Symbolic Proof Optimisation for Lean

Jannis Limperg
limperg@amazon.com
Amazon Web Services

Leaning In, March 2026, Berlin

https://github.com/JLimperg/talk-2026-03-leaning-in-proof-optimisation

Context

Objective

Fix obvious deficiencies in Lean proofs:

- Unused `have` tactic
- Tactic never called
- Manual proof can be replaced by `grind`/`linarith`/`aesop`
- Non-terminal `aesop` or non-squeezed `simp`
- ...

Context

Objective

Fix obvious deficiencies in Lean proofs:

- Unused `have` tactic
- Tactic never called
- Manual proof can be replaced by `grind`/`linarith`/`aesop`
- Non-terminal `aesop` or non-squeezed `simp`
- ...

Motivation

- Improve AI-generated proofs for training and human consumption
- (Eventually) improve sloppy human proofs and auto-apply linter suggestions

An LLM-Generated Proof

```
theorem cos_range_value (a : ℝ) (x : ℝ)
    (h_cos : cos x = (2 * a - 3) / (4 - a))
    (h_quad : -1 < cos x ∧ cos x < 0) :
    -1 < a ∧ a < 3 / 2 := by
  have h_denom_ne_zero : 4 - a ≠ 0 := by
    by_contra h
    have h₁ : 4 - a = 0 := by
      linarith
    rw [h₁] at h_cos
    norm_num at h_cos
    have h₂ : cos x = 0 := by
      linarith
    linarith [h_quad.2]
  have h_denom_pos : 4 - a > 0 := by

  ... 104 lines later ...

  exact ⟨h_neg_one_lt_a, h_a_lt_three_half⟩
```

Produced by DeepSeek-Prover-V2.

Symbolic Proof Optimisation

- Similar to an optimising compiler.
- Applies various optimisation passes that simplify the proof.
- Implemented in an internal (hopefully eventually open source) Lean library.

[1] Mathlib/Tactic/TacticAnalysis.lean

Symbolic Proof Optimisation

- Similar to an optimising compiler.
- Applies various optimisation passes that simplify the proof.
- Implemented in an internal (hopefully eventually open source) Lean library.

Mathlib's tactic analysis framework[1] is conceptually similar, but only generates optimisation suggestions without applying them.

[1] Mathlib/Tactic/TacticAnalysis.lean

Why Not LLM?

- Use specially trained LLM-based optimisers[2], or just ask Claude.
- Tradeoff: LLMs are more general, less development effort, more costly (time and money).
- Combined approach may be optimal: symbolic for obvious defects, LLMs for more interesting optimisations.
- Can integrate LLMs (or SLMs) as optimisation passes in a symbolic framework.

[2] Alex Gu et al. ProofOptimizer: Training Language Models to Simplify Proofs without Human Demonstrations. 2025. arXiv: 2510.15700 [cs.LG]. URL: https://arxiv.org/abs/2510.15700.

Redundancy 1: Unused Facts

```
have h₆ : a > 3 / 2 := by
  ...

... h₆ never used ...
```

Redundancy 2: Restating Known Facts

```
have h_denom_ne_zero : 4 - a ≠ 0 := by
  ...

have h₂ : 4 - a ≠ 0 := h_denom_ne_zero
```

Redundancy 3: Restating the Goal

```
have h₅₂ : (2 * a - 3) / (4 - a) ≥ 0 := by
  have h₅₃ : 4 - a < 0 := h₃
  have h₅₄ : (2 * a - 3) / (4 - a) ≥ 0 := by
    have h₅₅ : 2 * a - 3 ≤ 0 := h₅₁
    have h₅₆ : 4 - a < 0 := h₃
    have h₅₇ : (2 * a - 3) / (4 - a) ≥ 0 := by
      ...
    exact h₅₇
  exact h₅₄
```

Redundancy 4: Unused Tactics

```
field_simp [h₅₁, sub_ne_zero.mpr h₅₁] <;>
  ring_nf <;>
  field_simp [h₅₁, sub_ne_zero.mpr h₅₁] <;>
  nlinarith
```

Results

- Reduces our example proof from 115 lines to 60.
- Reduces code size of one of our SFT datasets (also produced by DeepSeek-Prover-V2[3]) by 60%.

Caveats

- Optimisation passes are generally incomplete.
- Optimisation may hurt when training an LLM to find Lean proofs in one shot.

[3] Z. Z. Ren et al. DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. 2025. arXiv: 2504.21801 [cs.CL]. URL: https://arxiv.org/abs/2504.21801.

Architecture

We distinguish three kinds of optimisation passes:

- Syntactic: purely syntactic, guaranteed semantics-preserving.
- Semantic: need elaboration information, e.g. goals before/after tactics.
  - Unchecked: guaranteed semantics-preserving.
  - Checked: generally not semantics-preserving; need to check individual rewrites.

Syntactic Passes

Examples

```
simp; skip      →   simp
solve | fail    →   fail
simp at h h     →   simp at h
```

Syntactic Passes

Examples

```
simp; skip      →   simp
solve | fail    →   fail
simp at h h     →   simp at h
```

```
theorem foo (n : Nat) : (m : Nat) → P n m := by
  intro m
  ...

→

theorem foo (n m : Nat) : P n m := by
  ...
```

Syntactic Passes

- Purpose: clean up after more interesting optimisations, e.g. after replacing an unused tactic with `skip`.
- Very cheap (compared to semantic passes), so we apply them before/after every semantic pass.
- We make them even cheaper:
  - Combine all optimisations into a single post-traversal with a fixpoint loop at each syntax node.
  - Index individual optimisations by their head node kind.

Custom Syntactic Sugar for Syntactic Passes

We define a custom elaborator for syntactic optimisations.

```
def solve := ls_analysis_fn%
  | `(tactic| solve $[| $tss]*) => do
    let tss' := tss.filter fun ts =>
      ! (isSkipTacticSeq ts || isFailTacticSeq ts)
    if tss'.size == tss.size then
      return none
    if tss'.isEmpty then
      `(tactic| fail)
    else
      `(tactic| solve $[| $tss']*)
```

Semantic Passes

Rely on the info trees produced during elaboration. These contain information about expression types, goals before/after each tactic, identifier shadowing, etc.

Semantic Passes

Rely on the info trees produced during elaboration. These contain information about expression types, goals before/after each tactic, identifier shadowing, etc.

E.g. to remove `have h : A := ...`, we must know that h was not used during the elaboration of the following tactics.

Info Tree Constraints

Info tree nodes are identified by syntax source position

Hence every rewrite invalidates the info trees.

Info Tree Constraints

Info tree nodes are identified by syntax source position

Hence every rewrite invalidates the info trees.

Syntax lookup is linear

Probably not a problem in practice, but we prematurely optimise by lazily building indices for the information we need.

Info Tree Constraints

Info tree nodes are identified by syntax source position

Hence every rewrite invalidates the info trees.

Syntax lookup is linear

Probably not a problem in practice, but we prematurely optimise by lazily building indices for the information we need.

Info tree information is sometimes incomplete

E.g. the connection between names in the syntax and fvars is sometimes lost.

Info Tree Constraints

Info tree nodes are identified by syntax source position

Hence every rewrite invalidates the info trees.

Syntax lookup is linear

Probably not a problem in practice, but we prematurely optimise by lazily building indices for the information we need.

Info tree information is sometimes incomplete

E.g. the connection between names in the syntax and fvars is sometimes lost.

Info trees are intended for use by the language server, so it’s expected that they do not perfectly support our use case.

The Performance Challenge

Naively, we can re-elaborate the command after each rewrite, both to check the rewrite and to update the info trees.

Problem

Prohibitively expensive: elaborating a proof can take seconds.

Partial solution: batching

Perform multiple rewrites and only then elaborate the changed command.

- Only works for non-checked passes.
- Only works if the rewrites are guaranteed not to interfere with each other.

The Performance Challenge

Naively, we can re-elaborate the command after each rewrite, both to check the rewrite and to update the info trees.

Problem

Prohibitively expensive: elaborating a proof can take seconds.

Partial solution: batching

Perform multiple rewrites and only then elaborate the changed command.

- Only works for non-checked passes.
- Only works if the rewrites are guaranteed not to interfere with each other.

Partial solution (future work): compositional optimisation

We ought to make use of the typical decomposition of Lean proofs into focused subproofs. However, there are subtleties, e.g. a focused subproof can have non-local effects if its initial goal contains a metavariable.

The Extensibility Challenge

Problem

Almost any invariant we rely on can be violated by custom tactics.

Solution (future work)

Define extension hooks for all passes that instruct the pass to skip certain tactics or process them differently. (Many linters have similar hooks.)

The Extensibility Challenge

Problem

Almost any invariant we rely on can be violated by custom tactics.

Solution (future work)

Define extension hooks for all passes that instruct the pass to skip certain tactics or process them differently. (Many linters have similar hooks.)

Solution (for now)

Ignore the problem and accept a certain failure rate (< 5% on our SFT dataset).

A Final Caveat

The pretty-printer somewhat regularly produces syntax that does not elaborate correctly, even with the `pp.analyze` option. This is a known (and very hard) problem that will hopefully be solved at some point.

Thanks!

Slides