PREMISE SELECTION FOR A LEAN HAMMER

Anonymous authors
Paper under double-blind review

ABSTRACT

Neural methods are transforming automated reasoning for proof assistants, yet integrating these advances into practical verification workflows remains challenging. A *hammer* is a tool that integrates premise selection, translation to external automatic theorem provers, and proof reconstruction into one overarching tool to automate tedious reasoning steps. We present LEANPREMISE, a novel neural premise selection system, and we combine it with existing translation and proof reconstruction components to create LEANHAMMER, the first end-to-end domain general hammer for the Lean proof assistant. Unlike existing Lean premise selectors, LEANPREMISE is specifically trained for use with a hammer in dependent type theory. It also dynamically adapts to user-specific contexts, enabling it to effectively recommend premises from libraries outside LEANPREMISE's training data as well as lemmas defined by the user locally. With comprehensive evaluations, we show that LEANPREMISE enables LEANHAMMER to solve 21% more goals than existing premise selectors and generalizes well to diverse domains. Our work helps bridge the gap between neural retrieval and symbolic reasoning, making formal verification more accessible to researchers and practitioners.

1 INTRODUCTION

Interactive proof assistants have long been used to verify the correctness of hardware, software, network protocols, cryptographic protocols, and other computational artifacts. Buoyed by successes like the Liquid Tensor Experiment (Lean Community, 2022) and the formalization of the Sphere Eversion Theorem (van Doorn et al., 2023), mathematicians are increasingly using the technology to verify mathematical theorems (Tao, 2023) and build substantial mathematical libraries (The Mathlib Community, 2020).

When working with a proof assistant, a user describes a proof in an idealized proof language, which is a programming language that provides sufficient detail for the computer to construct a precise formal derivation in the proof assistant's underlying axiomatic system. One of the challenges to formalization is the requirement to spell out what seem like straightforward inferences in painful detail. This problem is exacerbated by the fact that at the most basic level of interaction, users are required to name the required premises (i.e., definitions and lemmas) explicitly to justify an inference step, from a library of hundreds of thousands of previously derived facts.

A *hammer* (Meng et al., 2006; Paulson & Blanchette, 2012; Blanchette et al., 2016) is a tool designed to ease the pain of formalization by filling in small inferences automatically. Typically, a hammer has three components: given a goal to prove, one first selects a moderate number of premises from the library, project files, current file, and hypotheses that, one hopes, are sufficient to prove the goal. This is known as *premise selection*. Then one translates the premises and the goal into the language of powerful external automated theorem provers like Vampire (Kovács & Voronkov, 2013), E (Schulz et al., 2019), and Zipperposition (Vukmirović et al., 2022), or SMT solvers like Z3 (de Moura & Bjørner, 2008) and cvc5 (Barbosa et al., 2022). Finally, if the external prover succeeds in proving the goal, it reports back the specific premises used, from which a formal proof in the proof assistant is reconstructed.

In this paper, we present LEANPREMISE, a new premise selection tool for the Lean proof assistant (de Moura & Ullrich, 2021). We combine it with the DTT-to-HOL (dependent type theory to higher-order logic) translation tool, Lean-auto (Qian et al., 2025), and internal proof-producing tactics, Duper (Clune et al., 2024) and Aesop (Limperg & From, 2023), resulting in LEANHAMMER, the first end-to-end domain general hammer for Lean. Through comprehensive evaluations, we show that LEANHAMMER can hit nails.

Our work, which extends methods of premise selection used by Magnushammer (Mikuła et al., 2024) and LeanDojo (Yang et al., 2023), is therefore an auspicious combination of neural premise selection methods with symbolic proof search. For the first time, we specifically design contrastive learning methods for the first end-to-end domain general hammer in Lean. We explain the design choices to make LEANPREMISE performant for LEANHAMMER, including new *hammer-aware data extraction* techniques. An important feature of LEANPREMISE is that it dynamically augments the library of facts with locally defined facts from the user's project, which is essential in practice.

Our core contributions are as follows:

- We develop LEANPREMISE, a premise selection tool for a hammer in dependent type theory.
- We combine LEANPREMISE with Aesop, Lean-auto, and Duper to make LEANHAMMER, the first domain general hammer in Lean.
- We provide an accessible user-facing tactic interface that can dynamically process new premises in the environment.
- We conduct comprehensive evaluations of LEANHAMMER's performance on Mathlib and its ability to generalize to `miniCTX`-v2 (Hu et al., 2025). Through these evaluations, we show that LEANHAMMER solves 21% more goals with LEANPREMISE than with existing premise selectors and that LEANPREMISE enables LEANHAMMER to effectively use libraries and premises it hasn't seen before.

Note that premise selection can be used in other ways, for example, for calling various types of internal automation directly, for presenting suggestions to a user engaged in manual proof, or for use in a neural or neurosymbolic search. Although our focus here has been on a hammer, we expect that many of the methods we develop carry over to other settings.

2 RELATED WORK

2.1 HAMMERS IN INTERACTIVE PROOF ASSISTANTS

As explained in the introduction, hammers support interactive proving by completing small inferences, called *goals*. The first and still most successful hammer in use today is Isabelle's Sledgehammer, developed initially by Meng et al. (2006) and further developed by Paulson & Blanchette (2012); Blanchette et al. (2013), and many others. Since then hammers have been developed for HOL (Kaliszyk & Urban, 2015a), Mizar (Kaliszyk & Urban, 2015b), Rocq (Czajka & Kaliszyk, 2018), and Metamath (Carneiro et al., 2023), among others. Of these, only Rocq is based on dependent type theory. Despite Lean's popularity, no hammer has been developed for Lean.

2.2 NEURAL THEOREM PROVING

Numerous neural-network-based tools have been developed to prove theorems. A straightforward approach of using neural models is to let them generate steps in proofs, notable examples of which include GPT-*f* (Polu et al., 2023), HTPS (Lample et al., 2022), ReProver (Yang et al., 2023), DeepSeek-Prover (Xin et al., 2024a;b) for Lean, LISA (Jiang et al., 2021) and Thor (Jiang et al., 2022) for Isabelle, and PALM (Lu et al., 2024), Cobblestone (Kasibatla et al., 2024), and Graph2Tac (Blaauwbroek et al., 2024) for Coq/Rocq. Another line of work uses neural models to generate entire proofs or proof sketches (Jiang et al., 2023; Zhao et al., 2023; Wang et al., 2024; First et al., 2023; Lin et al., 2025a;b; Wang et al., 2025; Chen et al., 2025). These proof search approaches are complementary to a hammer, which serves as a tactic that may be used by neural models.

Hammers are embedded in a number of neural theorem proving frameworks such as Thor and Draft, Sketch, and Prove (Jiang et al., 2022; Zhao et al., 2023; Jiang et al., 2023; Wang et al., 2024) to fill small gaps in the proofs. It is worth noticing that all these works use the Isabelle proof assistant (Nipkow et al., 2002) where the communication infrastructure (Jiang et al., 2021) between neural models and the proof assistant is relatively mature and hammering is easy to set up. Our work makes calling a hammer in Lean possible.

Despite a large number of research works, practical tools that a working mathematician has access to without complex setup or prohibitive costs remain scarce. Recent state-of-the-art methods use reinforcement learning on e.g. 7B LLMs with thousands of passes for a single theorem and use infrastructures not callable from Lean (Wu et al., 2024; Lin et al., 2025a;b; Dong & Ma, 2025; Chen et al., 2025), so it is prohibitive for Lean users to train, test, or use them. Our work brings forward a tool that is packaged as a tactic and can be called straightforwardly from any IDE for Lean with low computational cost and latency, hence enabling better automation for the masses.

2.3 Premise selection

Formalizing mathematics in proof assistants requires users to select relevant premises from libraries of hundreds of thousands of facts. To help facilitate this task, premise selection has been developed for a variety of proof assistants, using both neural and symbolic techniques. MePo (Meng & Paulson, 2009) is a symbolic premise selector which has been widely used in Isabelle's Sledgehammer. Other premise selectors which target hammers but use traditional machine learning techniques include MaSh (Kühlwein et al., 2013), k-NN based premise selection for HOL4 (Gauthier & Kaliszyk, 2015), CoqHammer's premise selection (Czajka & Kaliszyk, 2018), and random forest based premise selection for Lean (Piotrowski et al., 2023). LeanPremise differs from these by using modern LM-based retrieval methods.

(L)LM-based premise selection trained by contrastive learning has also been explored for a variety of use cases. Lean State Search (Tao et al., 2025) recommends relevant premises directly to Lean users. Magnushammer (Mikuła et al., 2024) generates premises to supply directly to proof reconstruction tactics. ReProver and Lean Copilot (Yang et al., 2023; Song et al., 2024) retrieve premises to augment neural next-tactic generation. Unlike these, LeanPremise is specifically designed with hammer integration in mind, which requires specific data extraction and loss formulation, and the resulting selector to be fast and domain general.

3 Methods

3.1 LeanHammer pipeline

Hammers broadly consist of three primary components: premise selection, translation to external automatic theorem provers, and proof reconstruction. In traditional hammer pipelines, such as Isabelle's Sledgehammer, these components are composed in a linear fashion, with the premises from premise selection informing the translation to automatic theorem provers and the output from automatic theorem provers informing proof reconstruction. In other works, such as Magnushammer (Mikuła et al., 2024) and Lean Copilot (Song et al., 2024), premises from premise selection are provided directly to proof reconstruction tactics or language models, without translating and sending them to external automatic theorem provers. LeanHammer introduces a new, unified hammer pipeline that uses premise selection in both of these ways.

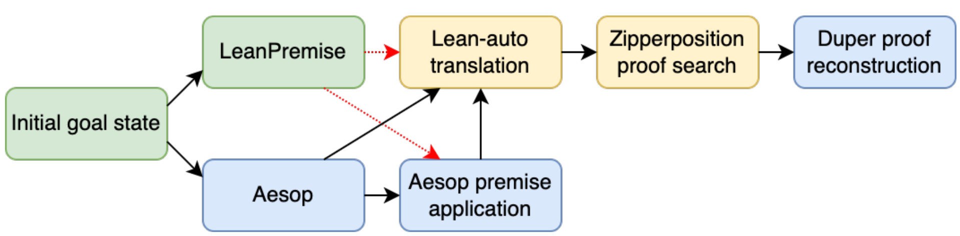


Figure 1: Overview of the LeanHammer pipeline. Phases that can neither fail nor produce a terminal proof are green, phases that can fail but cannot produce a terminal proof are yellow, and phases that can produce a terminal proof are blue. Black solid arrows indicate control flow, while red dashed arrows indicate the transfer of information between phases.

Figure 1 gives an overview of the LeanHammer pipeline. In addition to LeanPremise itself, LeanHammer is built upon Aesop, Lean-auto, and Duper. Aesop is a highly extensible proof search tool that can be augmented with new proof search rules and procedures. Lean-auto is a translation tool that does not search for proofs itself, but instead translates dependently typed Lean goals into higher-order logic problems which can be solved by external automatic theorem provers such as Zipperposition. Finally, Duper is a less powerful but proof-producing proof search tool which implements many of the techniques found in automatic theorem provers, and is therefore well suited to rediscovering and verifying proofs found by external automatic theorem provers.

In broad strokes, Aesop is called first and prioritizes finding a proof using its own built-in rules. If a short proof using only built-in rules is not found quickly, it explores direct premise applications using premises recommended by LEANPREMISE,[1] and it queries Lean-auto to see if subgoals can be closed using premises from the selector.[2] When Lean-auto is given a subgoal, it translates that subgoal (along with the premises provided by the selector) to higher-order logic and interfaces with Zipperposition to find a proof. If Zipperposition succeeds, then Duper is provided just the set of premises used by Zipperposition to solve the translated problem, and Duper attempts to reconstruct a proof from these premises. For an illustrative example of LEANHAMMER's pipeline in action, see Section B.

3.2 DATA EXTRACTION

To support LEANPREMISE, we develop a data extraction pipeline designed to gather not just information useful for next-tactic generation or human examination, but all of the information that may be helpful for a hammer tasked with discovering an end-to-end proof. This pipeline is used dynamically to extract premises that LEANPREMISE can retrieve at runtime, including premises or definitions defined by the user locally, and it is used statically to extract (state, premise) pairs for training. As we describe our data extraction pipeline, we note the measures taken to collect data that go beyond what appears explicitly in the source code for the formal proofs.

3.2.1 SIGNATURE EXTRACTION

A key aspect of premise selectors is how premises are presented to the model. Previous work (Yang et al., 2023) extracts raw strings from the source code, which ignores many details in the full signature (see Section A for an example). We adopt a new *normalized serialization* as follows. For each theorem and definition in each module, we extract the documentation description in the source code (its docstring), if it exists, as well as its kind (theorem or definition), name, arguments, and overall type. Together, these can be composed into a signature of the form `docstring? kind name arguments* : type`. When converting these signatures into strings, we disable notation pretty printing (e.g. we print ℕ as `Nat`), and we print every constant with its fully qualified name (e.g. we print `I` as `Complex.I`). This standardizes premise representation, so that it depends only on the type of the premise and does not depend on open namespaces, custom notations, and surface-level syntax, which may change at run time. For an illustrative example, see Section A.

The signatures extracted in this manner are used to form the set of premises $\mathcal{P}$ that LEANPREMISE is allowed to retrieve from. This signature extraction pipeline is also used to dynamically extract new premises at runtime (Section 3.3.2). To prevent LEANPREMISE from constantly recommending theorems that are technically relevant to the goal but never useful for our hammer's automation, we filter out a blacklist of 479 basic logic theorems such as `and_true` from $\mathcal{P}$. We also filter out Lean language-related (e.g. metaprogramming) definitions not useful for proofs.

3.2.2 STATE AND PREMISE EXTRACTION

The next key question is which (state, premise) pairs are extracted from human-written proofs to train the model. Previous premise selectors (Yang et al., 2023) that focus on tactic generation only extract from tactic-style proofs, and only extract explicit premises appearing in the raw source code of only the next tactic. Our *hammer-aware data extraction* improves upon this in several ways. First, we extract from both term-style and tactic-style proofs, significantly increasing training samples especially for short proofs that LEANHAMMER is intended to automate. Second, for multi-tactic

[1] Premise applications are rules added to Aesop of the form `(add unsafe 20% <premise>)` where `<premise>` is a premise selected by the premise selector.

[2] Lean-auto is added to Aesop as a rule of the form `(add unsafe 10% (by auto [*, <premises>]))` where `<premises>` is a list of premises selected by the premise selector.

proofs, the model is trained to select premises to close the goal (all tactics) rather than to modify the goal (first tactic), because hammers are designed to finish proofs. Third, we extract both implicit and explicit premises from the proof, including ones implicitly called by automation such as `simp`. Finally, we format states with the same normalized serialization as for premises.

Specifically, for each theorem in each module, we collect data on the premises used to prove it. Additionally, for each theorem proven via tactic-style proofs, we collect data on all intermediate goal states induced by the tactic sequence. For an illustrative example, see Section A. Ultimately, all data we extract contains:

- A proof state obtained either from the beginning of a theorem or from an intermediate step of a tactic-style proof.
- The name and signature of the theorem from which the state was extracted.
- The set of premises used to prove the theorem[3].

When theorems are proven via term-style proofs, meaning the theorem's proof term is explicitly written in the source code, the set of premises we extract is the set of theorems that appear in the proof term. When theorems are proven via tactic-style proofs, meaning automation is invoked to tell Lean how to build a proof term, the set of premises we extract contains both the theorems that appear in the proof term constructed by the tactic sequence (so that all implicit premises are collected), as well as any theorems and definitions that are explicitly used in `rw` or `simp` calls.

The benefit of collecting explicit theorems and definitions from `rw` and `simp` calls relates to Lean's dependent type theory. In Lean, terms can be definitionally equal without being syntactically equal, and because of this, tactic-style proofs can invoke definitional equalities that do not appear in final proof terms. We therefore collect these definitional equality premises. We experimentally verify that our hammer-aware data extraction benefits LEANHAMMER in Section 4.4.

3.3 PREMISE SELECTION

LEANPREMISE uses the standard method of retrieval using textual encoders. In order to retrieve k premises for a state s, we first determine the set $\mathcal{P}_s$ of accessible premises at position s, comprising lemmas and definitions that are imported from other modules or declared earlier in the file. We use an encoder-only transformer model E to embed both the state s and every premise $p \in \mathcal{P}_s$, and the resulting set of premises retrieved is

$$\texttt{select\_premises}(s, k, \mathcal{P}_s) = \text{top-}k_{p\in\mathcal{P}_s}\texttt{sim}(E(s), E(p)) \tag{1}$$

where $\texttt{sim}(u, v) = u^\top v/\|u\|_2\|v\|_2$ is cosine similarity. In Section 3.3.2 we describe the mechanism for caching embedding and quick retrieval of the premises.

We do not train a separate reranking model as in Mikuła et al. (2024), because we did not find it to increase performance in early experiments, especially since a hammer favors recall much more than precision, and we determined the optimal k to be at least 16, at which point reranking does not offer much improvement. It is also costly to deploy in practice.

3.3.1 MODEL TRAINING

We use a modified version of the InfoNCE loss (Oord et al., 2018) to train the encoder model. On a high level, each batch consists of (state, premise) pairs, and a contrastive loss is used to let the model learn to select the correct premise out of all premises in this batch. One problem is that there are many premises in the library that do not appear in any proof. This is mitigated by also sampling negative premises in each batch (Mikuła et al., 2024; Yang et al., 2023). Another problem is that there are many premises that are shared across many proofs, so not all premises in the batch are negative. We use the following masked contrastive loss to address these problems.

Specifically, for each training step, we sample a batch of B (state, premise) pairs, each consisting of a state s_i and a premise $p_i^+ \in \mathcal{P}_{s_i}^+$ where $\mathcal{P}_{s_i}^+$ is the set of ground-truth premises for s_i extracted as

[3] We also experimented with pairing states with just the set of premises used to close said states, as opposed to all premises used to prove the overall theorem, but our preliminary experiments showed that this yielded worse results than including all premises.

Premise selector	LM-based	Callable in Lean	New premises
ReProver (Yang et al., 2023)	✓	✗	✗
Lean Copilot (Song et al., 2024)	✓	✓	✗
Random forest (Piotrowski et al., 2023)	✗	✓	✗
MePo (Meng & Paulson, 2009)	✗	✓	✓
LEANPREMISE	✓	✓	✓

Table 1: Usability comparison of existing premise selection tools. Note that this is orthogonal to the quantitative performance comparisons (Table 2).

in Section 3.2. For each such pair (s_i, p_i^+), we sample B^- negative premises $\{p_{ij}^-\}_{j=1}^{B^-} \subseteq \mathcal{P}_{s_i} \setminus \mathcal{P}_{s_i}^+$, giving B states and $B(1+B^-)$ premises in total in each batch. Of these premises, we determine the set $\mathcal{N}_i = \{p_i^+\}_i \cup \{p_{ij}^-\}_{ij} \setminus \mathcal{P}_{s_i}^+$ of negative premises for state s_i, and mask out the positive ones in the loss to avoid mislabeling. The loss is:

$$\mathcal{L}(E) = \frac{1}{B}\sum_{i=1}^{B} \frac{\exp(\texttt{sim}(E(s_i), E(p_i^+))/\tau)}{\exp(\texttt{sim}(E(s_i), E(p_i^+))/\tau) + \sum_{p_i^- \in \mathcal{N}_i} \exp(\texttt{sim}(E(s_i), E(p_i^-))/\tau)} \quad (2)$$

where τ is a scalar temperature hyper-parameter (set to 0.05 in our experiments).

3.3.2 API INTEGRATION

In order to make LEANPREMISE and LEANHAMMER more accessible for Lean users as well as downstream methods, we design our pipeline to maximize usability—it is directly callable in Lean, able to take in new premises, and efficient to run. Our pipeline for premise selection is as follows: when a user invokes premise selection, the client side (Lean) collects all currently defined premises $\mathcal{P}_s$ defined in the environment and the current proof state s and sends them to a server that hosts the embedding model. The server embeds both the proof state and the list of premises, and then runs FAISS (Douze et al., 2024) on the premises to compute $\texttt{select\_premises}(s, k, \mathcal{P}_s)$, and returns this list of k premises back to the client. Since the typical size of $\mathcal{P}_s$ is on the scale of $\sim$70k, the server also caches the embeddings of premises at fixed versions of Mathlib, and only recomputes embeddings of signatures of new premises uploaded by the user (e.g. when working outside Mathlib or when the user has new premises in the context); the client side also caches the signatures of these new premises computed as in Section 3.2.1.

LEANHAMMER is built as a tactic that can be directly called in Lean. It calls LEANPREMISE as a subprocedure and the retrieved premises are then input to the LEANHAMMER pipeline (Section 3.1). In Mathlib, premise selection usually takes about 1 second amortized on a CPU server (and well under 1 second for a single-GPU server). The full LEANHAMMER pipeline on average takes well under 10 seconds (see Section 4.2).

To the best of our knowledge, LEANPREMISE is the first premise selector using language models that can be directly invoked in Lean and can incorporate new user-defined premises. It is also efficient to run and requires no system setup for the user, because the main computation is only a few string embeddings, and done centrally in a server by default. This makes the premise selector itself a desirable user-facing tactic for the Lean community. Similarly, the full LEANHAMMER can be called straightforwardly in Lean as a tactic. This bridges a gap that many previous LM-based retrievers and provers leave. See Table 1 for a comparison.

3.4 VARIATIONS AND EXTENSIONS

Here, we discuss variations on LEANHAMMER's design, implemented as settings that can be controlled by the user. Note that the pipeline described in Section 3.1 has premises input both to Aesop as premise application rules and to Lean-auto for translation to the external prover. We consider variants that disable either one:

1. `aesop`: This setting only inputs LEANPREMISE's premises to Aesop as premise applications, omitting calls to Lean-auto or the external prover.

2. `auto`: This setting inputs LEANPREMISE's premises directly to the external prover through Lean-auto without Aesop normalization or premise application.
3. `aesop+auto`: This setting keeps both Aesop and Lean-auto, but does not use premise applications as Aesop rules.
4. `full`: This default setting is the full pipeline described in Section 3.1.

These variants are appealing because they offer cheaper computational cost while still preserving much of LEANHAMMER's ability. Experiments offer insight to the ability of each part of the pipeline (see Section 4.2). We observe that the first three variants may prove theorems that `full` does not, so we also consider `cumul`, which tries all four variants.

We note that other common domain-general automation tactics that take premises as inputs, such as `simp_all`, may be roughly considered a subcase of `aesop` (which we verify in preliminary experiments), so we do not consider them. We also tried using a second-stage model to predict `simp_all` "hints"—whether a premise should be supplied to `simp_all` for preprocessing, and whether it should be applied in reverse direction, but the performance did not increase. We remark that additional automated reasoning tactics in the future may be easily added to our pipeline as a rule of Aesop, similarly to how Lean-auto is added.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

We extract theorem proofs from Mathlib, and premises from Mathlib, Batteries, and Lean core. In total, we extract 469,965 states from 206,005 theorem proofs, and extract 265,348 (filtered) premises. For each state, there are on average 12.45 relevant premises, giving 5,817,740 (state, premise) pairs in the training set. We randomly hold out 500/500 theorems as valid/test sets, respectively.

We train the model from three base models that were pre-trained for general natural language embedding tasks (Reimers & Gurevych, 2019). These are `small`[4] with 6 layers and hidden size 384 trained from MiniLM-L6, `medium`[5] with 12 and 384 from MiniLM-L12, and `large`[6] with 6 and 768 from DistilRoBERTa-base, respectively. We train our models with learning rate $2e{-}4$, $B = 256$, and $B^- = 3$, found by a hyperparameter sweep. Training the `large` model requires 6.5 A6000-days.

We test LEANHAMMER on proving theorems in (1) our hold-out sets extracted from Mathlib, and (2) the non-Mathlib splits of `miniCTX`-v2-test (Hu et al., 2025). We impose a 10-second time constraint for each call to Zipperposition, and for each theorem a 300-second wall-clock time-out and Lean's default heartbeat limit of 200,000. We tuned the value of k on Mathlib-valid (see Section D.3), and `full` uses the highest performing combination, which is $k_1 = 16$ premises supplied to Lean-auto (with Aesop priority 10%) and $k_2 = 32$ premises for premise application rules (with Aesop priority 20%). Similarly, we use $k = 16$ for `auto` and `aesop+auto`, and $k = 32$ for `aesop`.

For all experiments and data extraction tasks, we use Lean version v4.16.0. We run a maximum of 16 parallel tests on 16 CPUs with 512GB total memory, so 1 CPU and 32GB are allocated per test theorem. In practice, the actual memory used rarely exceeds 4GB. Each CPU is AMD EPYC 9354 (3.8GHz, 32 cores, 64 threads) or similar.

4.2 RESULTS

For each theorem, we record the average percentage of ground-truth premises retrieved in the top-k premises (*recall@k*), and the percentage of theorems proven (*proof rate*), shown in Tables 2 and 3. We favor recall over metrics like precision, because a hammer can tolerate irrelevant premises much more than missing important ones.

LEANHAMMER proves a significant number of theorems. As shown in Table 2, we find that LEANHAMMER proves a significant proportion of test theorems, with 33.3% proved by the `large`

[4]https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
[5]https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2
[6]https://huggingface.co/sentence-transformers/all-distilroberta-v1

Premise selector	Model size	Recall (%)		Proof rate (%)				
		@16	@32	`aesop`	`auto`	`aesop+auto`	`full`	`cumul`
None	—	0.0	0.0	16.9	9.4	16.9	16.9	16.9
Random forest*	—	22.1	22.3	19.1	11.9	19.1	19.1	19.1
MePo	—	38.4	42.1	23.3	14.5	21.5	26.3	27.5
ReProver	218M	35.1[†]	38.7[†]	11.4	12.9	20.5	12.0	22.3
LeanPremise (`small`)	23M	59.2	67.8	23.9	19.1	25.9	27.9	31.9
LeanPremise (`medium`)	33M	58.6	68.1	23.1	20.1	26.1	28.5	30.7
LeanPremise (`large`)	82M	**63.5**	**72.7**	**24.1**	**21.3**	**28.5**	**30.1**	**33.3**
LeanPremise (`large`) ∪ MePo				28.9	23.9	30.3	35.9	37.6
LeanPremise (cumulative)				27.5	25.5	31.1	34.5	37.3
LeanPremise (cumulative) ∪ MePo				30.3	27.1	32.3	38.2	39.6
Ground truth				27.7	33.1	37.8	41.0	43.0

*Performance upper bound, excluding errors. [†]Our definition is slightly different from Yang et al. (2023). See Section C.

Table 2: Performance of LeanHammer with different premise selectors on Mathlib-test.

Premise selector	Proof rate using `full` (%)						
	Carleson	ConNF	FLT	Foundation	HepLean	Seymour	Average
None	0.0	10.0	27.3	38.0	8.0	6.0	14.9
LeanPremise (`large`)	0.0	16.0	36.4	38.0	10.0	24.0	20.7
Ground truth	7.1	16.0	39.4	40.0	20.0	34.0	26.1

Table 3: Out-of-Mathlib performance of LeanHammer on `miniCTX`-v2-test (Hu et al., 2025) using the `large` model trained on Mathlib. For other settings than `full`, see Table 5.

model in the `cumul` setting, and 37.3% when accumulated over model sizes. We also test giving ground-truth premises (those that appear in the human-written proof) to LeanHammer, which serves as a theoretical best-case scenario of how LeanHammer would perform if the models achieved 100% recall, and this proves 43.0% of the theorems. Compared to previous work, LeanHammer approaches this limit in the settings considered.

Performance scales with model size and accumulation. In Table 2 and Figure 2a, as we increase our model size, for most settings the recall and proof rates also correspondingly increase (e.g., recall@32 increases from 67.8% to 72.7% and `full` proof rate increases from 27.9% to 30.1%). We also observe that by accumulating across different model sizes or taking the union of neural (our model) and symbolic (MePo) approaches, the proof rate increases much more than scaling the model alone (e.g., `full` proof rate increases to 34.5% when accumulated), meaning different selectors prove different sets of theorems. More effective methods of ensembling models may be explored in future work.

LeanHammer settings offer different abilities at different costs. For the settings `auto`, `aesop`, `aesop+auto`, and `full`, the proof rate roughly increases in this order for all models. This shows that each part of the full pipeline incrementally contributes to the final proof rate. Their mean run times on Mathlib-test are 4.3s, 0.92s, 4.9s, and 6.6s respectively, so the non-`full` variants are computationally appealing alternatives that recover some of the `full` performance. We also note that `cumul` achieves higher proof rates than `full`, so some cases benefit from a partial pipeline (e.g. if the `full` pipeline does not terminate).

LeanHammer shows robust out-of-Mathlib generalization. As shown in Table 3, the performance on `miniCTX`-v2-test (Hu et al., 2025) is comparable to the performance on Mathlib—the proportion of theorems proven by LeanHammer with the `large` selector, out of theorems proven with the ground-truth premises (i.e. the best-case scenario), is 73.5% on Mathlib and 79.4% on `miniCTX` with the `full` pipeline, showing that performance does not decrease (the other settings also have comparable numbers; see Table 5). We also confirm in the table that if no premises are supplied, the performance is much worse (except for the Foundation split), which indicates that the LeanHammer is not just proving trivial theorems.

Premise selector	Recall (%)		Proof rate (%)				
	@16	@32	`aesop`	`auto`	`aesop+auto`	`full`	`cumul`
LEANPREMISE (`medium`)	61.1	71.9	29.8	22.6	30.2	34.6	37.6
+ naive data	57.5	66.8	29.3	20.0	28.5	33.1	35.2
− negatives	51.8	59.5	28.6	20.0	28.4	33.0	36.8
− loss mask	59.1	69.6	29.4	21.2	29.0	34.4	38.4
Ground truth			30.8	32.0	38.4	41.2	43.6
+ naive data			31.2	30.0	37.0	39.8	42.4

Table 4: Ablation study of LEANHAMMER with different training settings on Mathlib-valid.

Across all benchmarks, there are a handful of common patterns characterizing problems that LEANHAMMER fails to solve. Some problems are not solved because LEANPREMISE fails to retrieve necessary lemmas, as can be seen from the fact that the ground truth outperforms all other premise selectors in Tables 2 and 3. Some problems are not solved because they are out of scope for Lean-auto's translation procedure, which can occur when the problem in question contains features from dependent type theory not easily translated to higher-order logic. And some problems are not solved because the solutions require forms of reasoning not supported by Aesop, Zipperposition, or Duper (e.g. induction or arithmetic). Comparatively, it is rare for LEANHAMMER to succeed at proof search with Zipperposition but fail at the proof reconstruction stage with Duper. See Sections D.4 and D.5 for additional analysis.

4.3 COMPARISONS

We compare LEANPREMISE against the following existing work: non-LM methods MePo (Meng & Paulson, 2009) and Piotrowski et al. (2023), and LM-based ReProver (Yang et al., 2023). We use a recent adaptation of MePo from Isabelle to Lean (implemented by Kim Morrison), tune its parameters p and c on our evaluation recall@k, and apply our premise blacklist. For Piotrowski et al. (2023), we select their random forest model with highest reported performance; in order to overcome errors, we modified its training and evaluation in a way that only gives them unfair advantage, so the reported performance is an upper bound. (See Section C for details of both methods.) We find that LEANPREMISE clearly outperforms either method—for the `large` model, our recall@32 is 73% higher relative to MePo and our `cumul` proof rate is 21% higher (Table 2). Meanwhile, the theorems that the union of theorems our models and MePo can solve is much higher than each method separately, indicating that symbolic and neural methods have complementary strengths. We believe effective combinations of neural and symbolic methods warrant future investigation.

We retrain ReProver using their training and retrieval scripts, but on our train/valid/test splits and an updated Mathlib version (Section C). LEANPREMISE clearly outperforms ReProver (Yang et al., 2023) in terms of recall and proof rate—LEANHAMMER using our `large` model (82M parameters) proves 150% more theorems relative to using ReProver (218M) in the `full` setting and 50% more in the `cumul` setting. We attribute the performance gap to two main factors. First, ReProver focuses on premises used in the next tactic for tactic generation, while LEANPREMISE focuses on finishing the entire proof, so the definitions of ground-truth premises are different (Section 3.2.2). Second, LEANHAMMER uses techniques such as term-style proof extraction, extraction of implicit premises, and better premise signature formatting (Section 3.2). ReProver also uses an ℓ^2 loss on the cosine similarity for training, rather than our contrastive loss, and we suspect this also contributes to our better performance.

4.4 ABLATIONS

Table 4 shows the performance of LEANHAMMER on Mathlib-valid with some components removed: (1) we use a naive data extraction script that (a) uses default pretty-printing options, (b) disables our premise blacklist, and (c) disables collection of premises from `simp` or `rw` calls; (2) we do not sample negative premises during training ($B^- = 0$); and (3) we disable masking positive in-batch premises in the contrastive loss, i.e. the denominator of Equation (2) being simply the sum over all $B(1 + B^-)$ premises in batch. We observe these changes clearly degrade performance.

Specifically, our data extraction (Section 3.2) is specifically designed with Lean-auto translation in mind, and we observe that settings with Lean-auto have a lower performance with a naive data extraction script. We observe that randomly sampling negative premises and the loss mask (Section 3.3.1) improve performance. (Although the `cumul` proof rate of LEANHAMMER without the loss mask is higher, we strongly believe this is due to random noise, because all individual settings give lower performances, the recall is lower, and proof rate has higher variance than recall.)

5 CONCLUSION

We developed LEANPREMISE, a novel premise selection tool for a hammer in dependent type theory, and combined neural premise selection with symbolic automation to build LEANHAMMER, the first domain-general hammer in Lean. With comprehensive experiments, we show that LEANHAMMER is performant on Mathlib compared to baselines, and generalizes well to `miniCTX`-v2. LEANPREMISE and LEANHAMMER are designed with accessibility for Lean users in mind, and lay down groundwork for future hammer-based neural theorem proving in Lean.

REPRODUCIBILITY STATEMENT

We make all code for data extraction, model training, evaluation, and API integration publicly available with an open-source license. We open source both LEANPREMISE and LEANHAMMER as tactics in Lean. We also release the extracted data and all trained models and baselines.

REFERENCES

Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 415–442, Cham, 2022. Springer International Publishing. doi: 10.1007/978-3-030-99524-9_24.

Lasse Blaauwbroek, Mirek Olšák, Jason Rute, Fidel Ivan Schaposnik Massolo, Jelle Piepenbrock, and Vasily Pestun. Graph2tac: Online representation learning of formal math concepts. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=A7CtiozznN.

Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *J. Autom. Reason.*, 51(1):109–128, 2013. doi: 10.1007/S10817-013-9278-5.

Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C Paulson, and Josef Urban. Hammering towards qed. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

Mario Carneiro, Chad E. Brown, and Josef Urban. Automated Theorem Proving for Metamath. In Adam Naumowicz and René Thiemann (eds.), *Interactive Theorem Proving (ITP)*, pp. 9:1–9:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.ITP.2023.9.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, et al. Seed-prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*, 2025.

Joshua Clune, Yicheng Qian, Alexander Bentkamp, and Jeremy Avigad. Duper: A Proof-Producing Superposition Theorem Prover for Dependent Type Theory. In Yves Bertot, Temur Kutsia, and Michael Norrish (eds.), *Interactive Theorem Proving (ITP)*, volume 309, pp. 10:1–10:20, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.ITP.2024.10.

Lukasz Czajka and C. Kaliszyk. Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61:423 – 453, 2018. URL https://api.semanticscholar.org/CorpusID:11060917.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi: 10.1007/978-3-540-78800-3_24.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe (eds.), *Conference On Automated Deduction (CADE)*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

Kefan Dong and Tengyu Ma. Beyond limited data: Self-play llm theorem provers with iterative conjecturing and proving. *arXiv preprint arXiv:2502.00212*, 2025.

Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In Satish Chandra, Kelly Blincoe, and Paolo Tonella (eds.), *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pp. 1229–1241. ACM, 2023. doi: 10.1145/3611643.3616243. URL `https://doi.org/10.1145/3611643.3616243`.

Thibault Gauthier and Cezary Kaliszyk. Premise selection and external provers for HOL4. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, New York, NY, USA, January 2015. ACM.

Jiewen Hu, Thomas Zhu, and Sean Welleck. miniCTX: Neural theorem proving with (long-)contexts. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=KIgaAqEFHW`.

Albert Q. Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. Lisa: Language models of isabelle proofs. In *6th Conference on Artificial Intelligence and Theorem Proving*, pp. 378–392, 2021.

Albert Q. Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygózdz, Piotr Milos, Yuhuai Wu, and Mateja Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL `http://papers.nips.cc/paper_files/paper/2022/hash/377c25312668e48f2e531e2f2c422483-Abstract-Conference.html`.

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2023. URL `https://openreview.net/forum?id=SMa9EAovKMC`.

Cezary Kaliszyk and Josef Urban. Hol(y)hammer: Online ATP service for HOL light. *Mathematics in Computer Science*, 9(1):5–22, 2015a. doi: 10.1007/s11786-014-0182-0.

Cezary Kaliszyk and Josef Urban. Mizar 40 for mizar 40. *Journal of Automated Reasoning*, 55(3): 245–256, 2015b. doi: 10.1007/s10817-015-9330-8.

Saketh Ram Kasibatla, Arpan Agarwal, Yuriy Brun, Sorin Lerner, Talia Ringer, and Emily First. Cobblestone: Iterative automation for formal verification. *CoRR*, abs/2410.19940, 2024. doi: 10.48550/ARXIV.2410.19940. URL `https://doi.org/10.48550/arXiv.2410.19940`.

Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith (eds.), *Computer Aided Verification (CAV)*, pp. 1–35, Berlin, Heidelberg, 2013. Springer.

Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. Mash: machine learning for sledgehammer. In *Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings 4*, pp. 35–50. Springer, 2013.

Guillaume Lample, Timothée Lacroix, Marie-Anne Lachaux, Aurélien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/a8901c5e85fb8e1823bbf0f755053672-Abstract-Conference.html.

Lean Community. Completion of the liquid tensor experiment. blog post, 2022. URL https://leanprover-community.github.io/blog/posts/lte-final/.

Jannis Limperg and Asta Halkjær From. Aesop: White-box best-first proof search for lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 253–266, 2023.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, et al. Goedel-prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*, 2025a.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, et al. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*, 2025b.

Minghai Lu, Benjamin Delaware, and Tianyi Zhang. Proof automation with large language models. In Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (eds.), *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, pp. 1509–1520. ACM, 2024. doi: 10.1145/3691620.3695521. URL https://doi.org/10.1145/3691620.3695521.

Jia Meng and Lawrence C Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009.

Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006. doi: https://doi.org/10.1016/j.ic.2005.05.010.

Maciej Mikuła, Szymon Tworkowski, Szymon Antoniak, Bartosz Piotrowski, Albert Q. Jiang, Jin Peng Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. Magnushammer: A transformer-based approach to premise selection. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=oYjPk8mqAV.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2002. doi: 10.1007/3-540-45949-9.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

Lawrence Charles Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In *IWIL@LPAR*, 2012. URL https://api.semanticscholar.org/CorpusID:598752.

Bartosz Piotrowski, Ramon Fernández Mir, and Edward Ayers. Machine-learned premise selection for lean. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pp. 175–186. Springer, 2023.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=-P7G-8dmSh4.

Yicheng Qian, Joshua Clune, Clark Barrett, and Jeremy Avigad. Lean-auto: An interface between lean 4 and automated theorem provers. In *Lecture Notes in Computer Science*, Lecture notes in computer science, pp. 175–196. Springer Nature Switzerland, Cham, 2025.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. URL https://arxiv.org/abs/1908.10084.

Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pascal Fontaine (ed.), *Conference on Automated Deduction (CADE)*, pp. 495–507. Springer, 2019.

Peiyang Song, Kaiyu Yang, and Anima Anandkumar. Lean copilot: Large language models as copilots for theorem proving in lean. *arXiv preprint arXiv:2404.12534*, 2024.

Terence Tao. Formalizing the proof of pfr in lean4 using blueprint: a short tour. blog post, 2023. URL https://terrytao.wordpress.com/2023/11/18/formalizing-the-proof-of-pfr-in-lean4-using-blueprint-a-short-tour/.

Yicheng Tao, Haotian Liu, Shanwen Wang, and Hongteng Xu. Assisting mathematical formalization with a learning-based premise retriever. *arXiv preprint arXiv:2501.13959*, 2025.

The Mathlib Community. The lean mathematical library. In Jasmin Blanchette and Catalin Hritcu (eds.), *Certified Programs and Proofs (CPP)*, pp. 367–381. ACM, 2020. doi: 10.1145/3372885.3373824.

Floris van Doorn, Patrick Massot, and Oliver Nash. Formalising the h-principle and sphere eversion. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic (eds.), *Certified Programs and Proofs (CPP)*, pp. 121–134. ACM, 2023. doi: 10.1145/3573105.3575688.

Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. Making higher-order superposition work. *Journal of Automated Reasoning*, 66: 541–564, 2022. doi: 10.1007/s10817-021-09613-z.

Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. Lego-prover: Neural theorem proving with growing libraries. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=3f5PALef5B.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025.

Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Jiayu Wang, Dahua Lin, and Kai Chen. Internlm2. 5-stepprover: Advancing automated theorem proving via expert iteration on large-scale lean problems. *arXiv preprint arXiv:2410.15700*, 2024.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *CoRR*, abs/2405.14333, 2024a. doi: 10.48550/ARXIV.2405.14333. URL https://doi.org/10.48550/arXiv.2405.14333.

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *CoRR*, abs/2408.08152, 2024b. doi: 10.48550/ARXIV.2408.08152. URL https://doi.org/10.48550/arXiv.2408.08152.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS)*, 2023.

Xueliang Zhao, Wenda Li, and Lingpeng Kong. Decomposing the enigma: Subgoal-based demonstration learning for formal theorem proving. *CoRR*, abs/2305.16366, 2023. doi: 10.48550/ARXIV.2305.16366. URL https://doi.org/10.48550/arXiv.2305.16366.

A Data extraction example

We provide an example of proof state and premise extraction to illustrate the features of our data extraction pipeline.

Consider the following theorem, `AlgebraicGeometry.Scheme.RationalMap.mem_domain`, proven in the module `Mathlib.AlgebraicGeometry.RationalMap`:

```
lemma RationalMap.mem_domain {f : X --→ Y} {x} :
    x ∈ f.domain ↔ ∃ g : X.PartialMap Y, x ∈ g.domain ∧ g.toRationalMap = f :=
  TopologicalSpace.Opens.mem_sSup.trans (by simp [@and_comm (x ∈ _)])
```

We extract its proof data, including its intermediate proof states and premises used (Section 3.2.2). This is a term-style proof (the proof does not start with `by`), and our data extraction extracts premises corresponding to the whole proof of the theorem, with state (note the pretty-printing options that disable notations and print full names):

```
X Y : AlgebraicGeometry.Scheme
f : X.RationalMap Y
⊢ ∀ {x : ↑↑X.toPresheafedSpace},
    Iff (Membership.mem f.domain x) (Exists fun g => And (Membership.mem g.domain
    x) (Eq g.toRationalMap f))
```

and the ground-truth set of premises used, which is the union of premises appearing in the compiled proof term (e.g. `exists_exists_and_eq_and` implicitly invoked by `simp`) and appearing explicitly (e.g. `and_comm` appearing in the argument of the `simp` call): [`Iff.trans`, `exists_prop`, `Eq.trans`, `of_eq_true`, `iff_self`, `congr`, `funext`, `exists_exists_and_eq_and`, `TopologicalSpace.Opens.mem_sSup`, `propext`, `and_comm`, `congrArg`].

This set of premises are then filtered by a blacklist of common logical premises and other ineligible premises, which removes trivial theorems such as `iff_self (p : Prop) : (p ↔ p) = true`. This helps the selector to only focus on the smaller subset of premises that are meaningful for a hammer. The resulting list of premises is [`exists_prop`, `funext`, `exists_exists_and_eq_and`, `TopologicalSpace.Opens.mem_sSup`].

For tactic-style (sub)proofs, the state before each tactic is also collected. In the proof above, this is the state before the `simp` call:

```
X Y : AlgebraicGeometry.Scheme
f : X.RationalMap Y
x : ↑↑X.toPresheafedSpace
⊢ Iff
    (Exists fun u =>
      And (Membership.mem (setOf fun x => Exists fun g => Exists fun x_1 => Eq
    g.domain x) u) (Membership.mem u x))
    (Exists fun g => And (Membership.mem g.domain x) (Eq g.toRationalMap f))
```

This state has the same ground-truth premises as the state above, since they correspond to the same theorem proof. Compare this data extraction with prior work (Yang et al., 2023) that only extract premises that explicitly occur in the next tactic in tactic-style proofs (which is sensible for their purpose of textual next-tactic generation but not for a hammer that needs all relevant premises to close the current goal).

Aside from extracting data from its proof, this theorem may be used as a premise for down-stream theorems. Therefore we also collect it as a premise (Section 3.2.1), with pretty-printed signature as follows:[7]

```
theorem AlgebraicGeometry.Scheme.RationalMap.mem_domain {X Y :
    AlgebraicGeometry.Scheme} {f : X.RationalMap Y} {x : ↑↑X.toPresheafedSpace} :
    Iff (Membership.mem f.domain x) (Exists fun g => And (Membership.mem g.domain
    x) (Eq g.toRationalMap f))
```

[7] We tried excluding the theorem name (here `AlgebraicGeometry.Scheme.RationalMap.mem_domain`) in a premise signature, but preliminary ablation experiments did not show performance gains.

Note the difference between this signature and the raw source code string at the start: pretty-printing notation shorthands like `-->` and ∃ is disabled, names are expanded to full names, implicit types are added (such as the type of `x`), previously declared variables like `X` and `Y` are included, and the proof is not included. This is because our signature printing is a function of the type of the premise and not its source string as in Yang et al. (2023). This gives the entire information of a premise while standardizing its signature printing.

B LEANHAMMER EXAMPLE

We provide an example of a proof produced by LEANHAMMER to illustrate the features of our hammer pipeline.

Consider the following theorem, `associated_gcd_right_iff`, proven in the module `Mathlib.Algebra.GCDMonoid.Basic`:

```
theorem associated_gcd_right_iff [GCDMonoid A] {x y : A} :
    Associated y (gcd x y) ↔ y ∣ x :=
  ⟨fun hx => hx.dvd.trans (gcd_dvd_left x y),
    fun hxy => associated_of_dvd_dvd (dvd_gcd hxy dvd_rfl) (gcd_dvd_right x y)⟩
```

The initial goal state produced by this theorem's signature is:

```
A : Type
inst¹ : CancelCommMonoidWithZero A
inst : GCDMonoid A
x y : A
⊢ Associated y (gcd x y) ↔ y ∣ x
```

Initial goal state (as printed by VS Code)

```
A : Type
inst¹ : CancelCommMonoidWithZero A
inst : GCDMonoid A
x y : A
⊢ Iff (Associated y (GCDMonoid.gcd x y))
    (Dvd.dvd y x)
```

String sent to premise selection server

Note that there are slight differences between this goal state as printed by VS Code and as printed by our state extraction procedure. These differences serve to disambiguate constants and remove potentially overloaded notation, and are described in Section 3.2.1.

Given this initial goal state, our premise selection server looks up premises accessible by this proof, which are premises either imported by the current module or defined earlier in the current module. Among these, it returns the following ordered list of 32 premises:

1. `GCDMonoid.gcd_dvd_left`
2. `dvd_gcd_iff`
3. `GCDMonoid.dvd_gcd`
4. `GCDMonoid.gcd_dvd_right`
5. `Associated.dvd_iff_dvd_right`
6. `Associated.dvd_iff_dvd_left`
7. `gcd_comm'`
8. `gcd_eq_zero_iff`
9. `Associated.symm`
10. `Associated.dvd`
11. `gcd_zero_right'`
12. `Associated.trans`
13. `gcd_dvd_gcd`
14. `Associated.refl`
15. `associated_one_iff_isUnit`
16. `gcd_zero_left'`
17. `instDecompositionMonoidOfNonemptyGCDMonoid`
18. `associated_of_dvd_dvd`
19. `instNonemptyGCDMonoid`
20. `associated_gcd_left_iff`
21. `gcd_mul_lcm`
22. `gcd_assoc'`
23. `dvd_dvd_iff_associated`
24. `gcd_mul_right'`
25. `gcd_mul_left'`
26. `GCDMonoid.gcd_mul_lcm`
27. `dvd_gcd_mul_of_dvd_mul`
28. `gcd_mul_dvd_mul_gcd`
29. `Associated.mul_left`
30. `gcd_one_right'`
31. `mul_dvd_mul_iff_left`
32. `gcd_pow_left_dvd_pow_gcd`

As described in Section 3.4, LEANHAMMER uses $k_1 = 16$ premises supplied to Lean-auto, and $k_2 = 32$ for premise application rules, so the first 16 premises are sent to Lean-auto and all 32 of the above premises are added to Aesop as premise application rules. The proof that LEANHAMMER discovers is equivalent to the following:

```
theorem associated_gcd_right_iff [GCDMonoid A] {x y : A} :
    Associated y (gcd x y) ↔ y ∣ x := by
  apply Iff.intro -- Applied by Aesop
  · intro a -- Applied by Aesop
    duper [a, GCDMonoid.gcd_dvd_left, Associated.dvd_iff_dvd_left]
  · intro a -- Applied by Aesop
    apply associated_of_dvd_dvd -- Premise application (18)
    · duper [a, GCDMonoid.dvd_gcd, Associated.dvd, Associated.refl]
    · apply GCDMonoid.gcd_dvd_right -- Premise application (4)
```

In this proof, Aesop begins by transforming the initial goal into subgoals with the constructor introduction rule `Iff.intro`. The first subgoal is provable using just the first 16 premises supplied by the premise selector, so after Lean-auto translates it into higher-order logic, Zipperposition reports that `GCDMonoid.gcd_dvd_left` (premise 1) and `Associated.dvd_iff_dvd_left` (premise 6) entail the first subgoal on their own. Then, since it is known that only these two premises are required to solve the first subgoal, these two premises can be passed to Duper on their own, and Duper is able to produce a proof for the first subgoal.

The second subgoal cannot be proven by Lean-auto and Zipperposition using just the first 16 premises, but Aesop sees that `associated_of_dvd_dvd` (premise 18) can be applied directly. After it does so, two more subgoals are created, the first of which can once again be proven with Lean-auto, Zipperposition, and Duper (using a different subset of the first 16 premises), and the second of which can be proven with a direct application of `GCDMonoid.gcd_dvd_right` (premise 4). Since `GCDMonoid.gcd_dvd_right` is part of the first 16 premises, it would also be possible for this final subgoal to be proven using Lean-auto, Zipperposition, and Duper, but because direct premise applications are assigned a higher priority than invocations of Lean-auto (20% as compared to 10%, see Section 3.4), Aesop discovers the simpler proof first.

C BASELINE SETTINGS

The following baselines are used in our analysis. The specific setup details of each baseline are listed below:

- None: No premises are supplied to LEANHAMMER.
- MePo (Meng & Paulson, 2009): A prototype implementation of MePo was recently adapted into Lean by Kim Morrison. We note that MePo is designed for selecting a much larger number of premises than what is typically optimal for LEANHAMMER (Section D.3). We select the final k premises selected by MePo, as we experimentally confirm that the final premises selected are the most relevant, while supplying too many premises to LEANHAMMER would decrease its performance (Figure 3). We tuned the parameters p and c in the range $p \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$ and $c \in \{0.9, 1.2, 2.4, 3.6\}$ on our test data, and found that the setting leading to highest recall@k is $p = 0.6$ and $c = 0.9$ (which effectively only runs the inner loop of MePo once). We also filter out ineligible premises and Lean language-related (e.g. metaprogramming) premises, similar to what we do in Section 3.2.1.
- Random forest (Piotrowski et al., 2023): this is a random forest model based on *features*, which are the collection of symbols appearing in the goal. We select their model with the highest reported performance, which is the model trained on data extracted with `n+b` features. We also modify their training to train on all Mathlib data (including our evaluation data), because it is nontrivial to modify their code to follow our data splits. This means that the training data given to their model includes our test/valid theorems. This gives their model an *unfair advantage*, so our observations are an upper bound on their model performance. We encountered non-terminating data extraction on a small number of modules, so we set a limit of 1,000 seconds for data extraction on each module. We also encountered time-out and out-of-memory issues during premise retrieval (some using >30GB RAM for a single retrieval), so we also increased LEANHAMMER time-out from

300 to 1,000 seconds. We did an additional pass to ensure we evaluate on as many theorems as possible. This still results in 300 premise retrieval errors out of 500 test theorems, so we only report the average over the successful ∼200 theorems in Table 2.

We do not test on their k-NN model because it is reported to be worse than the random forest model, and including our evaluation theorems in the training data gives k-NN significant unfair advantage.

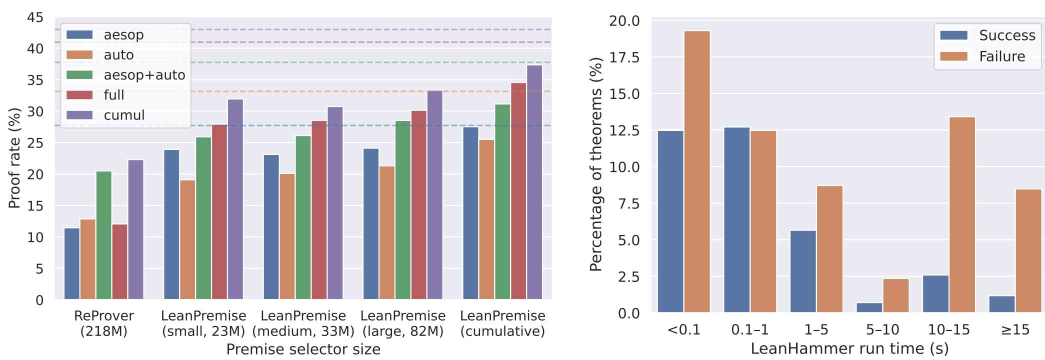
- ReProver (Yang et al., 2023): We run LeanDojo's data extraction script on our train/valid/test splits and Lean 4 version (version v4.16.0), and retrain their model on this data. We retrieve premises from accessible premises using their script. We note that our definition of ground-truth premises (premises used in the entire proof) is different from their definition (premises used in the next tactic), because we focus on finishing the goal and they focus on tactic generation, so there is discrepancy between recall@k in Table 2 and Yang et al. (2023). (We manually verified that the recall@10 value using their definition is similar to what they report, at about 38%, so our re-training worked properly.)

D Additional Results

D.1 Proof rate and run time

In Figure 2a, we show the performance of our models as the size of the premise selector scales on Mathlib-test. We note that using our models, performance scales well with model size.

In Figure 2b, we show the run time of LeanHammer on Mathlib-test using the `large` model, depending on if the theorem was proven. Most successful applications of LeanHammer run in under 1 second, but some theorems require a much longer time.



(a) Proof rate on Mathlib-test using different premise selectors. Dashed lines are ground-truth proof rates.

(b) LeanHammer (`full`) run time on Mathlib-test, depending on if the theorem was proven.

Figure 2: Analysis of proof rate and execution speed in different settings.

D.2 Extended miniCTX-v2 results

We present the complete results over all settings on `miniCTX`-v2 in Table 5. We note that over all settings, the ratio of our proof rate to the proof rate given ground-truth premises is largely preserved, or even increases for some settings, from Mathlib to `miniCTX`-v2. For the Carleson split, our pipeline resulted in an unexpected error which we believe is fixable, but in the meantime we put the proof rate as 0.0 (as a lower bound of performance).

D.3 Finding optimal k

In order to determine the number k of premises to retrieve and supply to LeanHammer, we perform a sweep of possible numbers (Figure 3) and determine that $k_1 = 16$ premises should be supplied to Lean-auto and $k_2 = 32$ premises should be supplied for premise application rules. Their

Premise selector	**Setting**	**Proof rate** (%)						
		Carleson	ConNF	FLT	Foundation	HepLean	Seymour	Average
None	`aesop`	0.0	10.0	27.3	38.0	8.0	6.0	14.9
LEANPREMISE (`large`)	`aesop`	0.0	16.0	39.4	38.0	10.0	20.0	20.6
Ground truth	`aesop`	2.4	16.0	30.3	40.0	12.0	22.0	20.4
None	`auto`	0.0	10.0	12.1	32.0	4.0	4.0	10.4
LEANPREMISE (`large`)	`auto`	0.0	10.0	15.2	32.0	4.0	10.0	11.9
Ground truth	`auto`	4.8	10.0	24.2	34.0	12.0	32.0	19.5
None	`aesop+auto`	0.0	10.0	27.3	38.0	8.0	6.0	14.9
LEANPREMISE (`large`)	`aesop+auto`	0.0	10.0	30.3	40.0	10.0	16.0	17.7
Ground truth	`aesop+auto`	4.8	10.0	36.4	40.0	18.0	30.0	23.2
None	`full`	0.0	10.0	27.3	38.0	8.0	6.0	14.9
LEANPREMISE (`large`)	`full`	0.0	16.0	36.4	38.0	10.0	24.0	20.7
Ground truth	`full`	7.1	16.0	39.4	40.0	20.0	34.0	26.1
None	`cumul`	0.0	10.0	27.3	38.0	8.0	6.0	14.9
LEANPREMISE (`large`)	`cumul`	0.0	16.0	39.4	40.0	12.0	26.0	22.2
Ground truth	`cumul`	7.1	16.0	39.4	40.0	20.0	38.0	26.8

Table 5: Extended table of performance of LEANHAMMER on each split of `miniCTX`-v2-test (Hu et al., 2025) using the `large` model trained on Mathlib.

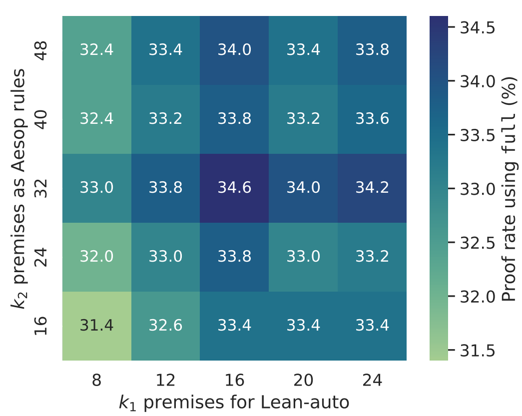
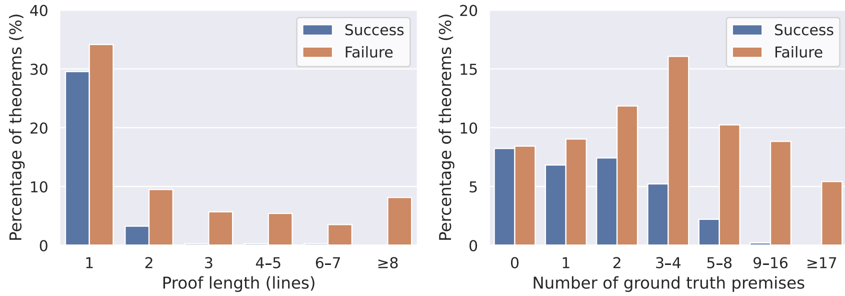


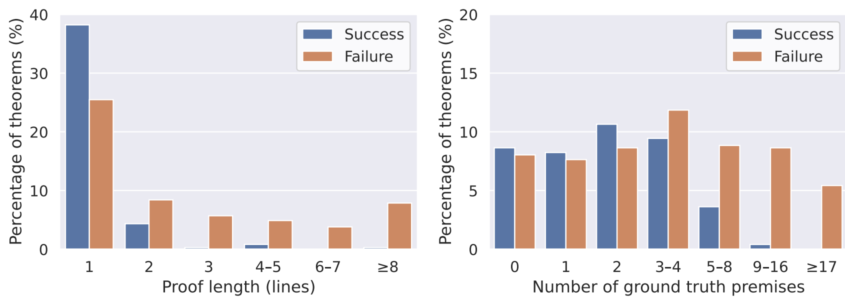
Figure 3: Proof rate on Mathlib-valid by number of retrieved premises under `full` setting.

respective Aesop priority values 10% and 20% are determined similarly, though their effect is much less than changing k, so we omit the details.

D.4 Analysis of proof rate by theorem difficulty



(a) Theorem statistics by difficulty using the `large` model.



(b) Theorem statistics by theorem difficulty using ground-truth premises.

Figure 4: Analysis of theorem statistics by difficulty, on Mathlib-test with the `full` setting, depending on if the theorem was proven.

For each theorem in Mathlib-test, we record the number of lines of the human-written proof uses[8] and the number of (filtered) premises used by the human-written proof, as proxy metrics of the difficulty of the theorem. Shown in Figure 4, we found that almost all theorems that LeanHammer solves, whether using our model or with ground-truth premises, use 1–2 lines in the human-written proof, while theorems not solved have a longer tail distribution. This means that, even if our premise selector were optimal, we would expect LeanHammer to primarily be useful for solving the final few lines or small gaps in a proof. Note that proofs in Mathlib are often "golfed", and 1–2-line proofs still have a wide range of difficulties. Similarly, the number of ground-truth premises of theorems that LeanHammer proves is usually no more than 8. These result imply that LeanHammer is good at filling in the small gaps in proofs, as the search space becomes prohibitively large for longer proofs.

D.5 Error analysis

The LeanHammer pipeline has multiple components, and each part may encounter an error during a proof attempt. In order to identify parts that may be improved in the future, we record the source of error leading to each unsuccessful proof, specifically in the `auto` setting (the part involving premise selection, Lean-auto translation, Zipperposition proof search, and Duper proof reconstruction), because the Aesop part is more established and well-understood (Limperg & From, 2023). When the ground truth premises (resp. premises retrieved by the `large` model) were supplied to LeanHammer, the results are as follows on Mathlib-test:

[8] Excluding proof headers such as `:= by` and `where`.

1. 21.7% (26.7%) of the theorems could not be translated by Lean-auto into the TH0 format used by Zipperposition (usually because the theorem itself or one of the premises supplied is outside the scope of the current translation procedure).
2. 43.6% (50.4%) of the theorems were translated by Lean-auto, but could not be proven by Zipperposition. This may be because necessary premises were not retrieved, Zipperposition was unable to perform the required form of reasoning (e.g. arithmetic or induction), or the translation did not preserve enough information (e.g. because the translation did not unfold some necessary constant).
3. 1.6% (1.2%) of the theorems were proven by Zipperposition, but its proof could not be reconstructed in Lean by Duper.
4. 0.0% (0.4%) of the theorems encountered another error.
5. The remaining 33.1% (21.3%) were successfully proven.

This shows that there may be improvements gained from (1) increasing recall@k of our premise selector, (2) improving translation of Lean into TH0, and (3) incorporating complementary tactics such as `grind` capable of solving problems not ideally suited to Duper, Zipperposition, or Aesop (e.g. problems involving arithmetic).