STP: Self-play LLM Theorem Provers with Iterative Conjecturing and Proving

Kefan Dong [1] **Tengyu Ma** [1]

Abstract

A fundamental challenge in formal theorem proving by LLMs is the lack of high-quality training data. Although reinforcement learning or expert iteration partially mitigates this issue by alternating between LLM generating proofs and finetuning them on correctly generated ones, performance quickly plateaus due to the scarcity of correct proofs (sparse rewards). To keep improving the models with limited data, we draw inspiration from mathematicians, who continuously develop new results, partly by proposing novel conjectures or exercises (which are often variants of known results) and attempting to solve them. We design the Self-play Theorem Prover (STP) that simultaneously takes on two roles, conjecturer and prover, each providing training signals to the other. The conjecturer is trained iteratively on previously generated conjectures that are barely provable by the current prover, which incentivizes it to generate increasingly challenging conjectures over time. The prover attempts to prove the conjectures with standard expert iteration. We evaluate STP with both Lean and Isabelle formal versifiers. With 51.3 billion tokens generated during the training in Lean, STP proves 28.5% of the statements in the LeanWorkbook dataset, doubling the previous best result of 13.1% achieved through expert iteration. The final model achieves state-of-the-art performance among whole-proof generation methods on miniF2F-test (65.0%), ProofNet-test (23.9%) and PutnamBench (8/644) with pass@3200.

1. Introduction

The reasoning capability of large language models (LLMs) is critical for various applications, including coding assistants, question-answering, and agents (Plaat et al., 2024; Shinn et al., 2023; Yao et al., 2022; Shao et al., 2024; Li et al., 2023; Nijkamp et al., 2022). It is also a key criterion for achieving artificial general intelligence (AGI). Automated theorem proving with formal languages by LLMs stands at the forefront of reasoning research (Yang et al., 2024a), partly because it allows objective and reliable evaluation through classical verifiers such as Lean (Moura & Ullrich, 2021) and Isabelle (Nipkow et al., 2002). Moreover, it arguably encapsulates the essence of advanced reasoning tasks while abstracting away the ambiguity of natural language, enabling meaningful studies on a relatively smaller scale.

However, a fundamental challenge in improving reasoning performance—whether in natural or formal languages—lies in the lack of high-quality training data. Collecting reasoning data requires domain experts, making it expensive to scale. There are only a limited number of advanced math papers and theorems in existence, orders of magnitude smaller than other data sources.

Reinforcement learning (RL) on datasets *without* solutions (e.g., datasets with theorem statements or reasoning questions and answers) is a prominent approach for improving the reasoning capability, as seen in the recent development of OpenAI o1 (Jaech et al., 2024), DeepSeek-Prover (Xin et al., 2024a) and DeepSeek R1 (Guo et al., 2025). Often referred to as expert iteration (Anthony et al., 2017), it partially mitigates the data scarcity issue by alternating between LLMs generating proofs and finetuning them on correctly generated ones (Kaliszyk et al., 2018; Wu et al., 2021; AlphaProof, 2024; Xin et al., 2024b; Ying et al., 2024).

However, as Wu et al. (2024) pointed out, RL or expert iteration often saturates at a low pass rate because the number of samples required to generate a correct proof for an unproven theorem grows exponentially. As a result, a massive amount of computation is wasted on generating incorrect proofs that provide no training signal to the model. For instance, in the proof sampling process of Wu et al. (2024), 98.5% of the compute yields no successful proofs, despite the pass rate being only 13.1% on the training dataset, LeanWorkbook (Ying et al., 2024). In other words, after a few rounds of expert iteration, re-training the model becomes less effective due to the limited number of new successful proofs.

In addition, RL's capability is fundamentally bounded by the difficulty level of the theorems in the training dataset—it

[1]Stanford University. Correspondence to: Kefan Dong <kefandong@cs.stanford.edu>, Tengyu Ma <tengyuma@stanford.edu>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

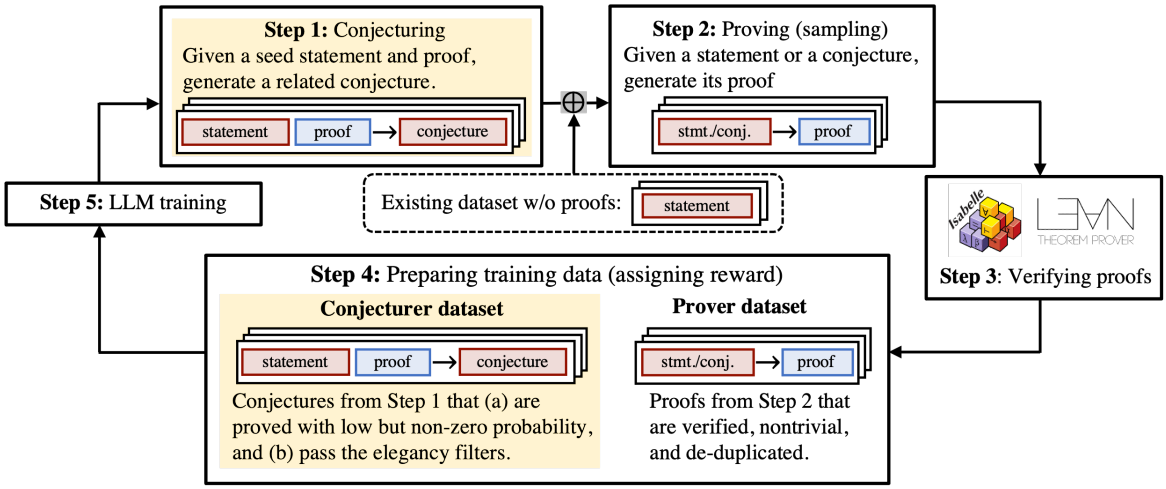

Figure 1: Self-play Theorem Prover (STP). Our model simultaneously takes on two roles — the conjecturer that generates new, related conjecture given a seed theorem with proof (Step 1), and the prover that attempts to prove the statements in an existing dataset and the generated conjectures (Step 2). Step 4 selects the correct, approachable, elegant, yet challenging conjectures to train the conjecturer, and the verifier selects correct proofs in Step 3 to train the prover. The main difference between STP and expert iteration is the conjecturer role highlighted with a yellow background.

is unlikely, in principle, for a model to learn college-level proof techniques solely by working on high school-level problems or to solve open math problems using RL on graduate-level problems. Moreover, there are likely not enough open problem statements available for RL training to generalize to other open problems, particularly more advanced ones. In other words, RL or expert iteration algorithms cannot train indefinitely without continuously collecting more theorem statements or math problems.

We need an algorithm that can run and self-improve indefinitely *without more data*. To this end, we draw inspiration from how mathematicians learn and develop advanced mathematics; they refine their understanding and sharpen their proof skills by working on synthesized exercises—variants, extensions, or combinations of existing theorems. Additionally, they frequently propose and publish conjectures, a process widely regarded as just as important, if not more so, than solving them. In other words, unlike the current training of LLMs, mathematicians engage with far more exercises and conjectures (referred to collectively as conjectures in this paper) than the polished, published results found in academic papers and books. Moreover, the continuous generation of new conjectures keeps mathematical fields dynamic and moving forward.

In this paper, we design Self-play Theorem Prover (STP), which mimics how mathematicians learn and develop mathematics. It simultaneously assumes two roles—conjecturer and prover—providing training signals to each other.

As illustrated in Fig. 1, the conjecturer, given a seed theorem with proof, proposes a new, related conjecture (Step 1), while the prover attempts to prove conjectures and statements from an existing dataset (Step 2). Then, the verifier selects correct proofs (Step 3) to train the prover using standard RL and identifies correct, approachable, elegant, yet challenging conjectures to supervise the training of the conjecturer (Step 4). More concretely, in each iteration, the conjecturer is trained on previously generated conjectures that: (a) are barely provable by the current prover (i.e., the prover's success probability with respect to its random seed is positive but low), and (b) pass certain elegancy filters. This iterative process gradually increases the difficulty of conjectures and proofs without requiring additional data. Our method can be viewed either as a self-play algorithm between conjectures and provers or as automated curriculum learning (Portelas et al., 2020) with a self-generated adaptive curriculum (via conjecturers).

We empirically evaluate our method with both Lean (Moura & Ullrich, 2021) and Isabelle (Nipkow et al., 2002). For the Lean experiments, we aim for the best performance and therefore choose DeepSeek-Prover-V1.5-SFT (Xin et al., 2024b) as the base model for STP. As shown in Fig. 2, after a self-play training of roughly 241M generated proofs and 3.6M generated conjectures, we successfully prove 28.5% of the statements in the training dataset LeanWorkbook (Ying et al., 2024), doubling the previous best result of 13.1% (Wu et al., 2024) achieved by expert iteration.[1] In Fig. 3, we compare the inference-time performance of existing

[1] Based on our reproduction, Deepseek-Prover-V1.5 has a better performance on LeanWorkbook, proving 13.2% statements with 32 samples per statement, although the original paper does not report this number.

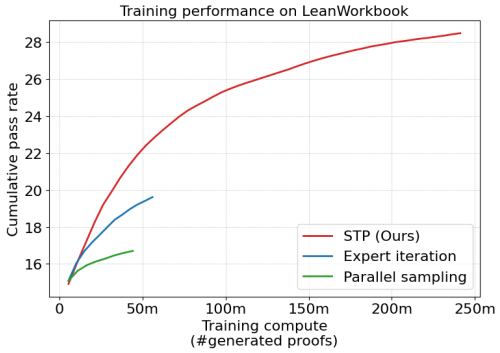

Figure 2: The cumulative pass rates of STP, expert iteration, and parallel sampling on LeanWorkbook shows that STP achieves a much better scaling in terms of the performance vs number of generated proofs. The compute for generating conjectures and training the conjecturer in STP is negligible because the number of generated proofs during training is 64 times the number of conjectures.

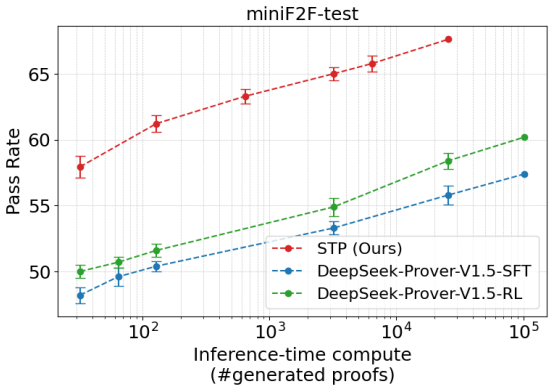

Figure 3: Comparison of pass rates on miniF2F-test (y-axis) with different numbers of inference-time samples (x-axis). The model trained with STP consistently outperforms the Deepseek-Prover-V1.5 series.

models and the final model trained with STP by taking multiple independent samples on a common benchmark, miniF2F-test (Zheng et al., 2021). Our model significantly outperforms the DeepSeek-Prover-V1.5 models across various sampling budgets. We also achieve state-of-the-art performance among whole-proof generation methods on miniF2F-test (65.0%, pass@3200), ProofNet-test (23.9%, pass@3200) (Azerbayev et al., 2023a) and PutnamBench (8/644, pass@3200) (Tsoukalas et al., 2024), where pass@k represents the percentage of statements proved with k independently sampled proofs per statement.

In the Isabelle experiments, we study the scalability of STP by starting from a generic math-focused model Llemma-7b (Azerbayev et al., 2023b) and run STP for more iterations (300M generated proofs in total). We compare the scaling of STP with expert iteration and parallel sampling, by taking several model checkpoints during the STP training run and then switching to the baseline methods. The results clearly demonstrate that STP achieves a better scaling behavior starting from various checkpoints with different capability (see Fig. 4 in Section 4.3). Ablation study also demonstrates that the main performance gain stems from the dense training signals given by the conjectures. Expert iteration wasted its compute on generating unsuccessful proofs to challenging theorems in the training dataset—at a checkpoint where the pass rate is around 11.4% on LeanWorkbook, only 131 out of 2.5M generated proofs of the unproved statements are correct, resulting in very limited training signals. In contrast, at least 47% of the generated conjectures in STP training are successfully proved because the conjecturer is trained to generate more approachable statements thanks to the design of its reward (see Fig. 6).

2. Additional Related Works

We refer the readers to Bibel (2013); Loveland (2016) and the reference therein for classical automated theorem proving. Below, we discuss recent works on modern LLM-based theorem provers in addition to what has been discussed in the intro.

Autoformalization. A relatively efficient way to create formal proof data is autoformalization, that is, translating natural language math statements and/or proofs to formal language (Jiang et al., 2023; Lu et al., 2024). A line of research focuses on generating proofs or reasoning steps in natural language and then formalizing the proofs (Jiang et al., 2022b; Zheng et al., 2023; Wang et al., 2023). Most recently, AlphaProof (2024); Xin et al. (2024a;b) autoformalize statements and then train with expert iteration / RL to write proofs, achieving significant improvement over prior works thanks to the large-scale natural language datasets.

Formal conjecturing. Prior works also study how to generate new formal statements/conjectures by neural networks (Urban & Jakubův, 2020; Einarsdóttir et al., 2024; Johansson & Smallbone, 2023) or human-written generators (Polu et al., 2022; Trinh & Luong, 2024), and find that the synthetic statements are generally useful for training the provers (Wang & Deng, 2020; Wu et al., 2020). Synthetic statements and proofs can also be extracted from an incorrect proof trajectory during RL with hindsight experience replay (HER) (Andrychowicz et al., 2017) to speed up the training process (Aygün et al., 2022; Dong et al., 2024). However, even though the training efficiency is improved, we argue that the final performance is still bounded by difficulty level of the existing dataset because synthetic statements are most likely easier than the given ones in the dataset.

Self-play and automatic goal generation. The closest re-

lated work to this paper is Poesia et al. (2024) which also designs a self-play training that iterates between conjecturing and theorem proving. The key difference between this paper and Poesia et al. (2024) is that we start with a pretrained model and work on practical formal languages like Lean and Isabelle with an infinite space of possible proof steps (which are actions in the RL algorithm), whereas Poesia et al. (2024) operates in a simplified and constrained setting with a finite action space and trains from scratch. As a result, Poesia et al. (2024) rely on constrained decoding to force the validity of generated conjectures, while we solely rely on the LLM itself to generate valid conjectures. Technically, since our training process is much longer (more than 50 iterations) than Poesia et al. (2024) (5 iterations), we must carefully design the conjecturing reward to maintain the diversity and relevance of the generated conjectures (see Section 3.2).

The idea of generating new tasks by the model is also explored in other domains such as alignment (Ye et al., 2024), programming puzzles (Haluptzok et al., 2022; Teodorescu et al., 2023; Pourcel et al., 2024b), video games (Zhang et al., 2023; Pourcel et al., 2024a), and classic RL environments (Parker-Holder et al., 2022; Colas et al., 2022). More generally, self-play training has demonstrated its potential to achieve super-human performance on two-player games in a fixed environment like Go (Silver et al., 2016).

3. Method

On the high level, Self-play Theorem Prover (STP) involves three training stages: (1) model initialization by supervised finetuning, (2) self-play training (visualized in Fig. 1), and (3) final re-training. Unless otherwise stated, we use the term 'statement' to refer to the statements in given datasets, and 'conjecture' the generated conjectures.

3.1. Model initialization by supervised finetuning

In this stage, we initialize the model with two roles, conjecturer and prover, by finetuning a generic LLM (such as the Llama (Touvron et al., 2023)) on a SFT dataset constructed from existing proof libraries such as Mathlib (mathlib Community, 2020). The proof libraries are organized into files containing human-written formal proofs of known mathematical theorems, and each file formalizes a relatively self-contained result, such as a chapter of a textbook. Our SFT data consists of the following two parts, for finetuning the prover and conjecturer, respectively. Also see concrete examples in Appendix A.1.

Prover SFT dataset. We construct a SFT dataset to teach the model to write formal proofs in the given format, where each example is the concatenation of a system prompt (to instruct the model to generate in formal language), a statement, and its corresponding proof. We only compute the next token prediction loss on the proof (which is the expected output of the model), while the rest is treated as input. To build this dataset, we simply extract all the statement-proof pairs from the proof library files and add a system prompt.

Conjecturer SFT dataset. Generally, the conjecturer is to generate a new, related conjecture, given a seed statement with proof that provide the initial ideas. Technically, to further guide the generation of conjecturer, we also provide it a lemma used in the proof of the seed statement, which can be extracted from the verifier,[2] so that the generated conjectures are more likely to be related to the theorem through the lemma. Therefore, the input is a concatenation of the system prompt, a lemma, and a seed statement and its proof, separated by special formatting tokens, and the expected output is a conjecture on which we compute training loss. We also allow the model to generate conjectures with a fixed trivial lemma. To construct this dataset, we extract (lemma, theorem X, theorem Y) tuples from every proof library file such that (a) the lemma and two theorems appears in the file in this particular order, and (b) the lemma is used in the proof of both theorems. The lemma and theorem X will be part of the inputs, and theorem Y will be the output.

3.2. Self-play training

Our self-play training stage of STP is shown in Fig. 1. The main difference compared to expert iteration is the conjecturer in Steps 1 and 4, highlighted in a yellow background.

Generating conjectures and proofs (Steps 1 & 2). The self-play training starts with collecting a list of the conjecturer's inputs in the same format as in the conjecture SFT dataset (system prompt, lemma, and theorem), but from theorem-proof pairs where the theorems are from the given dataset without proofs and proofs are previously generated. We extract a seed lemma from the proof, using the verifier.[3] To prevent the model from only focusing on a few particular proof techniques, we de-duplicate the list based on the seed statement and lemma, and randomly drop some inputs whose lemma appears excessively. Then, the LLM generates conjectures from the inputs, and we randomly select a subset of the generated conjectures with size no larger than the number of remaining unproved statements in the given dataset, so that the prover's compute budget is split equally between the conjectures and statements. (See the pseudocode and details in Appendix A.2.) For the prover's inputs, we combine the generated conjectures and the unproved statements in the existing dataset. Then, we independently sample K proofs per statement/conjecture in Step 2.

[2] There is no fundamental difference between lemmas and theorems in formal proofs — the naming is purely for better exposition.

[3] In our implementation, lemmas are extracted together with proof verification in Step 3 by configuring the verifiers accordingly.

Reward assignments (Step 4). The major technical challenge of STP is to design the reward function for the conjecturer (in other words, construct the conjecturer dataset in Step 4). The ultimate goal is to incentivize conjecturer to generate diverse, relevant, approachable yet challenging conjectures to provide enough training signals to the prover.

In Step 4, we first organize all generated conjectures and proofs into a list of examples $\mathcal{D} = \{(t_i, p_i^t, l_i, c_i, p_i^c)\}_{i=1}^n$ where t_i and p_i^t represents a seed statement and its proof, l_i is a lemma used in the proof p_i^t, and c_i, p_i^c are the generated conjectures and the generated proof. We will filter $\mathcal{D}$ as described below and then use (t_i, p_i^t, l_i) as the input to the conjecturer and c_i as the output, and p_i^c as the output of the prover w.r.t. the input c_i.

To decide whether a conjecture c is challenging, we use the (empirical) pass rate of the prover estimated by the K independently generated proofs:

$$\hat{P}(c) \triangleq (\#\{i : c_i = c, p_i^c \text{ is correct}\})/(\#\{i : c_i = c\}).$$

Then, we select the examples in $\mathcal{D}$ where (a) lemma l_i is used in the proof of conjecture p_i^c, and (b) the pass rate of the conjecture, $\hat{P}(c_i)$, is between $(0, 1/4]$:

$$\overline{\mathcal{D}} \leftarrow \{(t_i, p_i^t, l_i, c_i) \mid (t_i, p_i^t, l_i, c_i, p_i^c) \in \mathcal{D}, \\ \hat{P}(c_i) \in (0, 1/4], p_i^c \text{ is correct}, l_i \text{ is used in } p_i^c\}.$$

Here we discard the proofs (of the conjecture) p_i^c since they are not needed to train the conjecturer, and we remove the duplicated conjectures (that have multiple proofs).

Then, we apply a heuristic elegancy filter to discourage the model from generating artificially hard conjectures with complicated goals — we remove conjectures whose minimum proof length divided by the length of the conjecture is in the lowest 20% of remaining examples.

Finally, we re-weight the selected conjectures to preserve the diversity of the conjecturer — the reward for conjecturer cannot only depend on the generated conjectures individually because otherwise the conjecturer's optimal policy may degenerate to a singular distribution, whereas in reality, the given dataset typically has multiple modes because the statements focus on different topics like algebra, number theory, and calculus. Therefore, our idea is to push the *distribution* of the selected conjectures toward the unproved statements in the existing dataset to maintain the balance between multiple modes. To this end, we compute a distribution P supported on the selected conjectures that minimizes the Wasserstein distance to the uniform distribution over unproved theorems, denoted by Q. The matching cost or similarity metric between a conjecture and a statement, used for computing the Wasserstein distance between P and Q, is defined as the negative cosine similarity between their embeddings (given by the current model). Finally, we use the distribution P as the training set for the conjecturer. Pseudo-code of this step is in Appendix A.3, and an efficient implementation is in Appendix A.5.

For the prover dataset, we only select correct generated proofs where the empirical pass rate of the corresponding statement/conjecture is below 1/2. (We consider other correct proofs trivial). We de-duplicate the prover dataset by exact match. Then, the prover is trained on a replay buffer containing the selected proofs from the last three iterations.

LLM training (Step 5). We use weighted cross entropy loss computed on the conjectures or proofs (but not the inputs of the model). For the proof dataset, we weight the examples reciprocally to the number of verified proofs to the corresponding statement/conjecture. We also use a length penalization of the form γ^L to reward simpler proofs, where $\gamma < 1$ is the discount factor and L is the length of the proof. For the experiments with Lean, we additionally reward proofs that has faster verification time by a penalization of the form β^T, where T is the execution time of the Lean verifier.[4]

3.3. Final re-training

To avoid training instability caused by the changing data distribution during self-play, we re-train the final model checkpoint from the base model (before the SFT stage) on a combination of the SFT dataset and all the correct proofs generated during the self-play training whose corresponding statement/conjecture has an empirical pass rate no larger than 1/4. For every statement/conjecture, we randomly keep at most 16 distinct proofs to speedup the training.

4. Experiments

This section presents our implementation details of STP, the results of Isabelle and Lean experiments, and the ablation studies, followed by examples of generated conjectures.

4.1. Implementation details

Training datasets. Our primary source of statements without proofs is the de-duplicated LeanWorkbook (Ying et al., 2024), which contains around 89K Lean4 statements (see Appendix A.4 for details). For the Isabelle experiments, we translate the Lean4 statements to Isabelle using the DeepSeek V2.5 with few-shot prompting. For the Lean experiments, we combine LeanWorkbook, miniF2F-valid, and ProofNet-valid as the training dataset for STP.

The SFT dataset for the Isabelle experiments is extracted

[4] In our preliminary experiments, we found that without the penalization on verification time, the Lean verifier takes 2x more wall-clock time on CPU than sampling proofs on TPU for our cluster setup, which becomes a bottleneck for STP training.

from AFP[5] and Isabelle built-in files such as HOL. For the Lean experiments, we first sample 32 proofs per statement in LeanWorkbook since our base model, DeepSeek-Prover-V1.5-SFT, is already trained on it, and combine the correct proofs with examples extracted from the proof library Mathlib4 (mathlib Community, 2020) as the SFT dataset.

Periodic refreshing. With a limited replay buffer, the model may forget some proof skills learned in the SFT stage after many iterations. Therefore, during our STP training, we periodically re-train the model from the base model on all previously generated correct proofs, following a procedure similar to the final re-training in Section 3.3. After refreshing, we reset the replay buffer and restart the self-play training using the re-trained model checkpoint.

Verifiers' setup. To study the scalability of STP with limited compute, in the Isabelle experiments, we disable the advanced proof tactics `sledgehammer, mason, smt, metis, sos`, which require huge CPU compute, to allow more training iterations, sacrificing verification strength and overall performance. We use PISA (Jiang et al., 2021) to interact with Isabelle, and enforce a 10s timeout for any proof step and 360s timeout for entire proofs. For Lean, we follow Xin et al. (2024b), which allows all proof tactics, and set a 200s timeout and a 15GB memory limit for each proof.

Hyperparameters. For inference, we cap the number of generated tokens to 1024, and set the sampling temperature to 0.7 for Llemma-7b and 1.0 for DeepSeek-Prover, following Dong et al. (2024); Xin et al. (2024b), respectively. For training, we use batch size 2048 and Adam (Kingma & Ba, 2014) with a constant learning rate of 5e-5 in STP, and 1e-4 in SFT and final re-training. The discount factors are $\gamma = \exp(-0.001)$ and $\beta = \exp(-0.01)$

In each iteration of STP, we sample $K = 32$ proofs per conjecture/statement. For the expert iteration and parallel sampling, we use $K = 64$. Since we maintain the number of generated conjectures per iteration to be at most the number of unproved statements in the given dataset, STP has the same sample budget as the baseline methods per iteration.

4.2. Results with Lean

For the Lean experiments, we choose DeepSeek-Prover-V1.5-SFT as our base model, which is trained on proofs collected by expert iteration on a combination of public, such as LeanWorkbook, miniF2F-valid (Zheng et al., 2021), and ProofNet-valid (Azerbayev et al., 2023a), and proprietary datasets. We run 48 iterations of STP and generated 3.6M conjectures, 241M proofs, and 51.3B tokens in total. We use the cumulative pass rate, defined by the fraction of statements proved during the entire training, as the main metric for training progress. Fig. 2 plots the cumulative pass rate of STP and two major baselines, expert iteration, and parallel sampling, on the training dataset LeanWorkbook (Ying et al., 2024). Expert iteration alternates between generating proofs to the statements in the given dataset and finetuning the model on correct proofs. (See discussions and comparison about variants of expert iteration in Appendix A.6.) Parallel sampling simply generates proofs with the given model. Fig. 2 shows that STP achieves significantly better scaling than expert iteration, which simulates the performance of DeepSeek's model as if it were trained for more iterations.

Since the formal statements in our training dataset, LeanWorkbook, are translated from natural language statements, they are not always provable. In Appendix B.3, we randomly select 20 unproved statements from LeanWorkbook and manually assess whether (a) the formal statement is an accurate translation of the natural language statement, and (b) the formal statement itself is correct and provable. We find that 16 out of the 20 statements are translated correctly, but only 7 statements are provable and the remaining 13 statements are unprovable (e.g., due to missing assumptions in the corresponding natural language statement), suggesting that the best possible pass rate on LeanWorkbook is far below 100%.

In Table 1, we compare the final re-trained model of STP with prior works on two common benchmarks, miniF2F-test and ProofNet-test, which contain formal statements of high-school level and college level math questions, respectively.[6] Among the whole-proof generation methods, STP significantly outperforms DeepSeek-Prover-V1.5-RL (which is continuously trained with RL on top of their SFT model) and achieves SoTA performance across various inference-time sample budgets. We also report the performance of the model trained only on LeanWorkbook for 24 iterations, excluding miniF2F-valid and proofnet-valid, demonstrating that the model trained with STP also generalizes to out-of-domain theorems.[7]

Table 1 also compares STP with tree search methods such as InternLM2.5-StepProver (Wu et al., 2024), which use LLMs to generate single proof steps conditioned on the current verifier's proof state and then find a complete proof by best first search or MCTS. The sample budget of these methods are not directly comparable with whole-proof generation meth-

[5] https://www.isa-afp.org/

[6] In our experiments, we use the miniF2F statements in https://github.com/yangky11/miniF2F-lean4/commit/8c83bc4 and the ProofNet statements in https://github.com/deepseek-ai/DeepSeek-Prover-V1.5/blob/main/datasets/proofnet.jsonl.

[7] Our base model, DeepSeek-Prover-V1.5-SFT, is trained on miniF2F-valid and ProofNet-valid, though we only run STP on LeanWorkbook in this experiment. The penalization on verification time is also not included in this experiment.

Table 1: Pass rate on miniF2F (Zheng et al., 2021) and ProofNet (Azerbayev et al., 2023a) with different inference-time sample budgets. Our method, STP, achieves state-of-the-art performance among whole-proof generation methods across various sample budgets. For reference, we also include tree search methods, even though they are orthogonal to our main contribution. The sample budgets of tree search methods are not fully comparable to that of the whole proof generation because they also use the LLM to process the verifier's internal proof state.

Method	Sample budget (#Proofs)	Sample budget (#Steps)	MiniF2F-test	ProofNet-test
Whole-Proof Generation Methods				
TheoremLlama (Wang et al., 2024)	128	-	33.6%	-
DSP (Jiang et al., 2022b)	100	-	39.3%	-
DeepSeek-Prover-V1.5-SFT	128	-	50.4% ± 0.4%	15.9% ± 0.6%
(Xin et al., 2024b)	3200	-	53.3% ± 0.5%	21.0% ± 0.9%
DeepSeek-Prover-V1.5-RL	128	-	51.6% ± 0.5%	18.2% ± 0.5%
(Xin et al., 2024b)	3200	-	54.9% ± 0.7%	22.0% ± 0.5%
	25,600	-	58.4% ± 0.6%	23.7%
	102,400	-	60.2%	-
STP	128	1.1K	57.2% ± 0.6%	18.0% ± 0.7%
(w/o miniF2F-valid, ProofNet-valid)	3200	28K	61.1%	23.1%
STP	128	1.3K	**61.2% ± 0.6%**	**19.5% ± 0.7%**
	3200	32K	**65.0% ± 0.5%**	**23.9% ± 0.6%**
	25,600	254K	**67.6%**	**26.9%**
Tree Search Methods[8]				
ReProver (Yang et al., 2024b)	-	-	26.5%	-
PACT (Zheng et al., 2021)	-	$8 \times 16 \times 512 = 66$K	29.2%	-
GPT-f (Polu et al., 2022)	-	$64 \times 8 \times 512 = 262$K	36.6%	-
HTPS (Lample et al., 2022)	-	$64 \times 5000 = 320$K	41.0%	-
Lean-STaR (Lin et al., 2024)	-	$64 \times 1 \times 50 = 3.2$K	46.3%	-
DeepSeek-Prover-V1.5-RL + RMaxTS[9]	3200	-	55.0% ± 0.7%	21.5% ± 0.8%
(Xin et al., 2024b)	25,600	-	59.6% ± 0.6%	25.3%
	204,800	-	63.5%	-
InternLM2.5-StepProver	-	$4 \times 32 \times 600 = 77$K	58.5% ± 0.9%	-
(Wu et al., 2024)	-	$16 \times 32 \times 600 = 307$K	62.5% ± 0.5%	-
	-	$256 \times 32 \times 600 = 4.9$M	65.9%	-

ods because (a) the number of steps in a generated proof varies significantly, (b) LLMs in tree search methods need to process additional tokens related to the verifier's proof state, and (c) methods like InternLM2.5-StepProver (Wu et al., 2024) require an additional LLM as the value function. Moreover, it's conceivable that tree search methods can also be used with STP, so essentially these are orthogonal methods. Nonetheless, we compute the total number of proof steps per statement generated by STP as an proxy for the total number of LLM output tokens for STP and tree search methods, ignoring the additional compute required by tree search methods to process the verifier's proof state and query the value function. Results in Table 1 indicate that STP also outperforms prior tree search methods with similar (estimated) inference-time budgets.

As shown in Table 4, on PutnamBench (Tsoukalas et al., 2024) which consists of undergraduate-level mathematics competition questions, STP solves 7 out of 644 problems with 128 samples per problem, and 8 problems with 3200 samples per problem, outperforming the best result of 6 problems in prior works achieved by Wu et al. (2024).

4.3. Results with Isabelle

For Isabelle experiments, we start with the Llemma-7b (Azerbayev et al., 2023b), math-focused model, and run 58 iterations of STP to study its scalability. We take several checkpoints during STP training and then switch to the expert iteration and parallel sampling baselines to study the scalability of the algorithm from checkpoints with various capability. Fig. 4 compares their cumulative pass rates on LeanWorkbook (Isabelle translation), showing that STP consistently achieves a better scaling across the training

[8]The #Steps for tree search methods is typically calculated by #Independent runs × #Tactics generated per search step × #Search steps, or #Independent runs × #Search steps.

[9]DeepSeek-Prover-V1.5-RL + RMaxTS is a tree search method that uses the LLMs to generate complete proofs during the search instead of single proof steps. Therefore, we treat their sample budget as the number of generated proofs instead of steps.

Table 2: Comparison of STP with and without final re-training on miniF2F-test and ProofNet-test.

Method	Budget	miniF2F	ProofNet
STP	1	42.3%	7.7%
w/ re-training	3200	65.0%	23.9%
	25600	67.6%	26.9%
STP	1	43.2%	8.8%
w/o re-training	3200	65.1%	22.0%
	25600	66.0%	23.7%

process. The model also gradually improves on miniF2F over the training process, as shown in Fig. 5.

4.4. Ablation study

Generated conjectures provide denser training signals. Fig. 6 shows the histogram of empirical pass rates of the generated conjectures and the unproved statements in LeanWorkbook using a checkpoint in the Isabelle experiment. Only 131 out of 2.5M generated proofs for the 79K unproved statements are correct. As a result, finetuning the model on correct proofs has almost no effect, and thus expert iteration plateaus. In contrast, generated conjectures by STP offers higher pass rates and thus more training signals, leading to better scaling.

Re-training with generated conjectures still helps downstream performance. One may hypothesis that the self-play algorithm and generated conjectures only help improve the pass rate on LeanWorkbook. It turns out that in the final re-training stage, it is still beneficial to re-train with the generated conjectures in addition to the successfully proved statements in LeanWorkbook even for performance on miniF2F-test and ProofNet-test—it leads to about 2-3% performance gain (for pass@128) than re-training only on the latter (See Appendx B.1).

Final re-training improves the pass@k performance. In Table 2, we compare the performance of STP with the final model checkpoint from self-play training (i.e., without final re-training) on miniF2F-test and ProofNet-test. Table 2 shows that STP with final re-training achieves a better scaling (and therefore a better pass@k for large k), while sacrificing the pass@1 performance.

4.5. Analysis of generated conjectures

We observe that the generated conjectures generally require similar proof techniques as the original statements — we compute the 20 most frequently used lemmas in the proofs of generated conjectures, and find that among these 20 lemmas, 17 also appear among the top 20 most frequent

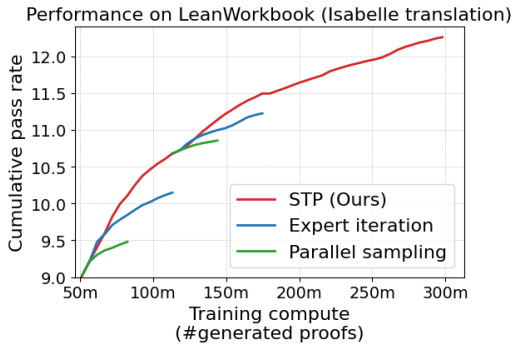

Figure 4: Cumulative pass rate on LeanWorkbook (translated into Isabelle) of STP, expert iteration, and parallel sampling, started from two checkpoints in STP training. STP achieves better scaling starting from both checkpoints. For better visualization, the x-axis starts with 50m here and we defer the full plot to Fig. 7 in the appendix.

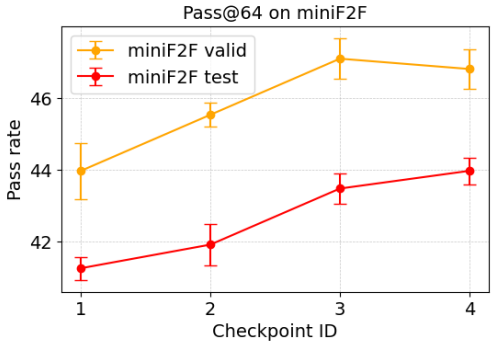

Figure 5: The performance of our model on miniF2F gradually improves during the training process. Note that our model is not trained on miniF2F valid and we disallow advanced tactics such as `sos`. The checkpoints are taken roughly per 68M generated proofs.

lemmas in the proofs of LeanWorkbook statements. At a late checkpoint of our experiments, 51.1% of the correct proofs of generated conjectures share at least one lemma with the proof of the seed statement, which indicates that a substantial portion of generated conjectures are meaningfully related to their seed statement. In Appendix B.4, we list 30 most frequently shared lemmas. While lemmas in the field of inequalities and algebra remain dominant (e.g., `sq_nonneg`, `Real.sin_sq_add_cos_sq`), we also see lemmas related to sequence of products (i.e., `Finset.prod_range_succ`), and number theory (e.g., `Nat.pow_mod`, `nat_sub_dvd_pow_sub_pow`).

4.6. Examples of generated conjectures

In this section, we list three manually selected examples of the generated conjectures at the last iteration of the Lean

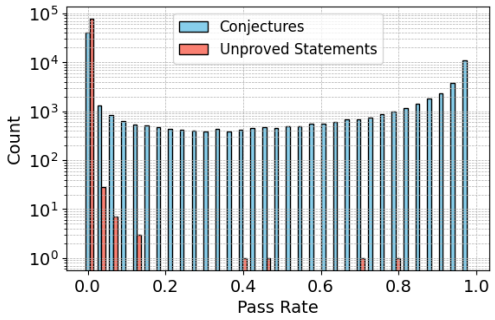

Figure 6: Histogram of empirical pass rates of generated conjectures and unproved statements in the training dataset at a checkpoint where the cumulative pass rate on LeanWorkbook (Isabelle translation) is 11.4%. The generated conjectures are significantly more likely to be proved (i.e., has a positive pass rate) than the unproved statements in the dataset, and therefore provide denser training signal. Note that the y-axis is in log scale.

experiment to demonstrate their quality.

Example 1. The generated conjecture says $(1+x)^{2n} \geq 1+x^n$ when $n \geq 1$ is an integer and $x \in [0,1]$:

```
theorem lean_workbook_36081' (x : ℝ) (hx :
    0 ≤ x ∧ x ≤ 1) : ∀ n :ℕ, n ≥ 1 → (1 +
     x)^(2*n) ≥ 1 + x^n
```

The seed statement says $1+x^2 \leq (1+x)^2$ when $x \in [0,1]$:

```
theorem lean_workbook_36081 (x : ℝ) (hx :
    0 ≤ x ∧ x ≤ 1) : 1 + x^2 ≤ (1 + x)^2
```

In this case, the conjecture is harder than the original statement but is proved with similar techniques — expanding the powers of a binomial and then using the fact that $x \geq 0$.

Example 2. The generated conjecture says $(x^n - 1) \bmod (x-1) \leq 1$ if x, n are integers:

```
theorem lean_workbook_54038' (x : ℕ) (n : ℕ
    ) (hn : 1 < n) : (x^n - 1) % (x - 1) ≤
    1
```

The seed statement says $n-1$ divides $n^k - 1$:

```
theorem lean_workbook_54038 (n : ℕ) (k : ℕ
    ) (hn : 2 ≤ n) : n - 1 ∣ n^k - 1
```

In this case, our model generates a variant of the original statement by realizing that $b \bmod a$ equals zero if a divides b. This conjecture may help the model connect its proof technique in algebra and number theory. However, the conjecture itself is somewhat unusual and the inequality is not tight. Therefore it is unlikely to be included in any datasets.

Example 3. The generated conjecture says $\sum_{i\geq 0}((1/4)^i \cdot a) = \frac{a}{1-1/4}$ if $0 < a \leq 1$.

```
theorem lean_workbook_plus_46203' (a : ℝ)
    (ha : 0 < a ∧ a ≤ 1) :  Σ' (i : ℕ), 1
    / 4 ^ i * a = a / (1 - 1 / 4)
```

The seed statement is a special case where $a = \sqrt{5}/3$:

```
theorem lean_workbook_plus_46203 :
   Σ' k : ℕ, (1 / 4)^k * (Real.sqrt 5 / 4)
    = (Real.sqrt 5 / 3)
```

In this case, the conjecture generalizes the given statement by replacing `Real.sqrt 5 / 4` with a variable `a`.

5. Conclusion

This paper designs Self-play Theorem Prover (STP) that simultaneously has two roles, conjecturer and prover. By providing training signals to each other, STP goes beyond the statements in the given dataset and its performance continuously improves. Our final model significantly outperforms Deepseek-Prover-V1.5 series and achieves state-of-the-art performance among whole-proof generation methods on common formal proof benchmarks. To facilitate future research, we release our code, model, and dataset in this url: https://github.com/kfdong/STP.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

Acknowledgment

The authors would like to thank Yinuo Ren, Zhizhou Ren, Woosuk Kwon, David Hall, Huajian Xin, Kaiyue Wen, and ICML anonymous reviewers for their helpful discussions. The authors would also like to thank the support from NSF RI 2211780, and NSF CIF 2212263, and the Google TPU Research Cloud for the computing resources that enabled these experiments.

References

AlphaProof. Ai achieves silver-medal standard solving international mathematical olympiad problems. 2024. URL https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Pieter Abbeel, O., and Zaremba, W. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

Anthony, T., Tian, Z., and Barber, D. Thinking fast and slow

with deep learning and tree search. *Advances in neural information processing systems*, 30, 2017.

Aygün, E., Anand, A., Orseau, L., Glorot, X., Mcaleer, S. M., Firoiu, V., Zhang, L. M., Precup, D., and Mourad, S. Proving theorems using incremental learning and hindsight experience replay. In *International Conference on Machine Learning*, pp. 1198–1210. PMLR, 2022.

Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., and Avigad, J. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023a.

Azerbayev, Z., Schoelkopf, H., Paster, K., Dos Santos, M., McAleer, S., Jiang, A., Deng, J., Biderman, S., and Welleck, S. Llemma: An open language model for mathematics. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS'23*, 2023b.

Bibel, W. *Automated theorem proving*. Springer Science & Business Media, 2013.

Colas, C., Karch, T., Sigaud, O., and Oudeyer, P.-Y. Autotelic agents with intrinsically motivated goal-conditioned reinforcement learning: a short survey. *Journal of Artificial Intelligence Research*, 74:1159–1199, 2022.

Dong, K., Mahankali, A., and Ma, T. Formal theorem proving by rewarding llms to decompose proofs hierarchically. *arXiv preprint arXiv:2411.01829*, 2024.

Einarsdóttir, S. H., Alhessi, Y., First, E., and Johansson, M. On lemma conjecturing using neural, symbolic and neuro-symbolic approaches. 2024.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Haluptzok, P., Bowers, M., and Kalai, A. T. Language models can teach themselves to program better. *arXiv preprint arXiv:2207.14502*, 2022.

Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., Helyar, A., Madry, A., Beutel, A., Carney, A., et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Jiang, A. Q., Li, W., Han, J. M., and Wu, Y. Lisa: Language models of isabelle proofs. In *6th Conference on Artificial Intelligence and Theorem Proving*, pp. 378–392, 2021.

Jiang, A. Q., Li, W., Tworkowski, S., Czechowski, K., Odrzygóźdź, T., Miłoś, P., Wu, Y., and Jamnik, M. Thor: Wielding hammers to integrate language models and automated theorem provers. *Advances in Neural Information Processing Systems*, 35:8360–8373, 2022a.

Jiang, A. Q., Welleck, S., Zhou, J. P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., and Lample, G. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022b.

Jiang, A. Q., Li, W., and Jamnik, M. Multilingual mathematical autoformalization. *arXiv preprint arXiv:2311.03755*, 2023.

Johansson, M. and Smallbone, N. Exploring mathematical conjecturing with large language models. 2023.

Kaliszyk, C., Urban, J., Michalewski, H., and Olšák, M. Reinforcement learning of theorem proving. *Advances in Neural Information Processing Systems*, 31, 2018.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Lample, G., Lacroix, T., Lachaux, M.-A., Rodriguez, A., Hayat, A., Lavril, T., Ebner, G., and Martinet, X. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349, 2022.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Lin, H., Sun, Z., Yang, Y., and Welleck, S. Lean-star: Learning to interleave thinking and proving. *arXiv preprint arXiv:2407.10040*, 2024.

Loveland, D. W. *Automated theorem proving: A logical basis*. Elsevier, 2016.

Lu, J., Wan, Y., Liu, Z., Huang, Y., Xiong, J., Liu, C., Shen, J., Jin, H., Zhang, J., Wang, H., et al. Process-driven autoformalization in lean 4. *arXiv preprint arXiv:2406.01940*, 2024.

mathlib Community, T. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pp. 367–381, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370974. doi: 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824.

Moura, L. d. and 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 28*, pp. 625–635. Springer, 2021.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Nipkow, T., Wenzel, M., and Paulson, L. C. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

Parker-Holder, J., Jiang, M., Dennis, M., Samvelyan, M., Foerster, J., Grefenstette, E., and Rocktäschel, T. Evolving curricula with regret-based environment design. In *International Conference on Machine Learning*, pp. 17473–17498. PMLR, 2022.

Plaat, A., Wong, A., Verberne, S., Broekens, J., van Stein, N., and Back, T. Reasoning with large language models, a survey. *arXiv preprint arXiv:2407.11511*, 2024.

Poesia, G., Broman, D., Haber, N., and Goodman, N. D. Learning formal mathematics from intrinsic motivation. *arXiv preprint arXiv:2407.00695*, 2024.

Polu, S., Han, J. M., Zheng, K., Baksys, M., Babuschkin, I., and Sutskever, I. Formal mathematics statement curriculum learning. *arXiv preprint arXiv:2202.01344*, 2022.

Portelas, R., Colas, C., Weng, L., Hofmann, K., and Oudeyer, P.-Y. Automatic curriculum learning for deep rl: A short survey. *arXiv preprint arXiv:2003.04664*, 2020.

Pourcel, G., Carta, T., Kovač, G., and Oudeyer, P.-Y. Autotelic llm-based exploration for goal-conditioned rl. In *Intrinsically Motivated Open-ended Learning Workshop at NeurIPS 2024*, 2024a.

Pourcel, J., Colas, C., Molinaro, G., Oudeyer, P.-Y., and Teodorescu, L. Aces: generating diverse programming puzzles with autotelic language models and semantic descriptors. *Neurips*, 2024b.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Zhang, M., Li, Y., Wu, Y., and Guo, D. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Shinn, N., Cassano, F., Labash, B., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning.(2023). *arXiv preprint cs.AI/2303.11366*, 2023.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7676):484–503, 2016.

Teodorescu, L., Colas, C., Bowers, M., Carta, T., and Oudeyer, P.-Y. Codeplay: Autotelic learning through collaborative self-play in programming environments. In *IMOL 2023-Intrinsically Motivated Open-ended Learning workshop at NeurIPS 2023*, 2023.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Trinh, T. and Luong, T. Alphageometry: An olympiad-level ai system for geometry. *Google DeepMind*, 17, 2024.

Tsoukalas, G., Lee, J., Jennings, J., Xin, J., Ding, M., Jennings, M., Thakur, A., and Chaudhuri, S. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. *arXiv preprint arXiv:2407.11214*, 2024.

Urban, J. and Jakubův, J. First neural conjecturing datasets and experiments. In *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings 13*, pp. 315–323. Springer, 2020.

Wang, H., Xin, H., Zheng, C., Li, L., Liu, Z., Cao, Q., Huang, Y., Xiong, J., Shi, H., Xie, E., et al. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656*, 2023.

Wang, M. and Deng, J. Learning to prove theorems by learning to generate theorems. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, pp. 18146–18157, 2020.

Wang, R., Zhang, J., Jia, Y., Pan, R., Diao, S., Pi, R., and Zhang, T. Theoremllama: Transforming general-purpose llms into lean4 experts. *arXiv preprint arXiv:2407.03203*, 2024.

Wu, M., Norrish, M., Walder, C., and Dezfouli, A. Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning. *Advances in Neural Information Processing Systems*, 34:9330–9342, 2021.

Wu, Y., Jiang, A. Q., Ba, J., and Grosse, R. Int: An inequality benchmark for evaluating generalization in theorem proving. *arXiv preprint arXiv:2007.02924*, 2020.

Wu, Z., Huang, S., Zhou, Z., Ying, H., Wang, J., Lin, D., and Chen, K. Internlm2. 5-stepprover: Advancing automated theorem proving via expert iteration on large-scale lean problems. *arXiv preprint arXiv:2410.15700*, 2024.

Xin, H., Guo, D., Shao, Z., Ren, Z., Zhu, Q., Liu, B., Ruan, C., Li, W., and Liang, X. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024a.

Xin, H., Ren, Z., Song, J., Shao, Z., Zhao, W., Wang, H., Liu, B., Zhang, L., Lu, X., Du, Q., et al. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *arXiv preprint arXiv:2408.08152*, 2024b.

Yang, K., Poesia, G., He, J., Li, W., Lauter, K., Chaudhuri, S., and Song, D. Formal mathematical reasoning: A new frontier in ai. *arXiv preprint arXiv:2412.16075*, 2024a.

Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R. J., and Anandkumar, A. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36, 2024b.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Ye, Z., Agarwal, R., Liu, T., Joshi, R., Velury, S., Le, Q. V., Tan, Q., and Liu, Y. Evolving alignment via asymmetric self-play. *arXiv preprint arXiv:2411.00062*, 2024.

Ying, H., Wu, Z., Geng, Y., Wang, J., Lin, D., and Chen, K. Lean workbook: A large-scale lean problem set formalized from natural language math problems. *arXiv preprint arXiv:2406.03847*, 2024.

Zhang, J., Lehman, J., Stanley, K., and Clune, J. Omni: Open-endedness via models of human notions of interestingness. *arXiv preprint arXiv:2306.01711*, 2023.

Zheng, C., Wang, H., Xie, E., Liu, Z., Sun, J., Xin, H., Shen, J., Li, Z., and Li, Y. Lyra: Orchestrating dual correction in automated theorem proving. *arXiv preprint arXiv:2309.15806*, 2023.

Zheng, K., Han, J. M., and Polu, S. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2021.

A. Additional Implementation Details

In this section, we list the missing implementation details.

A.1. Examples of inputs and outputs of our model

Here we present some concrete examples to demonstrate the input and output formats of our model.

Examples of the conjecturer. In the following, we show examples of the conjecturer's inputs and outputs. Note that <lemma>,<easy theorem>,<hard theorem>,</hard theorem> are the formatting tokens, and the system prompt is the first three lines in the input examples.

Input:

```
Complete the following Lean 4 code:

```lean4
<lemma>
lemma sq_nonneg (a : α) : 0 ≤ a ^ 2
<easy theorem>
theorem lean_workbook_9742 (a b c : ℝ) (ha : a ≥ 0) (hb : b ≥ 0) (hc : c ≥ 0) (hab : a +
    b + c = 1) : a^2 + b^2 + c^2 ≤ 1 / 4 + a^3 + b^3 + c^3  :=  by
nlinarith [sq_nonneg (a - 1 / 2), sq_nonneg (b - 1 / 2), sq_nonneg (c - 1 / 2),
mul_nonneg ha hb, mul_nonneg hb hc, mul_nonneg ha hc, hab]
<hard theorem>
theorem
```
```

Output:

```
lean_workbook_9229 (a b c : ℝ) (ha : 0 < a ∧ 0 < b ∧ 0 < c) (hab : a + b + c = 1) : a^3
    / (a * b) + b^3 / (b * c) + c^3 / (c * a) ≥ 1 - a^2 - b^2 - c^2</hard theorem>
```
```

In the following example, the lemma is replaced by a trivial statement, and the model can generate conjectures without focusing on any particular direction (see Section 4.1).

*Input*:

```
Complete the following Lean 4 code:

```lean4
<lemma>
theorem true: True
<easy theorem>
theorem lean_workbook_980 (a b : ℝ) (ha : 0 < a) (hb : 0 < b) (hab : 1/a + 1/b = 1) :
    1/(a + 1) + 2/(2 * b + 1) ≤ 3/4  :=  by
field_simp [ha.ne', hb.ne', ha.ne', hb.ne'] at hab
field_simp [ha.ne', hb.ne', ha.ne', hb.ne']
rw [div_le_div_iff (by positivity) (by positivity)]
nlinarith [sq_nonneg (a - b), sq_nonneg (a - 2 * b), sq_nonneg (2 * a - b), hab]
<hard theorem>
theorem
```
```

*Output*:

```
lean_workbook_982 (a b c : ℝ) (ha : 0 < a) (hb : 0 < b) (hc : 0 < c) (hab : a + b + c =
    1) :
a / a ^ 2 + b / b ^ 2 + c / c ^ 2 ≥ 3 / 4 ∧ 1 / (a + b + c) ^ 2 + 1 / (b + c + a) ^ 2 +
    1 / (c + a + b) ^ 2 ≥ 3 / 4</hard theorem>
```
```

Examples of the prover. In the following, we show an example of the prover's inputs and outputs.

Input:

```
Complete the following Lean 4 code:
```

```
```lean4
theorem lean_workbook_52370 (p q : ℕ) (hp : Nat.Prime p) (hq : Nat.Prime q) (hpq: p >= q)
    (hq5 : q >= 5) : 24 ∣ (p^2 - q^2)  :=  by
```

Output:

```
have : p^2 % 24 = (p % 24)^2 % 24 := by
rw [Nat.pow_mod]
have : q^2 % 24 = (q % 24)^2 % 24 := by
rw [Nat.pow_mod]
rw [← Nat.mod_add_div p 24, ← Nat.mod_add_div q 24]
ring_nf
norm_num
have : p % 24 = 1 ∨ p % 24 = 5 ∨ p % 24 = 7 ∨ p % 24 = 11 ∨ p % 24 = 13 ∨ p % 24 = 17 ∨
    p % 24 = 19 ∨ p % 24 = 23 := by
have := hp.eq_one_or_self_of_dvd 2
have := hp.eq_one_or_self_of_dvd 3
have := hp.eq_one_or_self_of_dvd 4
have := hp.eq_one_or_self_of_dvd 5
omega
have : q % 24 = 1 ∨ q % 24 = 5 ∨ q % 24 = 7 ∨ q % 24 = 11 ∨ q % 24 = 13 ∨ q % 24 = 17 ∨
    q % 24 = 19 ∨ q % 24 = 23 := by
have := hq.eq_one_or_self_of_dvd 2
have := hq.eq_one_or_self_of_dvd 3
have := hq.eq_one_or_self_of_dvd 4
have := hq.eq_one_or_self_of_dvd 5
omega
rcases this with (h | h | h | h | h | h | h | h) <;> rcases this with (h' | h' | h' | h' | h' |
    h' | h' | h') <;>
simp [h, h', pow_two, Nat.mul_mod, Nat.add_mod, Nat.mod_mod, Nat.mod_self, zero_add,
Nat.zero_mul, Nat.mod_one, Nat.one_mul, Nat.mod_add_mod]
<;> omega
```
```

A.2. Pseudo-code for selecting the conjecturer's inputs

In the following, we present the pseudo-code for selecting the conjecturing imports. Recall that the input for the conjecturer consists of a statement, its proof, and a lemma used in the proof (c.f., Section 3.1). In Step 1, we construct the prompts by taking the correct proofs to statements in the given dataset, and extract the lemmas used in the proof by the formal verifiers. We also allow the model to propose conjectures without focusing on any particular lemma, which is implemented by replacing the lemma statement with a fixed trivial statement in the prompt (see Appendix A.1 for concrete examples). Finally, we de-duplicate the inputs by the (statement, lemma) pair. After generating the conjectures, we randomly select a subset whose size does not exceed the number of remaining unproved statements in the given dataset, so that the prover's sample budget is distributed equally between the conjectures and the statements.

We run two heuristic methods to ensure the diversity of the inputs. First, we make sure that each lemma l appears at most $0.1n$ times in the inputs because we observe that some lemmas (e.g., `sq_nonneg`, `mul_self_nonneg`) are much more likely to be included. Second, we make sure that every statement-lemma pair only appear at most once in the prompt, even if there are multiple correct proofs.

Alg. 1 shows the complete pseudo-code for selecting the conjecturer's inputs.

A.3. Pseudo-code for preparing the conjecturer dataset.

The pseudo-code for preparing the conjecturer dataset is shown in Alg. 2. The motivations and explanations of each step in Alg. 2 can be found in Section 3.2.

A.4. Pre-processing LeanWorkbook

LeanWorkbook is a dataset that contains statements translated from natural language math statements (a.k.a., auto-formalization). The original dataset contains 140K (natural language statement, formal statement) pairs.

Algorithm 1 Prepare inputs for the conjecturer.

Input: a list of statements and proofs $L = \{(t_i, p_i)\}_{i=1}^n$.
Initialize prompt list $P = []$.
for $(t, p) \in L$ **do**
  Parse the proof and get the set of used lemmas S.
  With probability 0.5, add the trivial lemma to S.
  For every lemma $l \in S$, add (t, p, l) to the prompt list P.
end for
for $l \in S$ **do**
  if $\sum_{(t',p',l') \in P} \mathbb{I}[l = l'] > 0.1n$ **then**
    Randomly keep at most $0.1n$ prompts with lemma l in P.
  end if
end for
De-duplicate P randomly so that every (statement, lemma) pair (t, l) appears at most once.
Return: de-duplicated list of prompts P.

Algorithm 2 Prepare the conjecturer dataset.

Input: a list of (seed statement, proof of the seed statement, lemma, generated conjecture, generated proof of the conjecture) tuples $\mathcal{D} = \{(t_i, p_i^t, l_i, c_i, p_i^c)\}_{i=1,\cdots,n}$, and unproved statements $Q = \{t_j\}_{j=1,\cdots,m}$.
For each conjecture c, compute its empirical pass rate

$$\hat{P}(c) \triangleq (\#\{i : c_i = c, p_i^c \text{ is correct}\})/(\#\{i : c_i = c\}).$$

Select conjecturing examples that (a) have low but positive pass rates, and (b) the lemma l is used in the proof p^c:

$$\overline{\mathcal{D}} = \{(t, p^t, l, c) \,|(t, p^t, l, c, p^c) \in \mathcal{D}, \hat{P}(c) \in (0, 1/4], \\ p^c \text{ is correct}, l \text{ is used in } p^c\}.$$

De-duplicate $\overline{\mathcal{D}}$ based on the conjecture c.
Compute the elegancy score

$$E(c) \triangleq \frac{\min\{\text{len}(p_i^c) : 1 \le i \le n, p_i^c \text{ is correct}, c_i = c\}}{\text{len}(c)}$$

Let κ be the 20%-quantile of $E(c)$ for conjectures in $\overline{\mathcal{D}}$.
Apply elegancy filter: $\widetilde{\mathcal{D}} = \{(t, p^t, l, c) \in \overline{\mathcal{D}} \mid E(c) \ge \kappa\}$.
Find a distribution P supported on the conjectures in $\widetilde{\mathcal{D}}$ that minimizes the Wasserstein distance $W(P, Q)$ (Alg. 4).
Return: $\widetilde{\mathcal{D}}$ re-weighted by the density of P.

We de-duplicate the LeanWorkbook dataset by keeping only one formal statements per natural language statement. After de-duplication, we get 89,221 formal Lean4 statements as our existing dataset w/o proofs for Lean experiments.

For the Isabelle experiments, we translate the Lean4 statements to Isabelle using DeepSeek-V2.5 API with few-shot examples. The prompt to the model is listed below.

```
Please translate the following lean statement into Isabelle. Please make sure that
1. All the variables are well-typed.
2. All the functions are correctly translated into the corresponding Isabelle functions.
3. All the symbols are correctly translated into corresponding Isabelle symbols.
4. Please directly output the translation without explanation.

Here are some hints for the translation:
1. In Isabelle, the second operand of the operator ^ should be integer. For real numbers,
    please use powr instead.
2. Please define the types of numerals.
```

```
3. `Real.logb x y` should be translated to `log x y`.
4. `Real.sqrt x` should be translated to `sqrt x`.
5. Variables with subscripts should be disallowed. For any variable names of form a_b,
    translate it to ab.
6. Please translate superscripts to the corresponding exponential form. For example, x^-1
    should be translated to (x powr -1).
7. `a ∣ b` should be translated to `a dvd b`.
8. `x ≡ y [ZMOD p]` should be translated to `x mod p = y mod p`.
9. `x ∈ zmod p` should represent that x is nat and x < p.

## Input:
```lean
theorem lean_workbook_50 (a b c : ℝ)
(ha : a ≥ 0 ∧ b ≥ 0 ∧ c ≥ 0)
(hab : a + b + c = 3)
: a^3 + b^3 + c^3 + 216 * (a * b + b * c + c * a) / (24 + a * b + b * c + c * a) ≤ 27  :=
     by sorry
```

Output:
```Isabelle
theorem lean_workbook_50:
fixes a b c :: real
assumes "a ≥ 0 ∧ b ≥ 0 ∧ c ≥ 0"
assumes "a + b + c = 3"
shows "a^3 + b^3 + c^3 + 216 * (a * b + b * c + c * a) / (24 + a * b + b * c + c * a) ≤
    27"
sorry
```

Input:
```lean
{}
```

Output:
```

A.5. Re-weighting the conjecturing dataset

In this section, we describe the motivations and implementation details of the re-weighting method for the conjecturing dataset.

Motivation. In our early experiments, we observe that the generated conjectures tend to have mode collapse issue after several iterations of self-play training. For example, the generated conjectures are mostly about algebraic manipulations even when the seed statements contain questions about, for example, number theory. This is partly because the LeanWorkbook dataset contains a significant portion of inequality questions.

Therefore, in addition to the particular conjecturing format where we require that the proof of the conjecture must use the lemma given in the input, we also re-weight the conjecturing examples at every iteration. Intuitively, if there is a distance function that can separate statements of different topics, the Wasserstein projection of the conjectures will have a similar distribution of topics, and therefore alleviates the mode collapsing issue.

Cost function. We compute the cost $d(x, y)$ of matching conjecture x to a statement y by the negative of the cosine similarity between their embeddings, and the embedding is computed by the last hidden layer of the current model averaged over the sequence dimension. Since our model is trained to generate proofs of conjectures and statements, we expect that statements with similar proof techniques tend to have similar embeddings, and therefore smaller cost for the matching.

Algorithm. On the high level, our method computes a re-weighting of the generated conjectures that minimizes its Wasserstein distance to the unproved statements in the given dataset. Abstractly speaking, let $\mathcal{X}$ be the set of generated conjectures, and Q the set of unproved statements. Let $d(x, y)$ be the distance between a conjecture x and a statement y.

Then, the optimization problem can be written as

$$\operatorname*{argmin}_{P:P \text{ is a valid distribution, } \operatorname{supp}(P)\subseteq\mathcal{X}} W(P,Q), \tag{1}$$

where $W(P,Q)$ is the Wasserstein distance between P and Q (with little abuse of notation, we use Q to represent the uniform distribution over the unproved statements). The Wasserstein distance $W(P,Q)$ is defined by the following optimal transportation problem where μ is a matching between the distribution P and Q:

$$\begin{aligned}
W(P,Q) = \min_{\mu} \quad & \sum_{x\in\operatorname{supp}(P),y\in\operatorname{supp}(Q)} \mu(x,y)d(x,y) && (2)\\
\text{s.t.} \quad & \sum_{y\in\operatorname{supp}(Q)} \mu(x,y) = P(x), && (3)\\
& \sum_{x\in\operatorname{supp}(P)} \mu(x,y) = Q(y), && (4)\\
& \mu(x,y) \geq 0, \quad \forall x,y. && (5)
\end{aligned}$$

Combining the equations above, the re-weighting distribution P can be computed by

$$\begin{aligned}
\operatorname*{argmin}_{P:\operatorname{supp}(P)\subseteq\mathcal{X}} \min_{\mu} \quad & \sum_{x\in\operatorname{supp}(P),y\in\operatorname{supp}(Q)} \mu(x,y)d(x,y) && (6)\\
\text{s.t.} \quad & \sum_{y\in\operatorname{supp}(Q)} \mu(x,y) = P(x), && (7)\\
& \sum_{x\in\operatorname{supp}(P)} \mu(x,y) = Q(y), && (8)\\
& \mu(x,y) \geq 0, \quad \forall x,y, && (9)\\
& P(x) \geq 0, \quad \forall x, && (10)\\
& \sum_{x\in\mathcal{X}} P(x) = 1, && (11)
\end{aligned}$$

where the last two constraint ensures that P is a valid distribution. Equivalently, we get the following program,

$$\begin{aligned}
\operatorname*{argmin}_{P:\operatorname{supp}(P)\subseteq\mathcal{X}} \min_{\mu} \quad & \sum_{x\in\operatorname{supp}(P),y\in\operatorname{supp}(Q)} \mu(x,y)d(x,y) && (12)\\
\text{s.t.} \quad & \sum_{x\in\operatorname{supp}(P)} \mu(x,y) = Q(y), && (13)\\
& \sum_{x\in\mathcal{X},y\in\operatorname{supp}(Q)} \mu(x,y) = 1, && (14)\\
& \mu(x,y) \geq 0, \quad \forall x,y, && (15)\\
& P(x) = \sum_{y\in\operatorname{supp}(Q)} \mu(x,y). && (16)
\end{aligned}$$

Since $Q(y)$ is given, we can optimize $\mu(x,y)$ for every fixed y separately, and then compute the final $P(x)$ using Eq. (16). As a result, the program above has a closed-form solution $\mu^\star(x,y) = Q(y)\mathbb{I}\left[x = \operatorname{argmin}_{x'\in\mathcal{X}} d(x',y)\right]$ and $P(x) = \sum_{y\in\operatorname{supp}(Q)} \mu^\star(x,y)$. In other words, the optimal matching $\mu(x,y)$ for any given y is only supported at the x that minimizes the distance $d(x,y)$. Therefore, the (theoretical) algorithm that computes the optimal re-weighting is given in Alg. 3. Note that the last line in Alg. 3 is to make sure that the sum of the weights equals the number of generated conjectures (i.e., the sum of weights before re-weighting).

Our practical implementation is shown in Alg. 4. In this implementation, we additionally requires that the weighting P for every conjecture x cannot be too big because otherwise it might cause instability of the LLM training with weighted cross

Algorithm 3 Computing the optimal re-weighting (theory).

Input: generated conjectures $\mathcal{X} = \{x_1, \cdots, x_n\}$ of size n, unproved statements Q with size m, and a distance function $d(x, y)$.
Initialize the optimal re-weighting $P = [0, 0, \cdots, 0]$.
for $y \in Q$ **do**
  Compute $x^\star = \text{argmin}_{x \in \mathcal{X}} d(x, y)$.
  $P(x^\star) \leftarrow P(x^\star) + 1/m$.
end for
Return: the optimal re-weighting is $[P(x_1) * n, P(x_2) * n, \cdots, P(x_n) * n]$.

entropy loss. We also allow unproved statements in Q to have different matching weights — an important statement can be matched to more than one conjecture (see Line 5-6 of Alg. 4). In both the Isabelle and Lean experiments, the statements from LeanWorkbook have matching weight 1. The statements from miniF2F-valid and ProofNet-valid have matching weight 1 for the first 24 iterations in the Lean experiment, and 128 afterward.

Algorithm 4 Computing the optimal re-weighting.

1: **Input:** generated conjectures $\mathcal{X} = \{x_1, \cdots, x_n\}$ of size n, unproved statements Q with size m, and a distance function $d(x, y)$.
2: Initialize the optimal re-weighting $P = [0, 0, \cdots, 0]$.
3: Initialize the masks $M(x) = 1, \forall x \in \mathcal{X}$.
4: **for** $y \in Q$ **do**
5: Let k be the matching weight of y.
6: Let $x^1, \cdots, x^k$ be the k conjectures with smallest value of $d(\cdot, y)M(\cdot)$.
7: **for** $i = 1, \cdots, k$ **do**
8: $P(x^i) \leftarrow P(x^i) + 1/m$.
9: **if** $P(x^i) * n > 3$ **then**
10: $M(x^i) \leftarrow 0$.
11: **end if**
12: **end for**
13: **end for**
14: **Return:** the optimal re-weighting is $[P(x_1) * n, P(x_2) * n, \cdots, P(x_n) * n]$.

A.6. Implementation details for expert iteration.

In this section, we describe two different implementations of expert iteration and compare their performance.

Vanilla expert iteration. For vanilla expert iteration, we only sample proofs to the *unproved* statements in the given dataset. The LLM training dataset consists of all the correct proofs generated in this and previous iterations, and in each iteration, the model is trained from the base model.

Optimized expert iteration. The most significant issue of vanilla expert iteration is the limited correct proofs generated in each iteration. As a result, even though the model is re-trained at every iteration, the difference between two models in consecutive iterations are limited.

Therefore, in our optimized implementation of expert iteration, we generate proofs to all statements in the given dataset, regardless of whether they are previously proved or not. Then, to construct the LLM training dataset, we randomly choose at most 16 proofs per statement (so that the model does not overfit to the easy problems with many correct proofs). Note that this implementation requires slightly more sample budget per iteration. However, since the pass rate on the given dataset is low (less than 30% even for our best model), this difference is not significant.

In Fig. 7 (Left), we plot the cumulative pass rate of two implementations of expert iteration, STP and parallel sampling. STP outperforms both implementations of expert iteration, and the optimized implementation of expert iteration is better than the vanilla implementation.

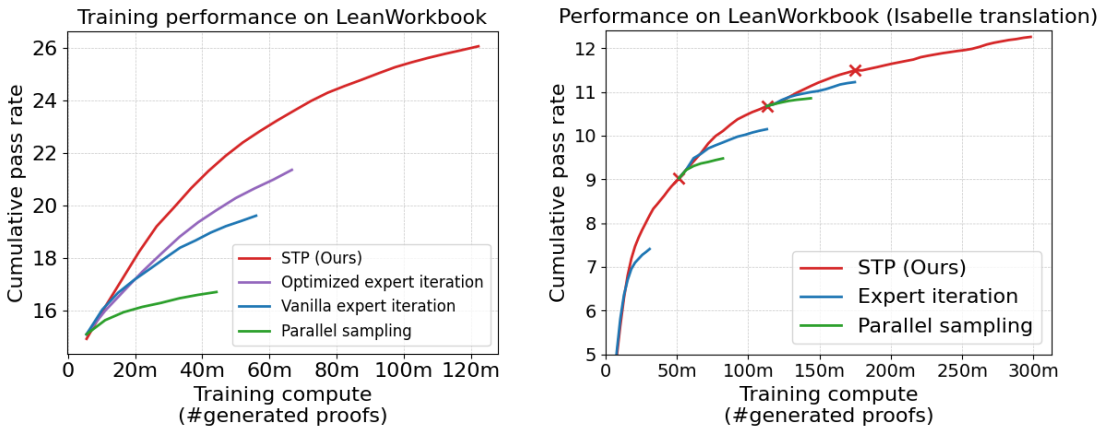

Figure 7: **Left:** Comparison of pass rates between STP, two implementations of expert iteration, and parallel sampling methods on LeanWorkbook. **Right:** Comparison of pass rates between STP and baseline methods on LeanWorkbook (Isabelle translation). The red crosses shows the points where we refresh the self-play training as described in Section 4.1.

For the figures of Isabelle experiments, we always use the optimizes implementation of expert iteration. For Fig. 2, we use the vanilla implementation.

A.7. Additional details for interacting with the Isabelle verifier

For the Isabelle experiments, we have an additional filter for the conjectures — if the generated conjecture is equivalent to the statement in the prompt (tested by `solve_direct` in Isabelle), we consider it invalid.

We disallow the tactics `sledgehammer, mason, smt, metis, sos` by invalidating proofs that contain any of these sub-strings. However, following the implementation of Jiang et al. (2022a), we still use the keyword 'sledgehammer' to replace the following simple tactics

```
[by auto, by simp, by blast, by fastforce, by force, by eval, by presburger,
by arith, by linarith, by (auto simp: field_simps)].
```

During proof verification, we try these tactics sequentially to replace the keyword 'sledgehammer'. If any of the tactics succeed, we proceed to the remaining proof steps. Otherwise we flag the proof incorrect.

A.8. Additional details for interacting with the Lean4 verifier

During the self-play training, we use the same imports as the miniF2F Lean4 project[10] instead importing the entire Mathlib to optimize the memory efficiency. This is because we do not have access to an additional CPU cluster for proof verification, and the available CPU memory in TPU-v4 VMs is limited.

A.9. Compute resources

Our experiments are primarily done on TPU-v4 VMs with 32 nodes. Each node contains 4 TPU chips (8 TPU cores), 240 CPU cores, and 400G memory. We use vLLM (Kwon et al., 2023) to generate LLM outputs, and Levanter[11] to train the LLM. In both STP and expert iteration, since the generated proofs are heavily filtered (based on the correctness, elegancy, trivialness, etc.) when constructing the training dataset, LLM training only takes less than 25% of the wall-clock time for TPU compute, and generating proofs takes the rest 75%.

[10] https://github.com/yangky11/miniF2F-lean4/tree/main/MiniF2F
[11] https://github.com/stanford-crfm/levanter

B. Additional Experiment Results

In this section we show the additional experiment results with both Lean and Isabelle formal verifier.

B.1. Additional results with Lean

In Table 4, we compare the performance of our method with prior works on PutnamBench. Note that DSP (Jiang et al., 2022b) uses Isabelle verifier where PutnamBench only has 640 statements. Our model STP achieves state-of-the-art performance by solving 8 out of 644 problems.

Table 3 compares the model obtained by final re-training with and without the proofs of generated conjectures, as discussed in the ablation study section (Section 4.4). The results show that it is still beneficial to re-train with the generated conjectures in addition to the successfully proved statements in LeanWorkBook even for performance on miniF2F-test and ProofNet-test, which leads to about 2-3% performance gain (for pass@128).

Table 3: Pass rate on miniF2F and ProofNet.

Method	Sample budget	MiniF2F-test	ProofNet-test
STP *(w/o conjectures)*	128	$58.3\% \pm 0.7\%$	$17.4\% \pm 0.4\%$
STP	128	$61.2\% \pm 0.6\%$	$19.5\% \pm 0.7\%$

B.2. Additional results with Isabelle

In Fig. 7 (Right), we plot the pass rates of STP and baseline methods on LeanWorkbook starting from iteration 0. The red crosses shows the points where we refresh the training process as described in Section 4.1. Our models are tested with PutnamBench (Tsoukalas et al., 2024), commit `d49896f`.[12]

Table 4: Results on PutnamBench.

Method	Sample budget (#Proofs / #Steps)	Result
Whole-Proof Generation Methods		
DSP (GPT-4o) (Jiang et al., 2022b)	10	4/640
STP	128	**7/644**
	3200	**8/644**
Tree Search Methods		
InternLM2.5-StepProver-BF+CG	$2 \times 32 \times 600$	6/644

B.3. Examples of unproved statements in LeanWorkbook

In this section, we list 20 randomly selected statements from LeanWorkbook that are not proved during STP training. The following table shows the formal statement, the corresponding natural language statement in LeanWorkbook, the correctness of formalization, and the correctness of the formal statement.

#	Lean formal statement	Natural language statement	Correct formalization?	Correct statement?
1	`theorem lean_workbook_7116 (x y z : ℝ) (hx : x + y + z = 3) : x ^ 2 + y ^ 2 + z ^ 2 + 3 ≤ 2 * (1 / x ^ 2 + 1 / y ^ 2 + 1 / z ^ 2) := by`	If $a = x^2, b = y^2, c = z^2$ it suffices to show that: $x+y+z = 3 \implies x^2+y^2+z^2+3 \leq 2(\frac{1}{x^2} + \frac{1}{y^2} + \frac{1}{z^2})$	Yes	No. The case $x = 0$ is ill defined.

[12] https://github.com/trishullab/PutnamBench/tree/d49896fdc87a128a70e15a185d8dfca3516dd894

2	`theorem lean_workbook_plus_72390 (a b n : ℕ) (h : a ≡ b [ZMOD n]) : a^n ≡ b^n [ZMOD n^2] := by`	Prove that if $a \equiv b \pmod n$, then $a^n \equiv b^n \pmod{n^2}$.	Yes	Yes
3	`theorem lean_workbook_35349 (a b c : ℝ) : (9 / (a + b + c + Real.sqrt (3 * (a * b + b * c + c * a)))) ≤ (1 / (a + b) + 1 / (b + c) + 1 / (c + a)) := by`	For: $\frac{9}{a+b+c+\sqrt{3(ab+bc+ca)}} \leq \frac{1}{a+b}+\frac{1}{b+c}+\frac{1}{c+a}$	Yes (but maybe missing the implicit assumption $a,b,c>0$)	No (e.g., $(a,b,c)=(-0.5,1,1)$.)
4	`theorem lean_workbook_8880 (a b c : ℝ) : a * Real.sqrt (b ^ 2 + c ^ 2) + b * Real.sqrt (c ^ 2 + a ^ 2) + c * Real.sqrt (a ^ 2 + b ^ 2) ≤ 3 * Real.sqrt 2 := by`	Prove that $a\sqrt{b^2+c^2}+b\sqrt{c^2+a^2}+c\sqrt{a^2+b^2}\leq 3\sqrt{2}$,	Yes	No
5	`theorem lean_workbook_plus_44018 (x : ℝ) (hx : 0 < x) (a : ℝ) (ha : a = x^(1/3)) : a^2 - 2*a - (a^3 - 4)*Real.sqrt (a^3 - 7) - 3*a^3 + 28 = 0 := by`	Put $\sqrt[3]{x}=a$. The equation is equivalent to $a^2-2a-(a^3-4)\sqrt{a^3-7}-3a^3+28=0$	Yes	No
6	`theorem lean_workbook_plus_35882 (a b c : ℝ) (ha : 0 < a) (hb : 0 < b) (hc : 0 < c) : (Real.sqrt ((a + 2 * b + 3 * c) / (4 * a + b + c)) + Real.sqrt ((3 * a + b + 2 * c) / (a + 4 * b + c)) + Real.sqrt ((2 * a + 3 * b + c) / (a + b + 4 * c))) ≥ 3 := by`	If $a,b,c>0$ prove or disprove $\sqrt{\frac{a+2b+3c}{4a+b+c}}+\sqrt{\frac{3a+b+2c}{a+4b+c}}+\sqrt{\frac{2a+3b+c}{a+b+4c}}\geq 3$	Yes	Yes
7	`theorem lean_workbook_12619 : ∀ x y : ℝ, (x^2+x+xy+y^2) ≤ 1 → -(1/3)*Real.sqrt ((1/2)*(69+11*Real.sqrt 33)) ≤ x^2+2*x*y ∧ x^2+2*x*y ≤ (1/3)*Real.sqrt ((1/2)*(69+11*Real.sqrt 33)) := by`	Let $x^2+x+xy+y^2\leq 1 (x,y\in R)$. Prove that $-\frac{1}{3}\sqrt{\frac{1}{2}(69+11\sqrt{33})}\leq x^2+2xy\leq \frac{1}{3}\sqrt{\frac{1}{2}(69+11\sqrt{33})}$	No (there is a xy term in Lean. Should be x * y)	No
8	`theorem lean_workbook_plus_20629 (f : ℝ → ℝ) (x : ℝ) : f (f x + 1) = f x + 1 := by`	Prove that $f(f(x)+1)=f(x)+1$ for all real x.	Yes	No
9	`theorem lean_workbook_37208 (n : ℕ) (hn : 0 < n) : (n : ℝ) / (n! : ℝ) ^ (1 / n) < (1 + 1 / n)^n := by`	Prove that: $\frac{n}{\sqrt[n]{n!}}<\left(1+\frac{1}{n}\right)^n$ for every positive integer n	Yes	Yes
10	`theorem lean_workbook_10259 (a b : ℕ) (hab : a ≠ b) (h : a + b ∣ a^2 + b^2) : a * b + 4 ≤ (Nat.gcd a b)^4 := by`	Given $a,b\in\mathbb{N}(a\neq b)$ so that $a+b \mid a^2+b^2$. Let $d=gcd(a,b)$. Prove that $ab+4\leq d^4$	Yes	Yes
11	`theorem lean_workbook_45322 (a b : ℝ) (ha : 0 < a) (hb : 0 < b) (hab : (a + 1 / a) * (b + 1 / b) = 2 + 3 / Real.sqrt 2) : 1 ≤ a ^ 4 + b ^ 4 ∧ a ^ 4 + b ^ 4 ≤ 4 := by`	Let $a,b>0$ and $(a+\frac{1}{a})(b+\frac{1}{b})=2+\frac{3}{\sqrt{2}}$. Prove that $1\leq a^4+b^4\leq 4$	Yes	Yes

12	`theorem lean_workbook_28189 (x y z : ℝ) : Real.sqrt (1 + 48 * x / (y + z)) ≥ (184 * x ^ 2 - 32 * (y ^ 2 + z ^ 2) + 289 * x * (y + z) + 127 * y * z) / (8 * (x ^ 2 + y ^ 2 + z ^ 2) + 47 * (y * z + z * x + x * y)) := by`	prove: $\sqrt{1+\frac{48x}{y+z}} \geq \frac{184x^2-32(y^2+z^2)+289x(y+z)+127yz}{8(x^2+y^2+z^2)+47(yz+zx+xy)}$	Yes (but maybe missing $x \geq 0$)	No $x = -1/24, y = z = 1$
13	`theorem lean_workbook_31673 (x : ℝ) (h₀ : Σ' k : ℕ, (7 / (2^k)) = x) : x = 14 := by`	Solution without Geometric Formula $\frac{7}{1}+\frac{7}{2}+\frac{7}{4}+\frac{7}{8}\cdots = x$ We divide everything by 2: $\frac{7}{2}+\frac{7}{4}+\frac{7}{8}+\frac{7}{16}\cdots = \frac{x}{2}$ We substitute the original equation in: $x-7 = \frac{x}{2}$ $\frac{x}{2} = 7$ Therefore, $\boxed{x = 14}$.	Yes	Yes
14	`theorem lean_workbook_4086 (g : ℕ → ℕ) (h₁ : g 1 = g 1 ^ 2) : g 1 = 1 := by`	Given $g(1) = g(1)^2 \Rightarrow g(1) = 1$	No (natural language statement is unclear)	No
15	`theorem lean_workbook_6922 (a b : ℝ) (ha : 0 ≤ a) (hb : 0 ≤ b) (hab : 2 ≤ a + b) : a * Real.sqrt (a / (2 + 7 * b)) + b * Real.sqrt (b / (2 + 7 * a)) + Real.sqrt (1 / (1 + 8 * a * b)) ≥ 1 := by`	Let $a, b \geq 0$ and $a+b \geq 2$. Prove that $a\sqrt{\frac{a}{2+7b}} + b\sqrt{\frac{b}{2+7a}} + \sqrt{\frac{1}{1+8ab}} \geq 1$	Yes	Yes
16	`theorem lean_workbook_plus_65183 (f : ℝ → ℝ): (∀ x y, f (x + f y) = y + f (x + 1)) ↔ (∀ x, f x = x + 1) ∨ (∀ x, f x = -x + 1) := by`	Find all functions $f : \mathbb{R} \to \mathbb{R}$ such that $f(x+f(y)) = y+f(x+1)$, for all $x, y \in \mathbb{R}$.	Yes	No
17	`theorem lean_workbook_26304 (a b c : ℝ) : a + b + c ≤ (a^2 * b^2 + b^2 * c^2 + c^2 * a^2) / (a * b * c) := by`	$\Leftrightarrow a+b+c \leq \frac{a^2b^2+b^2c^2+c^2a^2}{abc}$	No (natural language statement is unclear)	No (e.g., $abc < 0$ and $a+b+c > 0$)
18	`theorem lean_workbook_plus_30866 (x y z : ℝ) (hx : x^3 + y + z = 1) (hy : x + y^3 + z = 1) (hz : x + y + z^3 = 1) : x = y ∧ y = z ∧ z = x := by`	Solve the following system of equations: $\begin{cases} x^3+y+z=1 \\ x+y^3+z=1 \\ x+y+z^3=1 \end{cases}$	No	No
19	`theorem lean_workbook_plus_51637 (A : Matrix (Fin n) (Fin n) ℂ) (h : A * A.transpose = 0) : A = 0 := by`	Let $A \in M_n(\mathbb{C})$ be so that $A \cdot A^t = O_n$. Prove that $A = O_n$. Here, A^t is the transpose of A .	Yes	No (not true for complex matrix)
20	`theorem lean_workbook_plus_58359 (x y z : ℝ) (hx : 0 < x) (hy : 0 < y) (hz : 0 < z) : 1 ≤ x / (Real.sqrt (y * z)) * (1 / (x + 1)) + y / (Real.sqrt (z * x)) * (1 / (y + 1)) + z / (Real.sqrt (x * y)) * (1 / (z + 1)) ∧ x / (Real.sqrt (y * z)) * (1 / (x + 1)) + y / (Real.sqrt (z * x)) * (1 / (y + 1)) + z / (Real.sqrt (x * y)) * (1 / (z + 1)) ≤ Real.sqrt 2 := by`	We also have a nice inequality $1 \leq \frac{x}{\sqrt{yz}} \cdot \frac{1}{x+1} + \frac{y}{\sqrt{zx}} \cdot \frac{1}{y+1} + \frac{z}{\sqrt{xy}} \cdot \frac{1}{z+1} \leq \sqrt{2}$. With 1 and $\sqrt{2}$ are the best constant.	Yes	No (when $x, y, z = \epsilon \to 0$, this term goes to 3)

B.4. Most frequent shared lemmas in Lean

In this section, we list 30 most frequently shared lemmas in the Lean experiments.

```
sq_nonneg, mul_self_nonneg, div_le_div_iff, Real.sq_sqrt, Real.sin_add,
    Real.sin_sq_add_cos_sq, Real.cos_add, pow_two, div_le_iff, Real.cos_sub,
    Real.sin_sub, Real.cos_sq, Real.sin_two_mul, Real.cos_two_mul, pow_mul, div_le_one,
    Real.sqrt_pos, pow_add, Real.tan_eq_sin_div_cos, abs_le, Finset.prod_range_succ',
    pow_pos, Real.cos_sq_add_sin_sq, Nat.sum_range_choose, Nat.pow_mod, pow_three,
    abs_mul, abs_cases, Real.pi_pos, nat_sub_dvd_pow_sub_pow
```