Conjecture extraction for proof autoformalization

Simon Sorg[1], Wenda Li[1], and Soumya Banerjee[1][0000−0001−7748−9885]

University of Cambridge sb2333@cam.ac.uk
{tss52,wl302}@cam.ac.uk

Abstract. Autoformalization and ATP have each advanced the mechanization of mathematics, yet the translation of informal proofs into fully formalized counterparts remains an open challenge: especially for interactive theorem provers beyond Isabelle. We introduce conjecture extraction, a novel proof autoformalization pipeline tailored to Lean 4 that decomposes an informal proof into individual lemmas (conjectures), formalizes and proves each in isolation, and then reassembles them to recover the original argument. Unlike prior sketch-based methods, our approach is compatible with end-to-end proof generation models and leverages repeated conjecture refinement to incrementally improve performance. We implement an open-source system that integrates off-the-shelf autoformalization LLMs, automated theorem provers, and an online reinforcement-learning loop to optimize both conjecture generation and proof search. On the MiniF2F benchmark, conjecture extraction achieves an absolute improvement of 11.2 percentage points in pass@1 over the Draft, Sketch, and Prove port (DSP) for Lean 4, demonstrating the efficacy of proof decomposition and recomposition. Our results suggest that conjecture extraction not only bridges a gap in proof autoformalization for Lean but also offers a general framework for scaling formalization efforts across diverse proof assistants. We release our code and models to foster further research in large-scale formalized mathematics.

Keywords: AI for Math · Autoformalization · Automated theorem proving.

1 Introduction

Formalization is costly, requiring significant labor by a small group of human experts. To aid mathematicians and enable large-scale formalization, autoformalization has been introduced [19]. Autoformalization translates informal statements into formal versions using Large Language Models (LLMs) [19]. Orthogonal to autoformalization, Automated Theorem Proving (ATP) attempts to prove a formal conjecture. Combinations of autoformalization and ATP focus on large-scale conjecture autoformalization to enable more potential training data for ATP models [20].

Few works focus on proof autoformalization, translating existing informal proofs into their formalized counterparts [7]. This area is especially relevant for research mathematics, where current ATP solutions still fall short. Works on

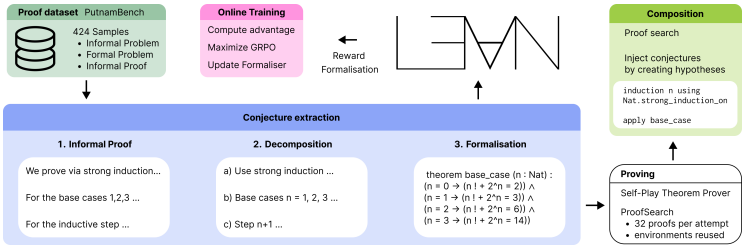

Fig. 1. A general overview of conjecture extraction.

proof autoformalization target the proof assistant Isabelle [12]. Transferring this to Lean 4 [11] is difficult [10]. The only successful implementation of proof autoformalization is, to the best of our knowledge, a port of *Draft, Sketch, and Prove* (DSP) [7] by [1]. In this work, we propose *conjecture extraction*. We decompose proofs into individual proof steps, prove them individually, and recompose the overarching proof. As our setup does not rely on proof sketching, it is compatible with whole-proof generation models. Further, we improve over DSP for Lean, and show how repeated conjecture extraction further boosts performance.

Our contributions are as follows:

- We present conjecture extraction, a proof autoformalization method designed for Lean 4, compatible with whole-proof generation models.
- We open-source a proof-search implementation combining ATP, autoformalization, and conjecture generation with online reinforcement learning for ATP and autoformalization.
- We provide initial experiments, showing the potential of conjecture extraction, improving absolute 11.2% over DSP in pass@1 proof rate on MiniF2F test [26].

2 Related Work

Our work builds upon existing methods in proof autoformalization, conjecture autoformalization, and ATP, with a focus on the Lean 4 proof assistant [11].

Proof autoformalization Proof autoformalization is translating an existing informal proof into a formal one [18]. A prominent method is DSP, where an LLM generates a high-level proof sketch from an informal proof, leaving smaller steps to be solved by pre-existing ATP methods. Originally designed for the Isabelle proof assistant, [10] report problems porting DSP into Lean 4. [1] successfully port DSP into Lean 4. But as LLMs struggle to adhere to consistently adhere to Lean 4 syntax, the performance of DSP in Lean remains limited [1]. Another limitation of DSP is its incompatibility with modern whole-proof generation models [17], as it requires filling in an existing proof structure [7].

ATP LLM-based ATP largely follows two paradigms: proof-step generation and whole-proof generation. Proof-step generation is an interactive process where a model predicts the next tactic based on the current proof state [15], often enhanced with tree-search methods [14]. In contrast, whole-proof generation models produce an entire proof in a single completion [22]. Due to greater computational efficiency, whole-proof generation has become the dominant approach in state-of-the-art ATP systems [17].

Conjecture generation The performance of modern ATP models relies on training via expert iteration, a process that requires large-scale datasets of formal conjectures. To overcome the scarcity of human-written formal problems, many works generate data by autoformalizing large corpora of informal mathematics, creating datasets like the Lean workbook. A more scalable approach is to generate entirely new conjectures. The effectiveness of this was demonstrated by [3], who trained a dedicated conjecture generation model and ran expert iteration on its outputs to achieve state-of-the-art ATP performance.

LegoProver [18] use conjecture generation for proof autoformalization for *LegoProver*, a work closely related to conjecture extraction. LegoProver uses Isabelle, where it aims to solve difficult problems by generating and maintaining a library of proven conjectures. It enhances the DSP framework by using a retrieval mechanism to select useful, already proven premises for a given proof attempt. These premises, along with their proofs, are then copied directly into the context of the LLM, which generates a proof sketch for the main theorem in a two-step process. The informal proof is first rewritten to explicitly state individual proof steps, before this rewritten informal proof is used for a proof-sketch. As LegoProver sketches similar to DSP, the approach is limited to proof-step generation models and does not generalize to whole-proof generation models.

Conjecture extraction also targets individual proof steps but formalizes each step into its own separate conjecture. After these smaller conjectures are proven, the full proof is recomposed. This method only requires the LLM for the natural language task of decomposing the proof, rather than prompting it to generate a complete, syntactically correct formal proof sketch.

Benchmarks Benchmarks for proof autoformalization are sparse. Only few ATP benchmarks also include informal proofs, enabling proof autoformalization evaluation. A popular ATP benchmark with informal proofs is MiniF2F [26]. PutnamBench [16], a famous ATP benchmark based on the William Lowell Putnam Mathematical Competition, only partially contains informal proofs. For our experiments, we web-scrape informal Putnam proofs of the Internet, leading to 424 samples with informal proofs. Other benchmarks, such as Lean workbook [25], do not provide informal proofs. While ProofNet by [2] contains informal proofs, the Lean 4 port by [22] does not. Therefore, our evaluations focus on MiniF2F and PutnamBench.

3 Methodology

This work's major contribution is *conjecture extraction*, which we cover first. Conjecture extraction requires conjecture autoformalization, and we discuss the two methods we tested next. We lastly end with brief implementation details, focusing on our open-source contribution.

Conjecture extraction Decomposing challenging proofs into easier subtasks has been proposed by [18]. Conjecture extraction achieves this by conditioning an LLM on a short prompt together with three problem-specific data points:

- the informal problem statement,
- an informal proof of the problem,
- and a formalized problem statement.

The LLM is tasked with extracting individual isolated proof steps as conjectures from the informal proof. We consider the initial conjecture used to extract conjectures the *parent conjecture*, whose *children* are generated using an LLM. These children adhere to a specified format to facilitate parsing. Specifically, a child conjecture consists of four parts:

1. a preceding *reasoning* block following the chain of thought paradigm
2. a *given* section which introduces variables and their ranges
3. an *assumes* block which states required hypotheses
4. a *shows* area with the conclusion that follows from the variables and assumptions

Note that the differentiation between variables and assumptions is rather blurry. For example, an integer a introduced in the given section could be stated as `positive integer` or as `integer`. In the latter case, positivity could be ensured with an assumption $a > 0$. Both conjectures could yield the same formalization. Still, we decided to separate variables and assumptions, as we found that this pattern arises naturally when prompting various LLMs. We use this format to achieve lower perplexity prompts, which benefits overall generation quality [5]. Once the conjectures are extracted, the LLM output is parsed and autoformalized.

Autoformalization To formalize conjectures, we test both a general-purpose model and a specialized model trained for autoformalization of competition problems [17].

The existing general-purpose LLMs often generate Lean 3 code, as Lean 4 was released in 2021 and existing Lean code still frequently contains Lean 3 syntax. When zero-shot prompting the generation of Lean 4 code for natural language conjectures, we often faced syntactical errors in the output code containing a mix of Lean 3 and Lean 4 syntax. To mitigate this, we rely on in-context examples [4].

For both models, following [13], we generate multiple formalizations until the first type-checks. Specifically, we query the LLM to generate a formalization, pass

it to the Lean REPL [9], and retry if the formalization is erroneous, as shown in Figure 2. We retry up to $F = 10$ times, balancing compute and success rates [13]. If no formalization is possible, we skip this natural language conjecture and exclude it from the rest of our pipeline.

We perform this formalization for each generated conjecture individually, parsing them using the format described above.

Naming collision When formalizing many conjectures, we faced the issue of theorem name collisions. Multiple conjectures might receive the same name from our formalization component. If these conjectures are proven and loaded in the same Lean environment, Lean fails, as the name is already in use. To mitigate this, two possibilities arise.

1. Using a separate namespace for each formalized conjecture
2. Introducing a unique affix per theorem that is added to the name

Both variants work equally well and have their implementation challenges. Namespaces are Lean's native way to support the same names in different contexts. We therefore decided to use namespaces to resolve naming collisions. We introduce one namespace per theorem, assigning namespace names via UUID.

Proving To prove each of the extracted conjectures, we rely on two pre-existing ATP methods. Initial experiments are performed with ReProver Tacgen, an encoder-decoder model for proof-step generation developed by [24]. ReProver supports premises natively, so it does not require the hypothesis injection described below, and thereby serves as a good starting point for our tests.

Later, we generate whole proofs with Self-Play Theorem Prover by [3] instead. For each conjecture, we sample $T = 32$ possible proofs from Self-Play Theorem Prover or $T = 32$ proof steps from ReProver, respectively. For both models, we enhance proof attempts with similar proven conjectures.

Premise selection We follow [18] in selecting the most relevant proven conjectures to facilitate further proving. When proving a conjecture, we embed its initial proof state using the ReProver retrieval encoder [24] and retrieve the $k = 10$ nearest premises for it, measured by comparing the cosine similarity. Note that Self-Play Theorem Prover was not trained to deal with additional premises. Simply adding the premises before the theorem to be proven resulted in detrimental performance in our preliminary tests. To nonetheless inject premises, we consider three strategies:

1. Explicitly adding tactics such as and for every premise at each proof step. Although straightforward, this method significantly enlarges the search space, impacting efficiency and overall proof rates with the same compute.
2. Converting each premise into a statement embedded within the proof for the overarching statement. This approach enables the LLM used in ATP to select the right premises by adding them to the prompt in a format usable even by whole-proof generation models. However, the resulting proofs are

significantly longer, as each proof might contain up to k additional subproofs of similar length. Lean only requires the existence of a term of a particular type, rather than the full proof given in the statement.
3. Transforming proven premises into hypotheses by extracting their types and appending the types directly to the theorem statement as hypotheses, which we term *hypothesis injection*. This provides the essential information that such a proof term exists, without exposing unnecessary details.

We choose the third variant as it efficiently provides sufficient information for ATP tools to guide the search space by letting it choose when to apply hypotheses, without unnecessary lengthy proofs.

Internally, hypothesis injection uses the pretty-printed type of a declaration and injects this as Lean syntax in the theorem statement. Over all tests combined, we noticed one edge-case where using the pretty-printed string fails. Definite integrals $\int_a^b f(x)dx$, with the correct syntax in Lean as are pretty-printed as . Note the missing parentheses and incorrect indentation.

Proof checking We execute the generated theorem with its proof in the Lean REPL [9]. The proof-step generation interface is used for ReProver and the whole-proof generation interface for Self-Play Theorem Prover. If unsolved goals remain or an error is encountered, the proof candidate is considered a failure.

We store all proven conjectures in a Lean project and run to guarantee the correctness in case of implementation errors in the Lean REPL. To utilize Lean's caching mechanism for faster build times, we place each conjecture in a separate file and import these into the project's main file.

If a conjecture could not be proven, it is re-enqueued for proof search. When next attempted, other new conjectures will have been proven and might be selected in the premise selection step, making the conjecture easier to prove.

Hypothesis rejection Similarly to [22], we use hypothesis rejection to filter conjectures with invalid assumptions before attempting a proof. We replace the conclusion of each conjecture with `False` and attempt a proof for `False` using the assumptions. If a proof is found, we discard the conjecture.

Technical details We open-source an implementation merging conjecture generation, autoformalization, and ATP via message queueing. It supports both whole-proof generation and proof-step generation models, searching via HyperTreeProofSearch [8], and online reinforcement learning for autoformalization and ATP through the interplay of various components, see Figure 3. With this, we are the first open-source implementation of online reinforcement learning for ATP and online reinforcement learning for autoformalization. Specific details for individual parts of the implementation can be found in the appendix.

While we evaluate conjecture extraction, the same implementation can also be used directly for online training of ATP models, without extracting any conjectures. Our evaluations show promising results, but should mostly be considered preliminary. We encourage researchers to explore more possibilities of the

new pipeline, hoping our contribution will facilitate open-source AI for formalization.

4 Evaluation

As conjecture extraction can be used with any ATP tool, we test it with hammers, proof-step, and whole-proof generation models.

DSP comparison Starting the conjecture extraction evaluations, we test it using the same naive hammers as [1]: , , and , as the only possible proof steps. For DSP, [1] similarly use GPT-4o [6] to generate one proof sketch per problem. Our setup therefore allows for direct comparison against their results with DSP in Lean.

Conjecture extraction **significantly outperforms DSP** as the current state of the art for proof autoformalization, achieving an **absolute increase of 11.2%** for proof rate on MiniF2F test. DSP errors primarily stem from the LLM not adhering to Lean 4 syntax, resulting in proof-sketch formalization failures. Remarkably, even a naive application of to each conjecture without any LLM generation performs similar to DSP.

Table 1. Proof rates of DSP, conjecture extraction, and the direct application of on the MiniF2F benchmark [26]. We sample one proof sketch and extract conjectures once, respectively.

Method	Validation	Test
Aesop	11.5%	12.7%
Pantograph (DSP)	12.7%	14.7%
Conjecture extraction	**21.3**%	**25.9**%

Subsequently, we compare conjecture extraction with a direct application of the ATP tool for proof-step and whole-proof generation models, discarding DSP due to the low proof sketch success ratio.

Proof-step generation We test proof-step generation using ReProver [24] on PutnamBench [16]. Conjecture extraction using ReProver [24] **solves 2 PutnamBench problems**, **improving over the 0 problems by the default ReProver** [16]. The only better-performing proof-step generation models are InternLM2-StepProver [21] and InternLM2.5-StepProver [20].

ReProver's parameter count is 300 million [23], compared to 7 billion for both InternLM models [21]. We highlight this difference in parameter counts in Figure 4. Importantly, **all proof-step models outperforming our approach are more than** 20× **larger**. Beyond proof-step generation, conjecture extraction is the first proof autoformalization method to work on whole-proof generation, which we evaluate next.

Whole-proof generation For whole-proof generation, we test conjecture extraction with Self-Play Theorem Prover [3] on MiniF2F [26]. In our tests, Self-Play Theorem Prover without conjecture extraction proves 135 MiniF2F problems with pass@32. We extract conjectures five times and generate 32 proof candidates for each conjecture, creating a pass@5×32 setting.

Figure 5 depicts the conjecture counts at the individual stages. 1192 conjectures are extracted and formalized within the five rounds, of which 200 are successfully proven. Twelve conjectures are rejected using hypothesis rejection. After five rounds of conjecture extraction, we cumulatively prove 144 MiniF2F problems. Our results indicate that **repeated conjecture extraction continuously boosts performance** of pre-existing whole-proof generation models, as shown in Figure 5. We attribute this to different natural language proof steps formed from the same informal proof and different formalizations of similar proof steps, which might be easier to prove.

5 Conclusion

In this work, we have presented *conjecture extraction*, an alternative proof autoformalization method that does not rely on proof sketches. Thereby, conjecture extraction can be used with whole-proof generation models. Our preliminary experiments demonstrate that conjecture extraction substantially outperforms DSP in Lean 4, achieving an 11.2% absolute improvement in the pass@1 proof rate on the MiniF2F test benchmark.

Iteratively extracting and proving conjectures leads to a continuous increase in the number of successfully proven theorems. We open-source an implementation of our pipeline, which combines conjecture generation, autoformalization, and ATP with online reinforcement learning, and can be used for large-scale training and proof autoformalization testing. By providing this framework, we hope to facilitate further research and development in the open-source AI community for formal mathematics. Ultimately, conjecture extraction represents a promising new direction for tackling the challenges of large-scale proof autoformalization, aimed at aiding mathematicians working on formalization projects.

References

1. Aniva, L., Sun, C., Miranda, B., Barrett, C., Koyejo, S.: Pantograph: A Machine-to-Machine Interaction Interface for Advanced Theorem Proving, High Level Reasoning, and Data Extraction in Lean 4. In: Gurfinkel, A., Heule, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 104–123. Springer Nature Switzerland, Cham (2025)
2. Azerbayev, Z., Piotrowski, B., Avigad, J.: ProofNet: A benchmark for Autoformalizing and Formally Proving Undergraduate-Level Mathematics. Poster at the MATH-AI: Toward Human-Level Mathematical Reasoning Workshop, 36$^{\text{th}}$ Conference on Neural Information Processing Systems (NeurIPS 2022) (2022)

3. Dong, K., Ma, T.: Beyond limited data: Self-play llm theorem provers with iterative conjecturing and proving. In: The Forty-Second International Conference on Machine Learning (Jul 2025)
4. Dong, Q., Li, L., Dai, D., Zheng, C., Ma, J., Li, R., Xia, H., Xu, J., Wu, Z., Chang, B., Sun, X., Li, L., Sui, Z.: A survey on in-context learning. In: Al-Onaizan, Y., Bansal, M., Chen, Y.N. (eds.) Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing. pp. 1107–1128. Association for Computational Linguistics, Miami, Florida, USA (Nov 2024)
5. Gonen, H., Iyer, S., Blevins, T., Smith, N., Zettlemoyer, L.: Demystifying prompts in language models via perplexity estimation. In: Bouamor, H., Pino, J., Bali, K. (eds.) Findings of the Association for Computational Linguistics: EMNLP 2023. pp. 10136–10148. Association for Computational Linguistics, Singapore (Dec 2023)
6. Hurst, A., Lerer, A., Goucher, A.P., Perelman, A., Ramesh, A., Clark, A., Ostrow, A.J., Welihinda, A., Hayes, A., Radford, A., Mądry, A., Baker-Whitcomb, A., Beutel, A., Borzunov, A., Carney, A., Chow, A., Kirillov, A., Nichol, A., Paino, A., Renzin, A., Passos, A.T., Kirillov, A., Christakis, A., Conneau, A., Kamali, A., Jabri, A., Moyer, A., Tam, A., Crookes, A., Tootoochian, A., Tootoonchian, A., Kumar, A., Vallone, A., Karpathy, A., Braunstein, A., Cann, A., Codispoti, A., Galu, A., Kondrich, A., Tulloch, A., Mishchenko, A., Baek, A., Jiang, A., Pelisse, A., Woodford, A., Gosalia, A., Dhar, A., Pantuliano, A., Nayak, A., Oliver, A., Zoph, B., Ghorbani, B., Leimberger, B., Rossen, B., Sokolowsky, B., Wang, B., Zweig, B., Hoover, B., Samic, B., McGrew, B., Spero, B., Giertler, B., Cheng, B., Lightcap, B., Walkin, B., Quinn, B., Guarraci, B., Hsu, B., Kellogg, B., Eastman, B., Lugaresi, C., Wainwright, C., Bassin, C., Hudson, C., Chu, C., Nelson, C., Li, C., Shern, C.J., Conger, C., Barette, C., Voss, C., Ding, C., Lu, C., Zhang, C., Beaumont, C., Hallacy, C., Koch, C., Gibson, C., Kim, C., Choi, C., McLeavey, C., Hesse, C., Fischer, C., Winter, C., Czarnecki, C., Jarvis, C., Wei, C., Koumouzelis, C., Sherburn, D., Kappler, D., Levin, D., Levy, D., Carr, D., Farhi, D., Mely, D., Robinson, D., Sasaki, D., Jin, D., Valladares, D., Tsipras, D., Li, D., Nguyen, D.P., Findlay, D., Oiwoh, E., Wong, E., Asdar, E., Proehl, E., Yang, E., Antonow, E., Kramer, E., Peterson, E., Sigler, E., Wallace, E., Brevdo, E., Mays, E., Khorasani, F., Such, F.P., Raso, F., Zhang, F., Lohmann, F.v., Sulit, F., Goh, G., Oden, G., Salmon, G., Starace, G., Brockman, G., Salman, H., Bao, H., Hu, H., Wong, H., Wang, H., Schmidt, H., Whitney, H., Jun, H., Kirchner, H., Pinto, H.P.d.O., Ren, H., Chang, H., Chung, H.W., Kivlichan, I., O'Connell, I., O'Connell, I., Osband, I., Silber, I., Sohl, I., Okuyucu, I., Lan, I., Kostrikov, I., Sutskever, I., Kanitscheider, I., Gulrajani, I., Coxon, J., Menick, J., Pachocki, J., Aung, J., Betker, J., Crooks, J., Lennon, J., Kiros, J., Leike, J., Park, J., Kwon, J., Phang, J., Teplitz, J., Wei, J., Wolfe, J., Chen, J., Harris, J., Varavva, J., Lee, J.G., Shieh, J., Lin, J., Yu, J., Weng, J., Tang, J., Yu, J., Jang, J., Candela, J.Q., Beutler, J., Landers, J., Parish, J., Heidecke, J., Schulman, J., Lachman, J., McKay, J., Uesato, J., Ward, J., Kim, J.W., Huizinga, J., Sitkin, J., Kraaijeveld, J., Gross, J., Kaplan, J., Snyder, J., Achiam, J., Jiao, J., Lee, J., Zhuang, J., Harriman, J., Fricke, K., Hayashi, K., Singhal, K., Shi, K., Karthik, K., Wood, K., Rimbach, K., Hsu, K., Nguyen, K., Gu-Lemberg, K., Button, K., Liu, K., Howe, K., Muthukumar, K., Luther, K., Ahmad, L., Kai, L., Itow, L., Workman, L., Pathak, L., Chen, L., Jing, L., Guy, L., Fedus, L., Zhou, L., Mamitsuka, L., Weng, L., McCallum, L., Held, L., Ouyang, L., Feuvrier, L., Zhang, L., Kondraciuk, L., Kaiser, L., Hewitt, L., Metz, L., Doshi, L., Aflak, M., Simens, M., Boyd, M., Thompson, M., Dukhan, M., Chen, M., Gray, M., Hudnall, M., Zhang, M., Aljubeh, M., Litwin, M., Zeng, M., Johnson, M., Shetty,

M., Gupta, M., Shah, M., Yatbaz, M., Yang, M.J., Zhong, M., Glaese, M., Chen, M., Janner, M., Lampe, M., Petrov, M., Wu, M., Wang, M., Fradin, M., Pokrass, M., Castro, M., Castro, M.O.T.d., Pavlov, M., Brundage, M., Wang, M., Khan, M., Murati, M., Bavarian, M., Lin, M., Yesildal, M., Soto, N., Gimelshein, N., Cone, N., Staudacher, N., Summers, N., LaFontaine, N., Chowdhury, N., Ryder, N., Stathas, N., Turley, N., Tezak, N., Felix, N., Kudige, N., Keskar, N., Deutsch, N., Bundick, N., Puckett, N., Nachum, O., Okelola, O., Boiko, O., Murk, O., Jaffe, O., Watkins, O., Godement, O., Campbell-Moore, O., Chao, P., McMillan, P., Belov, P., Su, P., Bak, P., Bakkum, P., Deng, P., Dolan, P., Hoeschele, P., Welinder, P., Tillet, P., Pronin, P., Tillet, P., Dhariwal, P., Yuan, Q., Dias, R., Lim, R., Arora, R., Troll, R., Lin, R., Lopes, R.G., Puri, R., Miyara, R., Leike, R., Gaubert, R., Zamani, R., Wang, R., Donnelly, R., Honsby, R., Smith, R., Sahai, R., Ramchandani, R., Huet, R., Carmichael, R., Zellers, R., Chen, R., Chen, R., Nigmatullin, R., Cheu, R., Jain, S., Altman, S., Schoenholz, S., Toizer, S., Miserendino, S., Agarwal, S., Culver, S., Ethersmith, S., Gray, S., Grove, S., Metzger, S., Hermani, S., Jain, S., Zhao, S., Wu, S., Jomoto, S., Wu, S., Shuaiqi, Xia, Phene, S., Papay, S., Narayanan, S., Coffey, S., Lee, S., Hall, S., Balaji, S., Broda, T., Stramer, T., Xu, T., Gogineni, T., Christianson, T., Sanders, T., Patwardhan, T., Cunninghman, T., Degry, T., Dimson, T., Raoux, T., Shadwell, T., Zheng, T., Underwood, T., Markov, T., Sherbakov, T., Rubin, T., Stasi, T., Kaftan, T., Heywood, T., Peterson, T., Walters, T., Eloundou, T., Qi, V., Moeller, V., Monaco, V., Kuo, V., Fomenko, V., Chang, W., Zheng, W., Zhou, W., Manassra, W., Sheu, W., Zaremba, W., Patil, Y., Qian, Y., Kim, Y., Cheng, Y., Zhang, Y., He, Y., Zhang, Y., Jin, Y., Dai, Y., Malkov, Y.: GPT-4o System Card (Oct 2024), arXiv:2410.21276 [cs]

7. Jiang, A.Q., Welleck, S., Zhou, J.P., Lacroix, T., Liu, J., Li, W., Jamnik, M., Lample, G., Wu, Y.: Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In: The Eleventh International Conference on Learning Representations (2023)
8. Lample, G., Lacroix, T., Lachaux, M.a., Rodriguez, A., Hayat, A., Lavril, T., Ebner, G., Martinet, X.: HyperTree Proof Search for Neural Theorem Proving. In: The Thirty-sixth Annual Conference on Neural Information Processing Systems (Oct 2022)
9. LeanProver-Community, T.: A read-eval-print-loop for Lean 4 (Jun 2025), https://web.archive.org/web/20250602110502/https://github.com/leanprover-community/repl
10. Lin, Y., Tang, S., Lyu, B., Wu, J., Lin, H., Yang, K., Li, J., Xia, M., Chen, D., Arora, S., et al.: Goedel-prover: A frontier model for open-source automated theorem proving. CoRR **abs/2502.07640** (Feb 2025)
11. Moura, L.d., Ullrich, S.: The Lean 4 Theorem Prover and Programming Language. In: Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings. p. 625–635. Springer-Verlag, Berlin, Heidelberg (2021)
12. Paulson, L.C.: Isabelle, Lecture Notes in Computer Science, vol. 828. Springer-Verlag, Berlin, Heidelberg (1994)
13. Poiroux, A., Weiss, G., Kunčak, V., Bosselut, A.: Improving autoformalization using type checking (Feb 2025), arXiv:2406.07222 [cs]
14. Polu, S., Han, J.M., Zheng, K., Baksys, M., Babuschkin, I., Sutskever, I.: Formal Mathematics Statement Curriculum Learning. In: The Eleventh International Conference on Learning Representations (May 2023)

15. Polu, S., Sutskever, I.: Generative Language Modeling for Automated Theorem Proving (Sep 2020), arXiv:2009.03393 [cs]
16. Tsoukalas, G., Lee, J., Jennings, J., Xin, J., Ding, M., Jennings, M., Thakur, A., Chaudhuri, S.: PutnamBench: Evaluating Neural Theorem-Provers on the Putnam Mathematical Competition. In: Globerson, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J., Zhang, C. (eds.) The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Dec 2024)
17. Wang, H., Unsal, M., Lin, X., Baksys, M., Liu, J., Santos, M.D., Sung, F., Vinyes, M., Ying, Z., Zhu, Z., Lu, J., Saxcé, H.d., Bailey, B., Song, C., Xiao, C., Zhang, D., Zhang, E., Pu, F., Zhu, H., Liu, J., Bayer, J., Michel, J., Yu, L., Dreyfus-Schmidt, L., Tunstall, L., Pagani, L., Machado, M., Bourigault, P., Wang, R., Polu, S., Barroyer, T., Li, W.D., Niu, Y., Fleureau, Y., Hu, Y., Yu, Z., Wang, Z., Yang, Z., Liu, Z., Li, J.: Kimina-Prover Preview: Towards Large Formal Reasoning Models with Reinforcement Learning (Apr 2025), arXiv:2504.11354 [cs]
18. Wang, H., Xin, H., Zheng, C., Liu, Z., Cao, Q., Huang, Y., Xiong, J., Shi, H., Xie, E., Yin, J., Li, Z., Liang, X.: LEGO-Prover: Neural Theorem Proving with Growing Libraries. In: The Twelfth International Conference on Learning Representations (May 2024)
19. Wu, Y., Jiang, A.Q., Li, W., Rabe, M.N., Staats, C.E., Jamnik, M., Szegedy, C.: Autoformalization with Large Language Models. In: The Thirty-sixth Annual Conference on Neural Information Processing Systems (Nov 2022)
20. Wu, Z., Huang, S., Zhou, Z., Ying, H., Wang, J., Lin, D., Chen, K.: InternLM2.5-StepProver: Advancing Automated Theorem Proving via Expert Iteration on Large-Scale Lean Problems (Oct 2024), arXiv:2410.15700 [cs]
21. Wu, Z., Wang, J., Lin, D., Chen, K.: Lean-GitHub: Compiling GitHub Lean repositories for a versatile Lean prover (Jul 2024), arXiv:2407.17227 [cs]
22. 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 (Aug 2024), arXiv:2408.08152
23. Xue, L., Barua, A., Constant, N., Al-Rfou, R., Narang, S., Kale, M., Roberts, A., Raffel, C.: ByT5: Towards a token-free future with pre-trained byte-to-byte models. Transactions of the Association for Computational Linguistics **10**, 291–306 (2022)
24. Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R.J., Anandkumar, A.: LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In: Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., Levine, S. (eds.) Advances in Neural Information Processing Systems. vol. 36, pp. 21573–21612. Curran Associates, Inc. (Dec 2023)
25. Ying, H., Wu, Z., Geng, Y., Wang, J., Lin, D., Chen, K.: Lean Workbook: A large-scale Lean problem set formalized from natural language math problems. In: The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Nov 2024)
26. Zheng, K., Han, J.M., Polu, S.: Minif2f: a cross-system benchmark for formal Olympiad-level mathematics. In: The Tenth International Conference on Learning Representations (Apr 2022)

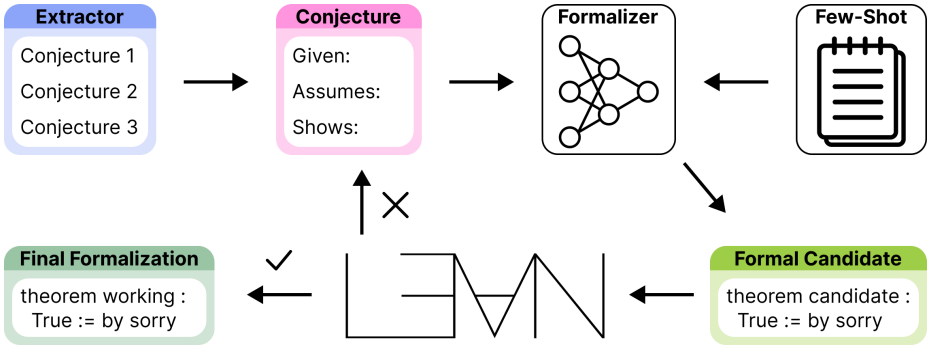

Fig. 2. We formalize each conjecture using one-shot in-context learning [4]. Lean [11] type-checks the conjecture using the Lean REPL [9]. If this fails, we retry the formalization of the same conjecture up to 10 times.

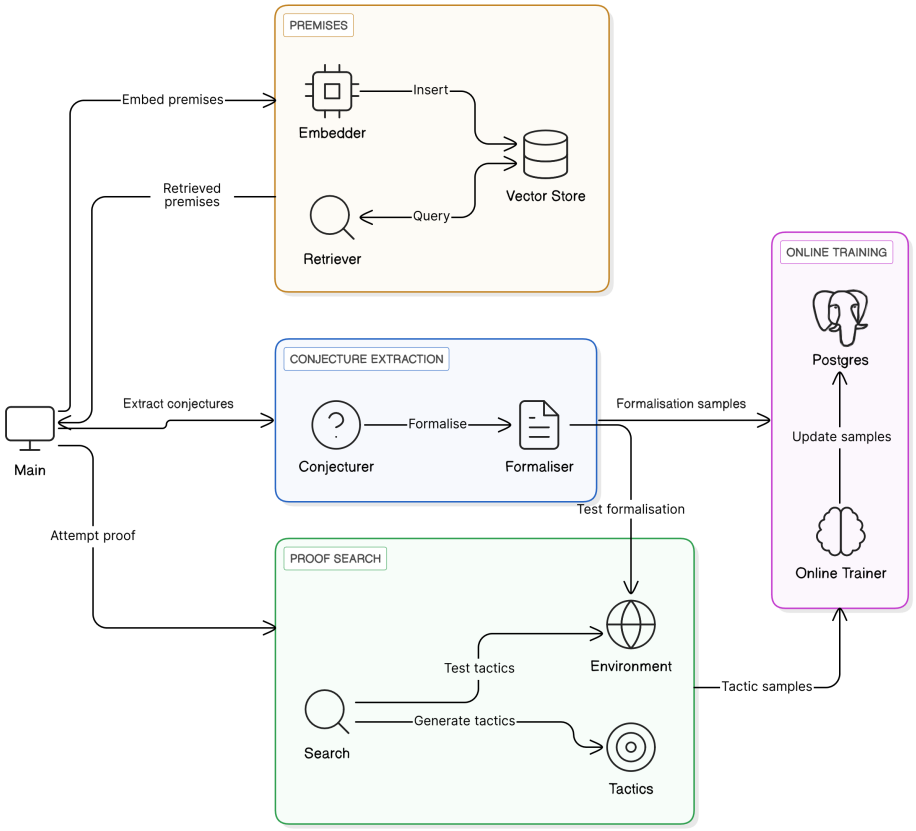

Fig. 3. A simplified overview of the open-sourced implementation.

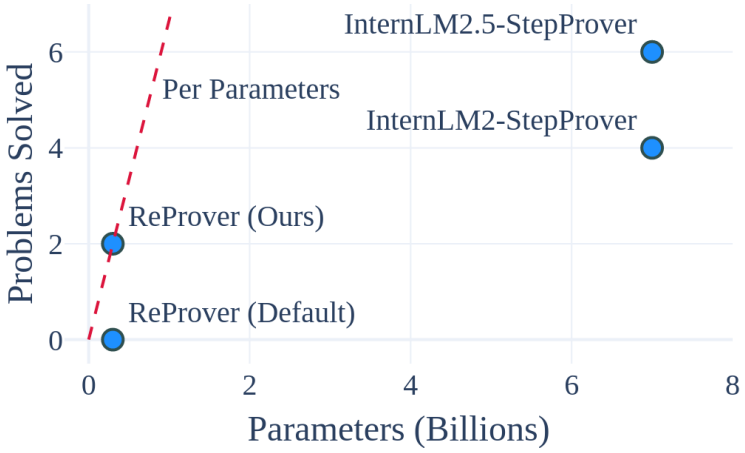

Fig. 4. Proven Putnam problems and parameter counts for proof-step generation models

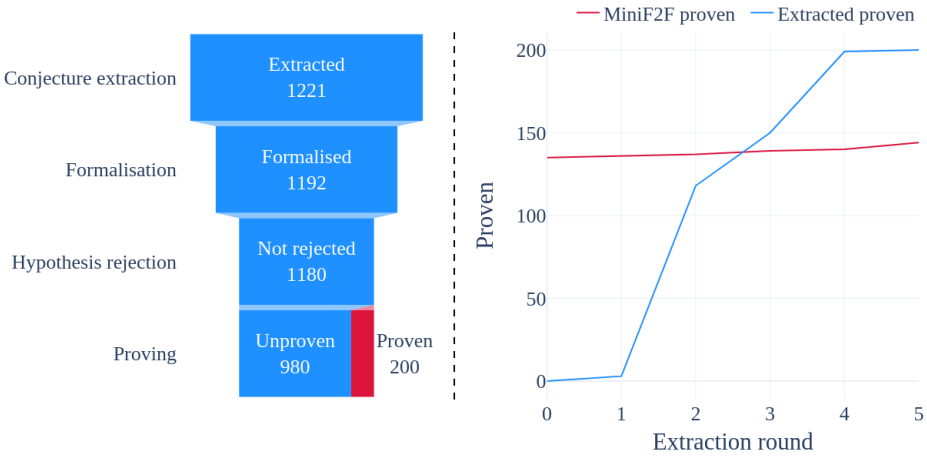

Fig. 5. Left: conjecture counts per pipeline step. Right: more MiniF2F problems are proven with repeated conjecture extraction.