \emptyset, \emptyset)} \qquad \frac{f : \theta}{\mathcal{A}_f(\bar{u}) = (\mathsf{app}_\theta(f, \bar{u}), \emptyset, \emptyset)} \tag{1}$$

$$\frac{\ell : \theta \qquad \mathcal{F}(\ell) = (\ell^\flat, C_\ell, E_\ell)}{\mathcal{A}_\ell(\bar{u}) = (\mathsf{app}_\theta(\ell^\flat, \bar{u}), C_\ell, E_\ell)} \tag{2}$$

The application rule translates every argument before applying the head:

$$\frac{h : \theta \qquad \mathcal{F}(a_i) = (a_i^\flat, C_{a_i}, E_{a_i}) \quad (1 \leq i \leq n) \qquad \mathcal{A}_h(\bar{a}^\flat) = (t, C^h, E^h)}{\mathcal{F}(h\, a_1 \cdots a_n) = \big(t,\ C^h \cup \bigcup_i C_{a_i},\ E^h \cup \bigcup_i E_{a_i}\big)} \tag{3}$$

Equality and logical connectives. Equality at function sort additionally generates the extensionality axiom ext_θ:

$$\frac{u, v : \tau \qquad \mathcal{F}(u) = (u^\flat, C_u, E_u) \qquad \mathcal{F}(v) = (v^\flat, C_v, E_v)}{\mathcal{F}(u = v) = (u^\flat = v^\flat,\ C_u \cup C_v,\ E_u \cup E_v)}$$

$$\frac{u, v : \theta \qquad \mathcal{F}(u) = (u^\flat, C_u, E_u) \qquad \mathcal{F}(v) = (v^\flat, C_v, E_v)}{\mathcal{F}(u = v) = (u^\flat = v^\flat,\ C_u \cup C_v,\ E_u \cup E_v \cup \{\mathsf{ext}_\theta\})}$$

$$\mathsf{ext}_\theta = (\forall f, g : \mathsf{Fn}_\theta.\ (\forall \bar{x}.\ \mathsf{app}_\theta(f, \bar{x}) = \mathsf{app}_\theta(g, \bar{x})) \to f = g).$$

For logical connectives and quantifiers, let $\circ \in \{\wedge, \vee, \to, \leftrightarrow\}$ and $Q \in \{\forall, \exists\}$:

$$\frac{\mathcal{F}(p) = (p^\flat, C_p, E_p)}{\mathcal{F}(\neg p) = (\neg p^\flat, C_p, E_p)} \qquad \frac{\mathcal{F}(p) = (p^\flat, C_p, E_p)}{\mathcal{F}(Qx : \sigma.\, p) = (Qx : \sigma^\flat.\, p^\flat, C_p, E_p)}$$

$$\frac{\mathcal{F}(p) = (p^\flat, C_p, E_p) \qquad \mathcal{F}(q) = (q^\flat, C_q, E_q)}{\mathcal{F}(p \circ q) = (p^\flat \circ q^\flat, C_p \cup C_q, E_p \cup E_q)}$$

Opening function bodies. To cover partial applications and lambdas whose bodies still have function type, define a helper $\mathcal{B}(e, \bar{z}) = (r, C, E)$ that translates e applied to fresh parameters $\bar{z}$ until its result is non-function. The parameter

types are the complete argument list of the type of e; $\bar{z}$ is empty when e has non-function type. Substitution below is capture-avoiding.

$$\frac{e : \tau \qquad \mathcal{F}(e) = (r, C, E)}{\mathcal{B}(e, ()) = (r, C, E)} \qquad \frac{\mathcal{B}(b[z_0/x], \bar{z}) = (r, C, E)}{\mathcal{B}(\lambda x.\, b, (z_0, \bar{z})) = (r, C, E)} \tag{4}$$

For a non-lambda term $h\,\bar{a}$, the remaining parameters complete the application. The list $\bar{a} = (a_1, \ldots, a_m)$ may be empty for a variable or constant, and $\bar{z}$ contains the remaining $n - m$ parameters.

$$\frac{\begin{array}{c} h : \theta \qquad m < n \\ \mathcal{F}(a_i) = (a_i^\flat, C_{a_i}, E_{a_i}) \quad (1 \le i \le m) \qquad \mathcal{A}_h(\bar{a}^\flat, \bar{z}) = (r, C^h, E^h) \end{array}}{\mathcal{B}(h\,\bar{a}, \bar{z}) = \big(r,\ C^h \cup \bigcup_i C_{a_i},\ E^h \cup \bigcup_i E_{a_i}\big)} \tag{5}$$

Fresh parameters are passed as first-order variables, including when their types are function types.

Closure construction. For any function-valued $e : \theta$, let ℓ be its lambda form obtained by opening leading lambda binders and η-expanding over any remaining arguments. Let $\bar{y}$ be its free variables and $\bar{z}$ fresh parameters for all arguments of θ.

$$\frac{e : \theta \qquad \ell = \lambda \bar{x}.\ e\ \bar{x} \qquad \bar{y} = \mathsf{FV}(\ell) \qquad \mathcal{B}(e, \bar{z}) = (r, C, E)}{\begin{array}{c} \mathcal{F}(e) = (\mathsf{clos}_\ell(\bar{y}),\ C \cup \{K_\ell\},\ E) \\ K_\ell = \forall \bar{y}, \bar{z}.\ \mathsf{app}_\theta(\mathsf{clos}_\ell(\bar{y}), \bar{z}) = r. \end{array}} \tag{6}$$

This generalizes the lambda clause of Fig. 9 and defines the function-valued variable, constant, and partial-application cases. Captures are passed directly as bound variables, without η-expansion. Opening nested lambda binders together avoids intermediate closures.

B.2 Model Extension

For any higher-order model M, we construct its canonical first-order extension $M^\star$ by retaining the carriers of base types and interpreting Fn_θ by the function space interpreting θ. The interpretation of app_θ applies its function argument to the supplied arguments. The interpretation of $\widehat{c}$ is the uncurried form of $M(c)$. A closure symbol reconstructs the captured environment and evaluates the corresponding lambda. This defines one interpretation for all generated symbols, independent of the particular formula being translated (see Definition 4).

B.3 Proof of Preservation of Denotations

Proof (of Lemma 1). We fix a higher-order model M and its extension $M^\star$ from Definition 4. We first establish 3 valuation identities by structural induction. For a type-preserving renaming π, write $e[\pi]$ for the renamed term and $(\rho \circ \pi)(x) = \rho(\pi(x))$. Then

$$[\![e[\pi]]\!]_M(\rho) = [\![e]\!]_M(\rho \circ \pi). \tag{7}$$

Agreement on free variables gives the rule

$$\frac{\forall x \in \mathsf{FV}(e).\ \rho_1(x) = \rho_2(x)}{[\![e]\!]_M(\rho_1) = [\![e]\!]_M(\rho_2)}. \tag{8}$$

In the binder cases, the induction hypothesis applies after extending both valuations with the same value. These identities also hold for FO terms.

For fresh z, the lambda and application semantics give

$$\begin{aligned}[\![\lambda z.\, e\, z]\!]_M(\rho) &= (\mathsf{fun}\ v \Rightarrow [\![e]\!]_M(\rho)\, v) \\ &= [\![e]\!]_M(\rho).\end{aligned}$$

Repeated η-expansion therefore preserves the denotation of the lambda form ℓ used in Eq. (6).

We prove preservation for $\mathcal{F}$ simultaneously with the following properties of the two helpers, by induction on the translation derivations. Each property quantifies over every valuation ρ and $\bar{w}$:

$$\frac{\mathcal{A}_h(\bar{u}) = (t, C, E)}{[\![t]\!]_{M^\star}(\rho) = [\![h]\!]_M(\rho)\, [\![u_1]\!]_{M^\star}(\rho) \cdots [\![u_n]\!]_{M^\star}(\rho)}. \tag{9}$$

$$\frac{\mathcal{B}(e, \bar{z}) = (r, C, E)}{[\![r]\!]_{M^\star}(\rho[\bar{z} \mapsto \bar{w}]) = [\![e]\!]_M(\rho)\, w_1 \cdots w_n}. \tag{10}$$

In Eq. (10), $\bar{z}$ is fresh for e and $\bar{w}$ is any tuple of values of the corresponding sorts. For an empty tuple, application to $\bar{w}$ leaves the value unchanged.

Base Case: Variables, Constants, and Literals. For $x, c : \tau$ where τ is a non-function type, the base translation rules and Fig. 8 yield

$$\begin{aligned}[\![x^\flat]\!]_{M^\star}(\rho) &= [\![x]\!]_M(\rho) = \rho(x) \\ [\![c^\flat]\!]_{M^\star}(\rho) &= [\![c]\!]_M(\rho) = M(c)\end{aligned}$$

The Boolean-literal rule preserves True and False. For $v_i = [\![u_i]\!]_{M^\star}(\rho)$, their interpretations give

$$\begin{aligned}[\![\widehat{c}(\bar{u})]\!]_{M^\star}(\rho) &= M(c)\, v_1 \cdots v_n, \\ [\![\mathsf{app}_\theta(f, \bar{u})]\!]_{M^\star}(\rho) &= \rho(f)\, v_1 \cdots v_n.\end{aligned}$$

This completes the proof of Eq. (9) for constant and variable head (Eq. (1)).

Closure Case. The induction hypothesis is Eq. (10) for the recursive call $\mathcal{B}(e, \bar{z})$. Preservation of the closure term itself follows directly from its interpretation. In Eq. (6), let $\bar{v} = \rho(\bar{y})$. Since $\bar{y} = \mathsf{FV}(\ell)$, the captured valuation $\rho_{\ell,\bar{v}}$ agrees with ρ on every free variable of ℓ. Thus

$$\begin{aligned}[\![e^\flat]\!]_{M^\star}(\rho) &= M^\star(\mathsf{clos}_\ell)(\bar{v}) && \text{Eq. (6), Fig. 8} \\ &= [\![\ell]\!]_M(\rho_{\ell,\bar{v}}) && \text{Definition 4} \\ &= [\![\ell]\!]_M(\rho) && \text{Eq. (8)} \\ &= [\![e]\!]_M(\rho) && \eta\text{-expansion.}\end{aligned}$$

This covers lambdas and the η-expanded forms of function-valued variables, constants, and partial applications.

Lambda Case. The induction hypothesis for the recursive call $\mathcal{F}(\ell)$ is $[\![\ell^\flat]\!]_{M^\star}(\rho) = [\![\ell]\!]_M(\rho)$ for every ρ. Let $v_i = [\![u_i]\!]_{M^\star}(\rho)$. The rule in Eq. (2) and the interpretation of app_θ give

$$\begin{aligned}[\![\mathsf{app}_\theta(\ell^\flat, \bar{u})]\!]_{M^\star}(\rho) &= [\![\ell^\flat]\!]_{M^\star}(\rho)\, v_1 \cdots v_n \\ &= [\![\ell]\!]_M(\rho)\, v_1 \cdots v_n.\end{aligned}$$

This establishes Eq. (9) for a lambda head.

Application Case. The induction hypotheses are $[\![a_i^\flat]\!]_{M^\star}(\rho) = [\![a_i]\!]_M(\rho)$ for every argument a_i, and Eq. (9) for $\mathcal{A}_h(\bar{a}^\flat)$, all at every valuation ρ. Applying them to Eq. (3) gives

$$\begin{aligned}[\![(h\,\bar{a})^\flat]\!]_{M^\star}(\rho) &= [\![h]\!]_M(\rho)\, [\![a_1^\flat]\!]_{M^\star}(\rho) \cdots [\![a_n^\flat]\!]_{M^\star}(\rho) \\ &= [\![h]\!]_M(\rho)\, [\![a_1]\!]_M(\rho) \cdots [\![a_n]\!]_M(\rho) \\ &= [\![h\,\bar{a}]\!]_M(\rho).\end{aligned}$$

Equality Case. The induction hypotheses, for every ρ, are

$$\begin{aligned}[\![u^\flat]\!]_{M^\star}(\rho) &= [\![u]\!]_M(\rho), \\ [\![v^\flat]\!]_{M^\star}(\rho) &= [\![v]\!]_M(\rho).\end{aligned}$$

The equality translation rule and Fig. 8 give

$$\begin{aligned}[\![u^\flat = v^\flat]\!]_{M^\star}(\rho) &= ([\![u^\flat]\!]_{M^\star}(\rho) = [\![v^\flat]\!]_{M^\star}(\rho)) \\ &= ([\![u]\!]_M(\rho) = [\![v]\!]_M(\rho)) \\ &= [\![u = v]\!]_M(\rho).\end{aligned}$$

This applies to equality at both function and non-function sorts.

Negation Case. The induction hypothesis is $[\![p^\flat]\!]_{M^\star}(\rho) = [\![p]\!]_M(\rho)$ for every ρ. By the negation translation rule and its semantics,

$$\begin{aligned}[\![\neg p^\flat]\!]_{M^\star}(\rho) &= \neg [\![p^\flat]\!]_{M^\star}(\rho) \\ &= \neg [\![p]\!]_M(\rho) \\ &= [\![\neg p]\!]_M(\rho).\end{aligned}$$

Binary Connective Case. The induction hypotheses are preservation for p and q at every ρ. For $\circ \in \{\wedge, \vee, \to, \leftrightarrow\}$,

$$\begin{aligned}[\![p^\flat \circ q^\flat]\!]_{M^\star}(\rho) &= [\![p^\flat]\!]_{M^\star}(\rho) \circ [\![q^\flat]\!]_{M^\star}(\rho) \\ &= [\![p]\!]_M(\rho) \circ [\![q]\!]_M(\rho) \\ &= [\![p \circ q]\!]_M(\rho).\end{aligned}$$

Quantifier Case. The induction hypothesis is preservation for p at every extended valuation:

$$\forall \rho, v. \quad [\![p^\flat]\!]_{M^\star}(\rho[x \mapsto v]) = [\![p]\!]_M(\rho[x \mapsto v]).$$

For $Q \in \{\forall, \exists\}$, corresponding sorts have the same carriers (i.e., $[\![\sigma^\flat]\!]_{M^\star} = [\![\sigma]\!]_M$), so the quantifier translation rule gives

$$\begin{aligned}[\![Qx : \sigma^\flat.\, p^\flat]\!]_{M^\star}(\rho) &= Qv : [\![\sigma^\flat]\!]_{M^\star}.\ [\![p^\flat]\!]_{M^\star}(\rho[x \mapsto v]) \\ &= Qv : [\![\sigma]\!]_M.\ [\![p]\!]_M(\rho[x \mapsto v]) \\ &= [\![Qx : \sigma.\, p]\!]_M(\rho).\end{aligned}$$

Non-function Body Case. The induction hypothesis for $\mathcal{F}(e) = (r, C, E)$ is

$$\forall \rho. \quad [\![r]\!]_{M^\star}(\rho) = [\![e]\!]_M(\rho).$$

The non-function rule in Eq. (4) has no fresh parameters, so this is exactly Eq. (10).

Lambda Body Case. The induction hypothesis for $\mathcal{B}(b[z_0/x], \bar{z}) = (r, C, E)$ is

$$\forall \rho', \bar{w}. \quad [\![r]\!]_{M^\star}(\rho'[\bar{z} \mapsto \bar{w}]) = [\![b[z_0/x]]\!]_M(\rho')\, \bar{w}.$$

Here, application to a tuple abbreviates successive application. To prove Eq. (10) for $\mathcal{B}(\lambda x.\, b, (z_0, \bar{z})) = (r, C, E)$, we need to prove

$$\forall \rho, w_0, \bar{w}. \quad [\![r]\!]_{M^\star}(\rho[z_0 \mapsto w_0, \bar{z} \mapsto \bar{w}]) = [\![\lambda x.b]\!]_M(\rho)\, w_0\, \bar{w}.$$

Let $\rho' = \rho[z_0 \mapsto w_0]$. The lambda rule in Eq. (4), Eq. (7) give

$$\begin{aligned}[\![r]\!]_{M^\star}(\rho[z_0 \mapsto w_0, \bar{z} \mapsto \bar{w}]) &= [\![r]\!]_{M^\star}(\rho'[\bar{z} \mapsto \bar{w}]) \\ &= [\![b[z_0/x]]\!]_M(\rho')\, \bar{w} \\ &= [\![b[z_0/x]]\!]_M(\rho[z_0 \mapsto w_0])\, \bar{w} \\ &= [\![b]\!]_M(\rho[x \mapsto w_0])\, \bar{w} \\ &= [\![\lambda x.\, b]\!]_M(\rho)\, w_0\, \bar{w}.\end{aligned}$$

Partial Application Body Case. The induction hypotheses are preservation for each supplied argument a_i and Eq. (9) for $\mathcal{A}_h(\bar{a}^\flat, \bar{z})$, at every valuation. For Eq. (5), let $\rho' = \rho[\bar{z} \mapsto \bar{w}]$. These hypotheses and freshness of $\bar{z}$ give

$$\begin{aligned}[\![r]\!]_{M^\star}(\rho') &= [\![h]\!]_M(\rho')\, [\![a_1^\flat]\!]_{M^\star}(\rho') \cdots [\![a_m^\flat]\!]_{M^\star}(\rho')\, \bar{w} \\ &= [\![h\, \bar{a}]\!]_M(\rho)\, \bar{w}.\end{aligned}$$

This establishes Eq. (10) and completes the recursive cases. □

B.4 Proof of Validity of Generated Axioms

Proof (of Lemma 2). We fix M and proceed by induction on the translation derivations: $M^\star \models C \cup E$ whenever $\mathcal{F}$, $\mathcal{A}$, or $\mathcal{B}$ returns auxiliary sets C, E. We use the following consequence of Definition 1 to combine these sets:

$$\frac{M^\star \models S_1 \quad \cdots \quad M^\star \models S_k}{M^\star \models \bigcup_{i=1}^{k} S_i}.$$

Base Case: Variables, Constants, and Literals. The base rules for $\mathcal{F}$, and the constant and variable head rules for $\mathcal{A}$, generate $C = E = \varnothing$ (i.e., $\top$). Thus $M^\star \models C \cup E$ holds vacuously.

Closure Case. The induction hypothesis for $\mathcal{B}(e, \bar{z}) = (r, C, E)$ is $M^\star \models C \cup E$. It remains to establish the newly generated closure equation K_ℓ. For Eq. (6), take any valuation ρ and let $\bar{v} = \rho(\bar{y})$ and $\bar{w} = \rho(\bar{z})$. Then

$$\begin{aligned}\llbracket \mathsf{app}_\theta(\mathsf{clos}_\ell(\bar{y}), \bar{z}) \rrbracket_{M^\star}(\rho) &= \llbracket \ell \rrbracket_M(\rho_{\ell,\bar{v}})\, \bar{w} \\ &= \llbracket e \rrbracket_M(\rho)\, \bar{w} \\ &= \llbracket r \rrbracket_{M^\star}(\rho).\end{aligned}$$

The first equality unfolds Definition 4; the second uses Eq. (8) and η-expansion; the third uses Eq. (10). By equality and universal quantification in Fig. 8,

$$\frac{\forall \rho.\ \llbracket \mathsf{app}_\theta(\mathsf{clos}_\ell(\bar{y}), \bar{z}) \rrbracket_{M^\star}(\rho) = \llbracket r \rrbracket_{M^\star}(\rho)}{M^\star \models \{K_\ell\}}.$$

Combining this with the induction hypothesis gives $M^\star \models C \cup \{K_\ell\} \cup E$, as required by the closure rule.

Function Equality Case. The induction hypotheses are $M^\star \models C_u \cup E_u$ and $M^\star \models C_v \cup E_v$. It remains to establish the newly generated axiom ext_θ. Let $F, G \in \llbracket \mathsf{Fn}_\theta \rrbracket_{M^\star}$ satisfy the premise of ext_θ. Definition 4 interprets these values as functions in $\llbracket \theta \rrbracket_M$ and app_θ as their application. Then,

$$\frac{\forall w_1, \ldots, w_n.\ F\, w_1 \cdots w_n = G\, w_1 \cdots w_n}{F = G} \quad \text{(function extensionality)}.$$

Since F and G are arbitrary, the implication and quantifier semantics give $M^\star \models \{\mathsf{ext}_\theta\}$. Together with the induction hypotheses, this yields

$$M^\star \models C_u \cup C_v \cup E_u \cup E_v \cup \{\mathsf{ext}_\theta\}.$$

Application and Partial Application Body Cases. The induction hypotheses are $M^\star \models C^h \cup E^h$ for the head and $M^\star \models C_{a_i} \cup E_{a_i}$ for every supplied argument. Taking their union gives

$$M^\star \models \big(C^h \cup \textstyle\bigcup_i C_{a_i}\big) \cup \big(E^h \cup \textstyle\bigcup_i E_{a_i}\big),$$

which is the output of Eqs. (3) and (5).

Non-function Equality and Binary Connective Cases. The induction hypotheses give validity of the auxiliary sets for each operand. These rules return their union, whose validity follows from the union rule above.

Negation and Quantifier Cases. The induction hypothesis is $M^\star \models C_p \cup E_p$ for the body p. Both rules retain these same sets. The generated axioms are already closed, so their validity is independent of the quantifier's bound value.

Lambda and Non-function Body Cases. The induction hypothesis is validity of the sets returned by $\mathcal{F}(\ell)$ or $\mathcal{F}(e)$, respectively. The corresponding rules for $\mathcal{A}$ and $\mathcal{B}$ return those sets unchanged.

Lambda Body Case. The induction hypothesis gives $M^\star \models C \cup E$ for the sets returned by $\mathcal{B}(b[z/x], \bar{z})$. The lambda rule in Eq. (4) retains these sets; renaming preserves validity by Eq. (7). This completes all cases. □

B.5 Proof of Soundness

Proof (of Theorem 1). We suppose $\mathcal{D}(T)$ is unsatisfiable and, for contradiction, assume a higher-order model M satisfying T. We construct $M^\star$ as in Definition 4. By satisfaction and Lemma 1, for each $\varphi \in T$ we have:

$$\frac{M \models T \qquad \varphi \in T \qquad [\![\varphi^\flat]\!]_{M^\star}(\varnothing) = [\![\varphi]\!]_M(\varnothing)}{M^\star \models \{\varphi^\flat\}}.$$

Lemma 2 supplies the auxiliary formulas. The same $M^\star$ interprets the symbols of every translated formula, so Definition 3 gives

$$\frac{\forall \varphi \in T.\ M^\star \models \{\varphi^\flat\} \qquad \forall \varphi \in T.\ M^\star \models C_\varphi \cup E_\varphi}{M^\star \models \mathcal{D}(T)}.$$

This contradicts the assumed unsatisfiability:

$$\frac{M^\star \models \mathcal{D}(T) \qquad \neg \exists N.\ N \models \mathcal{D}(T)}{\bot}.$$

Hence no M satisfies T, and T is unsatisfiable. □

B.6 Scope of Formalization

The language is monomorphic and nondependent. It contains no datatype constructors, pattern matching, or built-in arithmetic and bitvector operations. Such types and operations can be treated abstractly as base types and constants, but their concrete SMT encodings are not justified by this proof. Each base type has its own carrier, so the formalization also excludes embeddings into a larger sort and the guarded closure, application-result, and extensionality axioms used for those embeddings. Solver-native higher-order translation is not modeled.

Core-guided reconstruction

```
procedure Core(G, F, U)
  Γ ← {F(i) : A_i | i ∈ U,
      F(i) is defined}
  p ← Search(Γ, G, b)
  require Γ ⊢_kernel p : G
  return p

procedure Search(Γ, A, k)
  for t ∈ 𝒯 do
    try return t(Γ ⊢ A)
  if k = 0 then fail
  for r ∈ ℛ of the form
    r : ∀x̄. A_1 → ⋯ → A_n → B
  and σ with σ(B) ≡ A do
    try
      for i = 1, …, n do
        p_i ← Search(Γ, σ(A_i), k − 1)
      return r σ(x̄) p_1 ⋯ p_n
  fail
```

Proof replay

```
procedure Replay(G, F, Π)
  assume h : ¬G
  F′ ← F[i_G ↦ h]; P ← ∅
  for input (i, C) in Π do
    Γ_i ← {F′(i)} if defined,
      ∅ otherwise
    try p ← t(Γ_i ⊢ δ(C)),
      t ∈ 𝒯, until success
    require p : δ(C); P[i] ← p
  for step (i, r, C, j̄, ā) in Π do
    A ← δ(C); p̄ ← ⟨P[j] | j ∈ j̄⟩
    for handler η ∈ ℋ_r do
      try
        σ ← match(pattern_η, ā)
        p ← η(σ, p̄; A)
        require p : A
        P[i] ← p; break
    if P[i] undefined then fail
  require P[□] : False
  q ← byContradiction
    (fun h ⇒ P[□])
  require q : G by kernel checking
  return q
```

Fig. 13: Core-guided reconstruction (left) and proof replay (right).

There are two gaps remaining in the formalization. First, no mechanized result connects a Lean expression to a formula of this higher-order language. Second, first-order models interpret the entire collection of structurally identified symbols, whereas an SMT script declares a finite set of named symbols. Connecting these representations requires showing that all symbols occurring in the generated formulas are declared and that distinct structural symbols receive distinct names. We aim to complete these formalizations in future work.

C Reconstruction Procedures

Fig. 13 sketches reconstruction of a Lean goal G. $F(i) : A_i$ is the retained Lean proof for assertion i, U is the UNSAT core, and Π is the solver proof. $\mathcal{R}$ contains registered reconstruction lemmas, $\mathcal{T}$ contains finishing tactics, and b bounds backward search. A substitution σ instantiates lemma parameters or

binds handler captures. In the proof search procedure, **crush** first tries the supplied finisher tactics (such as `omega`), given the premises in the UNSAT core. If none of them can close the goal, **crush** then tries to match the current goal against the registered reconstruction lemma. If a lemma's conclusion matches the goal, it tries to `apply` the lemma and recursively search the proof for the premises of the lemma.

In replay of solver inference steps, δ decodes SMT clauses using term handlers; $\mathcal{H}_r$ includes handlers for rule r. A step records its identifier (i), rule (r), clause (C), premise identifiers $(\overline{j})$, and arguments $(\overline{a})$. i_G is the negated goal. We use $P[\square]$ to denote the proof of the final empty clause. For each such step, **crush** uses $\delta(C)$ to obtain the corresponding Lean obligation A. Then, it tries to match the pattern against the arguments. For a matching handler η, **crush** uses η to construct a proof term p and, together with the previously constructed proof terms $\overline{p}$, checks if the reconstructed proof term has type A. If any handler successfully closed the Lean obligation, **crush** records the constructed proof term and moves to the next step. If all handlers failed, **crush** will report the failure.

D Trusted SMT Comparison

Table 5 reports the head-to-head comparison when **crush** trusts SMT results. It uses the same VCs and baselines as Table 2. Runtime means and standard deviations are computed separately for each pair over the VCs both tools discharge.

Table 5: Head-to-head comparison with **crush**, trusting SMT verdicts. Times are mean ± standard deviation across VCs discharged by both tools, after averaging repetitions per VC.

		# VCs Discharged			Time on Both (ms)	
Benchmark	**Baseline**	**crush only**	**Baseline only**	**Both**	**crush**	**Baseline**
Curated (20 VCs)	`auto`	2	0	17	$234.8_{\pm 108.3}$	$322.2_{\pm 20.1}$
	Duper	8	1	11	$234.1_{\pm 114.0}$	$98.3_{\pm 24.1}$
	lean-smt	7	0	12	$239.4_{\pm 106.9}$	$106.8_{\pm 31.5}$
	`grind`	3	0	16	$234.6_{\pm 112.8}$	$5.5_{\pm 4.8}$
Cashmere (38 VCs)	`auto`	20	0	18	$227.2_{\pm 381.8}$	$294.0_{\pm 36.0}$
	Duper	17	0	21	$247.0_{\pm 400.5}$	$109.3_{\pm 102.6}$
	lean-smt	21	0	17	$151.4_{\pm 341.1}$	$135.6_{\pm 44.2}$
	`grind`	19	0	19	$215.4_{\pm 374.6}$	$9.8_{\pm 2.7}$
Velvet (504 VCs)	`auto`	73	10	405	$398.3_{\pm 490.2}$	$378.6_{\pm 225.7}$
	Duper	190	3	288	$232.0_{\pm 345.8}$	$220.3_{\pm 517.4}$
	lean-smt	185	9	293	$377.8_{\pm 480.7}$	$145.5_{\pm 112.5}$
	`grind`	43	9	435	$349.5_{\pm 371.7}$	$26.1_{\pm 26.8}$
PLean (192 VCs)	`auto`	2	0	172	$3583.1_{\pm 11750.4}$	$3621.0_{\pm 5972.5}$
	Duper	85	0	89	$609.6_{\pm 268.2}$	$2755.4_{\pm 8169.2}$
	lean-smt	74	0	100	$1521.8_{\pm 8680.9}$	$1298.7_{\pm 3712.7}$
	`grind`	12	0	162	$3528.6_{\pm 12102.2}$	$1206.5_{\pm 2327.6}$