VeriBench-FTP: A Formal Theorem Proving Benchmark in Lean 4 for Code Verification

Slim Barkallah
Stanford University
slimbark@stanford.edu

Srivatsava Daruru
Stanford University
srivatsavad@cs.stanford.edu

Brando Miranda
Stanford University
brando9@stanford.edu

Leni Aniva
Stanford University
aniva@stanford.edu

Allen Nie
Stanford University
anie@stanford.edu

Sanmi Koyejo
Stanford University
sanmi@stanford.edu

Abstract

Theorem proving in Lean 4 offers a promising avenue for advancing the reasoning capabilities of large language models. Evaluating current provers is crucial, as many achieve near-perfect accuracy on existing benchmarks such as MiniF2F [6], highlighting the need for novel evaluation tasks. We introduce VERIBENCH-FTP, a benchmark designed to assess formal theorem proving in Lean 4 through code verification. The task requires models to generate proofs for theorems that capture key aspects of program verification. Our benchmark consists of 857 theorems derived from 140 problems across five difficulty levels: 56 HumanEval problems, 41 foundational programming exercises, 10 classical algorithms, 28 security-critical programs adapted from real-world vulnerabilities, and 5 problems from the Python standard library. On our benchmark, Goedel-Prover V2-8B [3] achieves 39.56% Pass@32, highlighting the difficulty of the tasks. VERIBENCH-FTP provides a rigorous alternative to existing datasets, enabling more realistic evaluation of formal provers in Lean 4. VERIBENCH-FTP translates "theorem-proving ability" into a measurable route toward trustworthy code, advancing progress toward secure, dependable software infrastructure.

1 Introduction

Large language models (LLMs) have demonstrated remarkable success in automated theorem proving (ATP), achieving near-saturation performance on established mathematical benchmarks like MiniF2F [6] and ProofNet [7]. This progress signals a new frontier in formal reasoning [5]. However, this success also masks a critical gap: the predominant focus on competition-style mathematics leaves the capabilities of these models on software verification—a domain with profound real-world implications—largely untested. The saturation of existing benchmarks necessitates new, more challenging datasets that evaluate a different, and arguably more practical kind of reasoning.

We introduce VERIBENCH-FTP, a new benchmark in Lean 4 designed specifically for formal code verification. The dataset contains 857 theorems derived from 140 problems spanning five categories: HumanEval puzzles, foundational exercises, classical algorithms, real-world security vulnerabilities, and selected Python standard library programs. Each theorem captures correctness or safety properties of Lean functions corresponding to Python code, pushing models to reason about invariants, algorithmic behavior, and bug-prone implementations. On this benchmark, state-of-the-art provers—including DeepSeek-Prover and Claude with Draft Sketch Prove (DSP) [8] pipelines—achieve only 18–28% pass@1 accuracy, underscoring both the novelty and difficulty of the task. By shifting the focus from competition math to program reasoning, VERIBENCH-FTP

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: MATH-AI.

offers a complementary and realistic testbed for advancing the capabilities of next-generation theorem provers.

Our **contributions** are:

1. **Realistic proving challenges.** We release 857 theorems from a diverse set of 140 problems across five splits (HumanEval, Easy, CS, Security, RealCode), targeting correctness and safety properties.
2. **Establish strong baselines and headroom.** We conduct a thorough empirical evaluation of state-of-the-art provers, revealing their significant limitations (<40% pass@32) and exposing a critical gap between mathematical and formal code-reasoning capabilities.
3. **Broaden the evaluation landscape.** We broaden the evaluation landscape for automated reasoning by establishing formal software verification as a vital, complementary domain to traditional mathematical benchmarks.

2 Related Work

Formal theorem proving Benchmarks in Lean Recent works have proposed benchmarks for automated theorem proving in Lean, covering competition problems, undergraduate math, and large-scale formalizations. MINIF2F [6] provides a formal-to-formal benchmark of 488 competition problems from AMC, AIME, and IMO, split evenly between validation and test sets. FIMO [14] contains 149 Lean 3 problems formalized from IMO statements using GPT-4 and human verification. ProofNet [7] contributes 371 undergraduate-level theorem statements in Lean 3. PutnamBench [15] comprises 657 college-level mathematics problems across algebra, analysis, geometry, combinatorics, probability, and set theory, derived from the William Lowell Putnam Mathematical Competition. LeanDojo [16] extracts proofs directly from Lean's mathlib [17] and introduces a test set to evaluate retrieval-augmented provers. Recently, MathOlympiadBench [3] formalizes 360 Olympiad-level problems from Compfiles [19] and the IMOSL Lean 4 repository [20]. FormalMATH [4] offers 5,560 formalized problems from high school competitions and undergraduate mathematics. ProverBench [1] introduces 325 problems, including 15 AIME-style statements and additional problems from tutorials spanning high school to undergraduate mathematics.

Formal Benchmarks for Code Generation Recent benchmarks such as *Verina* [22] (189 Lean tasks jointly testing specification, code, and machine-checked proofs) and CLEVER [23] (161 Lean problems requiring both formal specifications and verified implementations) evaluate LLM-based code generation in Lean. The closest work to ours is VERIBENCH [9], a benchmark designed to evaluate the code verification capabilities of llms, requiring them to generate complete Lean 4 artifacts from reference Python programs or their docstrings. We used the VERIBENCH dataset in our data collection process. However, the VERIBENCH paper does not address theorem proving and our benchmark includes a larger set of problems. VERIBENCH-FTP provides a distinct platform for evaluating proof writing in Lean through code verification tasks that differ from existing benchmarks.

3 VeriBench-FTP

Overview. VERIBENCH-FTP is a benchmark designed to assess the theorem proving capabilities of large language models in Lean 4, specifically in the context of code verification. The theorems in our benchmark aim to verify one or more properties of a Lean function that corresponds to a Python implementation, ensuring that the function behaves as intended. In contrast to benchmarks such as MiniF2F [6], which focus primarily on mathematical problems formalized from competition-level tasks, VERIBENCH-FTP offers a complementary perspective by evaluating proof generation in Lean for code verification.

Concretely, VERIBENCH-FTP consists of 857 theorems divided into five subsets:

1. **HumanEval** – 387 theorems from 56 programming puzzles [18];
2. **EasySet** – 278 theorems extracted from 41 introductory logic and programming tasks;
3. **CSSet** – 68 theorems drawn from 10 classical data-structure and algorithm problems;
4. **SecuritySet** – 112 theorems taken from 28 examples of buffer overflows, privilege escalation, and race condition labs based on real code;

5. **RealCodeSet** – 12 theorems extracted from 5 Python standard library programs, used to test model performance on production-grade code.

This design covers a spectrum of tasks, from simple correctness theorems to more complex invariants, algorithmic properties, and real-world code verification.

Construction. To construct the benchmark, we based our work on VERIBENCH files, specifically the gold Lean implementation of their theorems [9]. The files contain unproved theorems with a set of definitions, examples, and variable declarations. The theorems in the files are generally unproved and contain the `sorry` symbol.

The construction process is divided into two phases. We extracted the theorems from the original files and isolate them. For each theorem, we kept only the necessary definitions and declarations, removing other theorems, examples, and comments. Our goal was to assume a minimal context to make the evaluation efficient and the data as clear as possible. A final human verification was performed. We formatted all the theorems uniformly, ensuring that each theorem ends with `:= sorry` on the same line as the final tokens of the example.

Finally, we compiled the entire dataset in Lean using `Mathlib` and `Aesop` as the only imports, to ensure that our benchmark is syntactically correct.

Examples. We provide some examples from the dataset; more examples can be found in Appendix A.

```
def myAdd : Nat →Nat →Nat := Nat.add
infixl:65 "++ "=> myAdd
def Pre (a b : Nat) : Prop := (0 ≤a) ∧(0 ≤b)
def right_identity_prop (n : Nat) : Prop := myAdd n 0 = n

theorem right_identity_thm (n : Nat) : right_identity_prop n := sorry
```

Listing 1: Example theorem statement from the VERIBENCH-FTP EasySet dataset.

```
open List
def min3 (a b c : Nat) : Nat :=
  min (min a b) c
def editDistanceAux [DecidableEq α] : List α→List α→Nat
  | [], [] => 0
  | [], ys => ys.length
  | xs, [] => xs.length
  | x :: xs, y :: ys =>
    if x = y then
      editDistanceAux xs ys
    else
      1 + min3
        (editDistanceAux xs (y :: ys))
        (editDistanceAux (x :: xs) ys)
        (editDistanceAux xs ys)
def editDistance [DecidableEq α] (s1 s2 : List α) : Nat :=
  editDistanceAux s1 s2
def Pre {α : Type*} (s1 s2 : List α) : Prop := True
def reflexivity_prop {α : Type*} [DecidableEq α] (s : List α) : Prop := editDistance s s = 0

theorem reflexivity_thm {α : Type*} [DecidableEq α] (s : List α) : reflexivity_prop s := sorry
```

Listing 2: Example theorem statement from the VERIBENCH-FTP CS Set dataset.

4 Evaluation

Models and Lean Compilation We evaluated three approaches: Aesop (a search-based ATP) [21], dedicated provers with prompting, and DSP (Draft–Sketch–Prove) using Claude models [10]. Pantograph [12] was used as the Lean interface to check code (see Appendix C for details).

Results The results in Tables 1 and 2 show that the models struggle to resolve most of the challenges, even with Pass@32. The best model achieved 28.94% on Pass@1 and 39.56% on Pass@32. Overall, the prover models outperformed the combination of Claude and DSP. Among DeepSeek models, V2 outperformed V1.5-RL, which outperformed SFT, with minor differences. This ranking is consistent with the results reported on MiniF2F for the same models [6].

Performance varies across splits. As expected, the *Easy* set achieved the highest success rates, as it contains comparatively simpler problems. In contrast, the *Advanced CS* set includes highly challenging questions. None of the models proved any theorems on real code problems (see Appendix B for an example).

Model + Prompting	**Easy**	**CS**	**Real**	**HE**	**Security**	**Pass@1**
Aesop	45/278	6/68	0/12	53/387	9/112	113/857 **13.19%**
Claude-3.5 Sonnet v1 (2024-06-20) + DSP	31/278	0/68	0/12	14/387	0/112	45/857 **5.25%**
Claude-3.7 Sonnet (2025-02-19) + DSP	81/278	2/68	0/12	67/387	6/112	156/857 **18.2%**
DeepSeek-ProverV1.5-SFT [2]+ prompting	59/278	2/68	0/12	57/387	15/112	133/857 **15.55%**
DeepSeek-ProverV2-7B [1]+ prompting	114/278	6/68	0/12	85/387	43/112	248/857 **28.94%**
STP [13] + prompting	69/278	3/68	0/12	71/387	21/112	164/857 **19.13%**
Goedel-Prover V2-8B [3] + prompting	109/278	9/68	0/12	76/387	31/112	225/857 **26.25%**

Table 1: Performance of different models on VERIBENCH theorem-proving tasks. The table shows results for LLMs evaluated under the Draft–Sketch–Prove (DSP) protocol (+DSP) [8], dedicated provers with a single prompt (+prompting), and Aesop (Traditional search-based ATP) [21]. All results are reported at pass@1. VERIBENCH-FTP splits : Easy Set (Easy), CS Set (CS), Real Python Code (Real), HumanEval Set (HE), Security Set (Security).

Model	**Easy**	**CS**	**Real**	**HE**	**Security**	**Pass@32**
DeepSeek-ProverV1.5-SFT	130/278	9/68	0/12	105/387	56/112	300/857 **35.00%**
DeepSeek-ProverV1.5-RL	132/278	8/68	0/12	107/387	57/112	304/857 **35.47%**
DeepSeek-ProverV2-7B	144/278	9/68	0/12	113/387	66/112	332/857 **38.74%**
STP	139/278	9/68	0/12	113/387	59/112	320/857 **37.34%**
Goedel-Prover V2-8B	156/278	9/68	0/12	113/387	61/112	339/857 **39.56%**

Table 2: Performance of various LLMs on VERIBENCH-FTP at pass@32. VERIBENCH-FTP splits : Easy Set (Easy), CS Set (CS), Real Python Code (Real), HumanEval Set (HE), Security Set (Security).

5 Discussion, Limitations, and Future Work

While VERIBENCH-FTP provides a novel testbed for theorem proving in code verification, it also has several limitations. First, the dataset is derived from a fixed set of problems, many of which are relatively short programs; this limits coverage of larger-scale software verification tasks such as modular reasoning, concurrency, or higher-order specifications. Second, our current evaluation focuses on pass@k accuracy with Lean compilation, which, while rigorous, does not fully capture proof quality, proof length, or generalization to novel proof styles.

These limitations suggest several promising directions for future work. Expanding the dataset to include larger and more diverse codebases—such as system libraries or verified kernels—would provide a stronger measure of scalability. Integrating richer property types, including temporal logic and probabilistic guarantees, could more closely reflect real-world verification needs. Finally, community-driven contributions and standardized leaderboards will be essential to track progress and stimulate advances in this emerging intersection of formal verification and AI.

Acknowledgments The authors would also like to thank the support from NSF 2046795 and 2205329, IES R305C240046, ARPA-H, the MacArthur Foundation, Schmidt Sciences, HAI, OpenAI, Microsoft, and Google.

References

[1] Zhenzhen Ren, Zhenyu Shao, Jialin Song, Hao Xin, Hao Wang, Wenqi Zhao, Lei Zhang, Zhiqiang Fu, Qingyu Zhu, Di Yang, Zhi-Fan Wu, Zhi Gou, Shuang Ma, Hongyu Tang, Yuxuan Liu, Wen Gao, Dongxu Guo, and Chuanqi Ruan. DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. *arXiv:2504.21801*, 2025.

[2] Hao Xin, Zhenzhen Ren, Jialin Song, Zhenyu Shao, Wenqi Zhao, Hao Wang, Bing Liu, Lei Zhang, Xiaoyu Lu, Qiang Du, Wen Gao, Qingyu Zhu, Di Yang, Zhi Gou, Zhi-Fan Wu, Feng Luo, and Chuanqi Ruan. DeepSeek-Prover-V1.5: Harnessing Proof Assistant Feedback for Reinforcement Learning and Monte-Carlo Tree Search. *arXiv:2408.08152*, 2024.

[3] Yilun Lin, Shuaijiang Tang, Bingjie Lyu, Jin-Hwa Chung, Hao Zhao, Liang Jiang, Yuchen Geng, Jiawei Ge, Jiarui Sun, Jiacheng Wu, Junjie Gesi, Xinchen Lu, David Acuna, Kaiyang Yang, Hao Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chenguang Jin. Goedel-Prover-V2: Scaling Formal Theorem Proving with Scaffolded Data Synthesis and Self-Correction. *arXiv:2508.03613*, 2025.

[4] Albert Yu, et al. FormalMATH: Benchmarking Formal Mathematical Reasoning of Large Language Models. *arXiv:2505.02735*, 2025.

[5] Bingxin Xin, Chenyu Zhang, Yichen Wang, et al. BFS-Prover: Mastering MiniF2F through Iterative Reinforcement Learning with Backward-Forward Search. *arXiv:2509.06493*, 2025.

[6] Kaiyu Zheng, Zhaoyang Chen, Jianshu Han, and Yiming Wu. MiniF2F: A cross-system benchmark for formal theorem proving. *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[7] Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. ProofNet: Autoformalizing and Formally Proving Undergraduate-Level Mathematics. *International Conference on Learning Representations (ICLR)*.

[8] Albert Q. Jiang, Chengxi Cui, Zeyu Zhang, Zhenyu Shao, Anant Abhyankar, Yuxuan Chi, Siddhartha Srinivasan, Wenhan Wang, Weizhe Chen, Yining Yang, Chen Li, Yikai Dai, Zhengyan Wang, Jiawei Lin, Junteng Liu, Chaowei Xiong, Michihiro Yasunaga, Quoc Le, Percy Liang, and Denny Zhou. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. *arXiv:2305.18058*, 2023.

[9] Brando Miranda, Zeyuan Zhou, Anson Nie, Eman Obbad, Laila Aniva, Kristian Fronsdal, Will Kirk, Dogan Soylu, Albert Yu, Yihan Li, and Sanmi Koyejo. VeriBench: End-to-End Formal Verification Benchmark for AI Code Generation in Lean 4. In *Workshop: AI for Math @ ICML 2025*, 2025.

[10] Anthropic. Claude: A family of large language models. Available at: `https://www.anthropic.com`, 2025.

[11] DeepSeek-AI. DeepSeek-V3: Scaling Open-Source Language Models with Mixture-of-Experts. *arXiv:2412.19437*, 2025.

[12] Leni Aniva and Chuyue Sun and Brando Miranda and Clark Barrett and Sanmi Koyejo Pantograph: A Machine-to-Machine Interaction Interface for Advanced Theorem Proving, High Level Reasoning, and Data Extraction in Lean 4 *arXiv:2410.16429*, 2024.

[13] Kaixuan Dong and Tengyu Ma. STP: Self-play LLM Theorem Provers with Iterative Conjecturing and Proving. *arXiv:2502.00212*, 2025.

[14] Changliu Liu, Juntong Shen, Hao Xin, Zheng Liu, Yifan Yuan, Hao Wang, Wenqi Ju, Chen Zheng, Yichao Yin, Liang Li, Mingyu Zhang, and Qiang Liu. FIMO: A Challenge Formal Dataset of Mathematical Olympiad Problems. *arXiv:2309.04295*, 2023.

[15] George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. PutnamBench: Evaluating Neural Theorem-Provers on the Putnam Mathematical Competition. *arXiv:2407.11214*, 2024.

[16] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *NeurIPS 2023 Datasets and Benchmarks Track*, 2023.

[17] Mathlib Community. The Lean mathematical library. Available at: `https://leanprover-community.github.io/`, 2020.

[18] Chen, Mark and Tworek, Jerry and Jun, Heewoo and Yuan, Qiming and Pinto, Henrique Ponde De Oliveira and Kaplan, Jared and Edwards, Harri and Burda, Yuri and Joseph, Nicholas and Brockman, Greg and others Evaluating large language models trained on code In *arXiv preprint arXiv:2107.03374*, 2021.

[19] David Renshaw. IMO Compendium in Lean. Available at: `https://dwrensha.github.io/compfiles/imo.html`, 2023.

[20] Mortar Sanjaya. IMO Solutions in Lean 4. Available at: `https://github.com/mortarsanjaya/IMOSLLean4`, 2023.

[21] Jannis Limperg and Asta Halkjær From. Aesop: White-Box Best-First Proof Search for Lean. In Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23), January 16–17, 2023, Boston, MA, USA. ACM, New York, NY, USA, 14 pages. Available at: `https://doi.org/10.1145/3573105.3575671`, 2023.

[22] Zhe Ye and Zhengxu Yan and Jingxuan He and Timothe Kasriel and Kaiyu Yang and Dawn Song VERINA: Benchmarking Verifiable Code Generation In *arXiv preprint arXiv:2505.23135*, 2025.

[23] Thakur, Amitayush and Lee, Jasper and Tsoukalas, George and Sistla, Meghana and Zhao, Matthew and Zetzche, Stefan and Durrett, Greg and Yue, Yisong and Chaudhuri, Swarat CLEVER: A Curated Benchmark for Formally Verified Code Generation In *arXiv preprint arXiv:2505.13938*, 2025.

A Dataset examples

```
def gcd_fun (a b : Nat) : Nat :=
  if b = 0 then a else gcd_fun b (a % b)
termination_by b
decreasing_by
  have h : b > 0 := Nat.pos_of_ne_zero (by assumption)
  have : a % b < b := Nat.mod_lt a h
  exact this
def myGcd : Nat →Nat →Nat
  | 0, b => b
  | a, 0 => a
  | a + 1, b + 1 =>
    if a < b then
      myGcd (a + 1) (b - a)
    else
      myGcd (a - b) (b + 1)
def Pre (a b : Nat) : Prop := True
def gcd_self_prop (a : Nat) : Prop := gcd_fun a a = a

theorem gcd_self_thm (a : Nat) : gcd_self_prop a := sorry
```

Listing 3: Example theorem statement from the VERIBENCH-FTP HumanEval Set.

```
structure ProgramSpec where
  name : String
  description : String
  deriving Repr, DecidableEq
structure SafeProgramExecution where
  prog : String
  args : List String
  deriving Repr, DecidableEq
def program_whitelist : List ProgramSpec := [
  { name := "cat", description := "Safe file reading" },
  { name := "ls", description := "Safe directory listing" },
  { name := "echo", description := "Safe text output" },
  { name := "wc", description := "Safe word/line counting" }
]
def is_program_whitelisted (prog_name : String) : Bool :=
  program_whitelist.any (fun spec => spec.name = prog_name)
def tokenize_arguments (args : String) : List String :=
  if args.trim.isEmpty then []
  else args.splitOn ""|>.filter (· ≠"")
def execute_program_safe (prog_name : String) (args : String) : Option SafeProgramExecution :=
```

```
  if is_program_whitelisted prog_name then
    let user_args := tokenize_arguments args
    some { prog := prog_name, args := user_args }
  else
    none
def Pre (_ _ : String) : Prop := True
def whitelist_enforcement_prop (prog_name : String) (args : String) : Prop :=
  (∃ exec, execute_program_safe prog_name args = some exec) →is_program_whitelisted prog_name = true

theorem whitelist_enforcement_thm (prog_name : String) (args : String) : whitelist_enforcement_prop
      prog_name args := sorry
```

Listing 4: Example theorem statement from the VERIBENCH-FTP Security dataset.

```
def bisect_right
    {α γ}
    (a : Array α)
    (x : γ)
    (lo : Nat := 0)
    (hi? : Option Nat := none)
    (key?: Option (α →γ) := none)
    [LT γ] [DecidableRel (· < ·: γ→γ→Prop)]
    [Coe αγ] [Inhabited α] :
    Nat := Id.run do
  let hi := hi?.getD a.size
  match key? with
  | none =>
    let mut loM := lo
    let mut hiM := hi
    while loM < hiM do
      let mid := (loM + hiM) / 2
      let y : γ:= (↑(a[mid]!) : γ)
      if x < y then
        hiM := mid
      else
        loM := mid + 1
    return loM
  | some key =>
    let mut loM := lo
    let mut hiM := hi
    while loM < hiM do
      let mid := (loM + hiM) / 2
      let y := key (a[mid]!)
      if x < y then
        hiM := mid
      else
        loM := mid + 1
    return loM
def insort_right
    {α γ}
    (a : Array α)
    (x : α)
    (lo : Nat := 0)
    (hi? : Option Nat := none)
    (key?: Option (α →γ) := none)
    [LT γ] [DecidableRel (· < ·: γ→γ→Prop)]
    [Coe αγ] [Inhabited α] :
    Array α:= Id.run do
  let idx :=
    match key? with
    | none => bisect_right a ((↑x : γ)) lo hi? (key? := none)
    | some key => bisect_right a (key x) lo hi? (key? := some key)
  let left := a.extract 0 idx
  let right := a.extract idx a.size
  return left ++ #[x] ++ right

def defaultProj {α : Type u} {γ : Type v} [Coe αγ] : α→γ:=
  fun a => (↑a : γ)
def Proj {α : Type u} {γ : Type v} (key? : Option (α →γ)) [Coe αγ] : α→γ:=
  key?.elim defaultProj id
def SortedSlice {α : Type u} {γ : Type v}
    (a : Array α) (lo hi : Nat) (proj : α→γ)
    [LE γ] [DecidableRel (· ≤·: γ→γ→Prop)] [Inhabited α] : Prop :=
  ∀i j, lo ≤i →i ≤j →j < hi →proj (a[i]!) ≤proj (a[j]!)
def Pre {α : Type u} {γ : Type v}
    (a : Array α) (lo hi : Nat) (key? : Option (α →γ))
    [LE γ] [DecidableRel (· ≤·: γ→γ→Prop)] [Coe αγ] [Inhabited α] : Prop :=
  lo ≤hi ∧hi ≤a.size ∧SortedSlice a lo hi (Proj key?)

theorem insort_preserves_sorted_slice
```

```
  {α γ}
  [LT γ] [DecidableRel (· < ·: γ→γ→Prop)]
  [LE γ] [DecidableRel (· ≤·: γ→γ→Prop)]
  [Coe αγ] [Inhabited α]
  (a : Array α) (xα : α) (lo : Nat) (hi? : Option Nat) (key? : Option (α →γ))
  (hpre : Pre a lo (hi?.getD a.size) key?) :
  let x := match key? with | none => (↑xα : γ) | some key => key xα
  let i := bisect_right a x lo hi? key?
  let a' := insort_right a xα lo hi? key?;
  SortedSlice a' lo (i+1) (Proj key?) ∧
  SortedSlice a' (i+1) (hi?.getD a.size + 1) (Proj key?) := sorry
```

Listing 5: Example theorem statement from the VERIBENCH-FTP Real code dataset.

B Failed Proof Example from the Real Code Split

The example in Listing 5 illustrates a theorem from the *Real Code* split. The context of the theorem provides a function `bisect_right`, which finds the position where a new element should be inserted into a sorted array. The function `insort_right` then inserts the element at the appropriate position. Additional helper functions introduce the projection concept, used to compute the metric for comparing elements in the array. The theorem to be proved states that if we insert an element at position i, the two resulting slices (before and after i) must remain sorted.

We provide two proof attempts generated by Goedel-Prover V2-8B [3]. In the first attempt (Figure 6), the errors stem from the model's use of undefined constants (`SortedSlice.not_gt`, `SortedSlice.ge`). In the second attempt (Figure 7), the prover applies `simp` without making progress.

```
theorem insort_preserves_sorted_slice
  {α γ}
  [LT γ] [DecidableRel (· < ·: γ→γ→Prop)]
  [LE γ] [DecidableRel (· ≤·: γ→γ→Prop)]
  [Coe αγ] [Inhabited α]
  (a : Array α) (xα : α) (lo : Nat) (hi? : Option Nat) (key? : Option (α →γ))
  (hpre : Pre a lo (hi?.getD a.size) key?) :
  let x := match key? with | none => (↑xα : γ) | some key => key xα
  let i := bisect_right a x lo hi? key?
  let a' := insort_right a xα lo hi? key?;
  SortedSlice a' lo (i+1) (Proj key?) ∧
  SortedSlice a' (i+1) (hi?.getD a.size + 1) (Proj key?) := by
    cases key?
    case none =>
      let x := (↑xα : γ)
      let i := bisect_right a x lo hi? none
      constructor
      ·refine SortedSlice.not_gt; dsimp only [SortedSlice, Prod]; intro j h₁ h₂ h₃; exact ?_
        have h₄ := SortedSlice.not_gt a' lo i j h₁ (Nat.succ_le_of_lt h₂) h₃
        exact Nat.succ_le_succ (by simpa [SortedSlice.not_gt] using h₄)
      ·exact SortedSlice.ge a' hi? (Nat.succ_le_of_lt hi!) h₁
    case some k =>
      have := hpre.2
      simp only [Pre, Pi.lt_def, Decidable.lt_iff_le_not_le] at this
      constructor
      ·exact SortedSlice.gt a' lo (i+1) k (fun j hj => this.1 j hj) (by simp only [true_and, SortedSlice.
           gt])
      ·exact SortedSlice.ge a' hi? (Nat.succ_le_of_lt hi!) h₁
```

Listing 6: Example proof of a statement from the VERIBENCH-FTP Real Code dataset, generated by Goedel-Prover V2-8B. The function declarations providing context have been removed for clarity.

```
theorem insort_preserves_sorted_slice
  {α γ}
  [LT γ] [DecidableRel (· < ·: γ→γ→Prop)]
  [LE γ] [DecidableRel (· ≤·: γ→γ→Prop)]
  [Coe αγ] [Inhabited α]
  (a : Array α) (xα : α) (lo : Nat) (hi? : Option Nat) (key? : Option (α →γ))
  (hpre : Pre a lo (hi?.getD a.size) key?) :
  let x := match key? with | none => (↑xα : γ) | some key => key xα
  let i := bisect_right a x lo hi? key?
  let a' := insort_right a xα lo hi? key?;
  SortedSlice a' lo (i+1) (Proj key?) ∧
  SortedSlice a' (i+1) (hi?.getD a.size + 1) (Proj key?) := by
 intro x i a'
 match hi? with
```

```
| none =>
  dsimp [i]
  match key? with
  | none =>
    dsimp [insort_right, Array.extract]
    simp
    constructor
    exact fun _ _ _ => le_trans (le_of_lt <| List.find?_lt _ _ _) i.prop
  | some key =>
    dsimp [insort_right, Array.extract]
    simp [show x = fun a →key a from rfl]
    constructor
    intro i' j' hi' hj' ij'
    simp at ij'
    cases hi', hj'
    exact rfl
    constructor
    ·rintro i' j' <hi', hj'> ij'
      simp at ij'
      cases hi', hj'
      simp [proj]
    ·exact fun i' j' ij' => i'.prop
| some hi =>
  dsimp [insort_right, Array.extract]
  simp
  constructor
  ·intro i' j' hi' hj' ij'
    refine List.find?_le_of_le _ <i'.prop, ij'>
    intro x hx
    exact lt_of_lt_of_le ij' (hi x).prop hx
  ·intro i' j' hi' hj' ij'
    simp at ij'
    cases hi', hj'
    exact rfl
```

Listing 7: Example proof of a statement from the VERIBENCH-FTP Real Code dataset, generated by Goedel-Prover V2-8B. The function declarations providing context have been removed for clarity.

C Models

We evaluated three approaches: Aesop (a search-based ATP) [21], dedicated provers with prompting, and DSP (Draft–Sketch–Prove) using Claude models [10]. For dedicated provers, we used sampling parameters temperature=1 and top_p=0.95.

- **DeepSeek-Prover-V1.5-SFT and DeepSeek-Prover-V1.5-RL**: Trained on proofs collected via expert iteration over a combination of public datasets [2].
- **DeepSeek-Prover-V2**: Trained using curriculum reinforcement learning, leveraging subgoal data extracted with DeepSeek-V3 [11] [1].
- **Godel-Prover-V2**: A family of models trained with scaffolded data synthesis, a self-correction loop, and model averaging [3]. In our evaluation, we used the variant without the self-correction loop. We employed the 7B model.
- **STP**: A model trained with self-play while simultaneously training a prover–conjecturer (generator of new statements)[13].
- **Claude Series**: We evaluated three models from the Claude Sonnet release. These were used with the DSP approach, which first drafts an informal solution, then generates a proof sketch, and finally attempts a complete proof [10].

D Prompts

We used vanilla prompts for provers and a chat-template prompt when required by the model. For Claude + DSP evaluation, we employed specific prompts for sketching and drafting.

```
Complete the following Lean 4 code:

```lean4
{}
```

Before producing the Lean 4 code to formally prove the given theorem, provide a detailed proof plan
     outlining the main proof steps and strategies.
```

```
The plan should highlight key ideas, intermediate lemmas, and proof structures that will guide the
     construction of the final formal proof.
```

Listing 8: Prompt for evaluating LLM provers with chat templates.

```
Complete the following Lean 4 code :

```lean4
```

Listing 9: Prompt for evaluating LLM provers without chat templates.

```
Draft an informal solution similar to the one below. The informal solution will be used to sketch a formal
      proof in the Lean 4 Proof Assistant. Here are some examples of informal problem solutions pairs:

Informal:
(*### Problem

Prove that for any natural number n, n + 0 = n.

### Solution

Consider any natural number n. From properties of addition, adding zero does not change its values. Thus,
     n + 0 = n.*)

Informal:
(*### Problem

Prove that for any natural number n, n + (m + 1) = (n + m) + 1.

### Solution

Consider any natural numbers n and m. From properties of addition, adding 1 to the sum of n and m is the
     same as first adding m to n and then adding 1. Thus, n + (m + 1) = (n + m) + 1.*)

Informal:
(*### Problem

Prove that for any natural number n and m, n + m = m + n.

### Solution

Consider any natural numbers n and m. We will do induction on n. Base case: 0 + m = m + 0 by properties of
      addition. Inductive step, we have n + m = m + n. Then (n + 1) + m = (n + m) + 1 = (m + n) + 1 = m +
     (n + 1). Thus, by induction, n + m = m + n, qed.*)

Informal:
(*### Problem

### Solution
```

Listing 10: Prompt template for the drafting phase using the DSP framework.

```
Translate the informal solution into a sketch in the formal Lean 4 proof. Add <TODO_PROOF_OR_HAMMER> in
     the formal sketch whenever possible. <TODO_PROOF_OR_HAMMER> will be used to call a automated theorem
     prover or tactic in Lean 4. Do not use any lemmas.Provide only one theorem in your formal sketch.
     Here are some examples:

Informal:
(*### Problem

Prove that for any natural number n, n + 0 = n.

### Solution

Consider any natural number n. From properties of addition, adding zero does not change its values. Thus,
     n + 0 = n.*)

Formal:
import Mathlib.Data.Nat.Basic
import Aesop

theorem n_plus_zero_normal : ∀n : Nat, n + 0 = n := by
  -- We have the fact of addition n + 0 = n, use it to show left and right are equal.
  have h_nat_add_zero: ∀n : Nat, n + 0 = n := <TODO_PROOF_OR_HAMMER>
  -- Combine facts with to close goal
  <TODO_PROOF_OR_HAMMER>

Informal:
(*### Problem
```
```

```
Prove that for any natural number n, n + (m + 1) = (n + m) + 1.

### Solution

Consider any natural numbers n and m. From properties of addition, adding 1 to the sum of n and m is the
     same as first adding m to n and then adding 1. Thus, n + (m + 1) = (n + m) + 1.*)

Formal:
import Mathlib.Data.Nat.Basic
import Aesop

theorem plus_n_Sm_proved_formal_sketch : ∀n m : Nat, n + (m + 1) = (n + m) + 1 := by
  -- We have the fact of addition n + (m + 1) = (n + m) + 1, use it to show left and right are equal.
  have h_nat_add_succ: ∀n m : Nat, n + (m + 1) = (n + m) + 1 := <TODO_PROOF_OR_HAMMER>
  -- Combine facts to close goal
  <TODO_PROOF_OR_HAMMER>

Informal:
(*### Problem

Prove that for any natural number n and m, n + m = m + n.

### Solution

Consider any natural numbers n and m. We will do induction on n. Base case: 0 + m = m + 0 by properties of
      addition. Inductive step, we have n + m = m + n. Then (n + 1) + m = (n + m) + 1 = (m + n) + 1 = m +
     (n + 1). Thus, by induction, n + m = m + n, qed.*)

Formal:
import Mathlib.Data.Nat.Basic
import Aesop

theorem add_comm_proved_formal_sketch : ∀n m : Nat, n + m = m + n := by
  -- Consider some n and m in Nats.
  intros n m
  -- Perform induction on n.
  induction n with
  | zero =>
    -- Base case: When n = 0, we need to show 0 + m = m + 0.
    -- We have the fact 0 + m = m by the definition of addition.
    have h_base: 0 + m = m := <TODO_PROOF_OR_HAMMER>
    -- We also have the fact m + 0 = m by the definition of addition.
    have h_symm: m + 0 = m := <TODO_PROOF_OR_HAMMER>
    -- Combine facts to close goal
    <TODO_PROOF_OR_HAMMER>
  | succ n ih =>
    -- Inductive step: Assume n + m = m + n, we need to show succ n + m = m + succ n.
    -- By the inductive hypothesis, we have n + m = m + n.
    have h_inductive: n + m = m + n := <TODO_PROOF_OR_HAMMER>
    -- 1. Note we start with: Nat.succ n + m = m + Nat.succ n, so, pull the succ out from m + Nat.succ n
         on the right side from the addition using addition facts Nat.add_succ.
    have h_pull_succ_out_from_right: m + Nat.succ n = Nat.succ (m + n) := <TODO_PROOF_OR_HAMMER>
    -- 2. then to flip m + S n to something like S (n + m) we need to use the IH.
    have h_flip_n_plus_m: Nat.succ (n + m) = Nat.succ (m + n) := <TODO_PROOF_OR_HAMMER>
    -- 3. Now the n & m are on the correct sides Nat.succ n + m = Nat.succ (n + m), so let's use the def
         of addition to pull out the succ from the addition on the left using Nat.succ_add.
    have h_pull_succ_out_from_left: Nat.succ n + m = Nat.succ (n + m) := <TODO_PROOF_OR_HAMMER>
    -- Combine facts to close goal
    <TODO_PROOF_OR_HAMMER>

Informal:
(*### Problem

{nl_problem}

### Solution

{nl_solution}*)

Formal:

### Problem


### Solution
```

Listing 11: Prompt template for sketch generation using the DSP framework.