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

Anonymous Author(s)
Affiliation
Address
email

Abstract

Theorem proving in Lean 4 offers a promising avenue for advancing the reasoning capabilities of LLMs. Evaluating current provers is crucial, as many achieve near-perfect accuracy on existing benchmarks such as MiniF2F, highlighting the need for more novel tasks. We introduce VERIBENCH-FTP, a benchmark designed to assess formal code verification knowledge in Lean 4. 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, DeepSeek-Prover-V2-8B [4] achieves 28% Pass@1, highlighting both the novelty and difficulty of the tasks . VERIBENCH provides a rigorous alternative to existing datasets, enabling more realistic evaluation of formal provers in Lean 4 on problems that go beyond their training distribution. VERIBENCH-FTP translates 'theorem-proving ability' into a measurable route toward trustworthy code, making progress oward 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]. This progress signals a new frontier in formal reasoning. 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) pipelines—achieve only 18–28% pass@1 accuracy, underscoring both the novelty and difficulty of the task [8]. By shifting the focus from competition math to program reasoning, VERIBENCH-FTP offers a complementary and realistic testbed for advancing the capabilities of next-generation theorem provers.

Our **contributions** are:

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

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 (<29% pass@1) 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 introduced a variety of benchmarks for evaluating automated theorem proving in Lean, spanning mathematical competition problems, undergraduate-level mathematics, and large-scale formalizations. MINIF2F (Zheng et al., 2021) provides a formal-to-formal benchmark of 488 competition problems from AMC, AIME, and IMO, split evenly between validation and test sets. FIMO (Liu et al., 2023) contains 149 Lean 3 problems formalized from IMO statements using GPT-4 and human verification. ProofNet (Azerbayev et al., 2023) contributes 371 undergraduate-level theorem statements in Lean 3. PutnamBench (Tsoukalas et al., 2024) comprises 657 college-level mathematics problems across algebra, analysis, geometry, combinatorics, probability, and set theory, derived from the William Lowell Putnam Mathematical Competition. LeanDojo (Yang et al., 2023) extracts proofs directly from Lean's mathlib (Mathlib Community, 2020) and introduces a test set to evaluate retrieval-augmented provers. Recently, MathOlympiadBench (Lin et al., 2025) formalizes 360 Olympiad-level problems from Compfiles and the IMOSL Lean 4 repository. FormalMATH (Yu et al., 2025) offers 5,560 formalized problems from high school competitions and undergraduate mathematics. ProverBench (Ren et al., 2025) introduces 325 problems, including 15 AIME-style statements and additional problems from tutorials spanning high school to undergraduate mathematics.

Formal Benchmarks for Code Generation The closest work to ours is VERIBENCH, a benchmark designed to evaluate the end-to-end code verification capabilities of large language models, requiring them to generate complete Lean 4 artifacts from reference Python programs or their docstrings. We used the VERIBENCH implementation and files in our data collection process. However, the VERIBENCH paper does not address theorem proving, while our benchmark includes a larger set of problems. VERIBENCH-FTP thus provides a distinct platform to evaluate the ability of LLMs to generate Lean proofs on problems unseen during training.

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, 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 derived from 56 standard programming puzzles from **?**;
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** – 121 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 `Aeosp` 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 -- insertions
  | xs, [] => xs.length -- deletions
  | x :: xs, y :: ys =>
    if x = y then
      editDistanceAux xs ys
    else
      1 + min3
        (editDistanceAux xs (y :: ys)) -- deletion
        (editDistanceAux (x :: xs) ys) -- insertion
        (editDistanceAux xs ys) -- substitution
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 We evaluated two types of methods: dedicated provers using prompting, and DSP (Draft–Sketch–Prove) with the model Claude family [10].

- **DeepSeek-Prover-V1.5-SFT and DeepSeek-Prover-V1.5-RL**: Trained on proofs collected via expert iteration over a combination of public datasets.
- **DeepSeek-Prover-V2**: Trained using curriculum reinforcement learning, leveraging subgoal data extracted with DeepSeek-V3 [11].
- **Godel-Prover-V2**: A family of models trained with scaffolded data synthesis, a self-correction loop, and model averaging. 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).
- **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.

Lean Compilation For the evaluation, we used PyPantograph [12] as the Lean interface to verify theorems and to apply the DSP method with Claude.

Results The results in Table 1 show that the models struggle to resolve most of the challenges, even with Pass@32. The best model achieved 28% on Pass@1 and 38% on Pass@32. Overall, the prover models outperformed the combination of Claude and DSP. Among the DeepSeek models, V2 performed better than V1.5-RL, which in turn outperformed the SFT model, with small differences. This ranking is consistent with the results reported on MiniF2F for the same models [TO DO].

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 succeeded in proving theorems on real code problems.

Model + Prompting	Easy	CS	Real	HE	Security	Pass@1
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 (**?**) + prompting	59/278	2/68	0/12	57/387	15/112	133/857 **15.55%**
DeepSeek-ProverV2-7B (**?**) + prompting	114/278	6/68	0/12	85/387	43/112	248/857 **28.94%**
Goedel-Prover V2-8B (**?**)+ 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, and Prove (DSP) protocol (+ DSP) (**?**) and dedicated provers evaluated using a standard single prompt (+ prompting). 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	–/387	61/112	226+/857 **TODO%**

Table 2: Performance of different models on VERIBENCH theorem-proving tasks. The table shows results for LLMs evaluated 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.

References

[1] Ren, Z. Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., Wu, Z. F., Gou, Z., Ma, S., Tang, H., Liu, Y., Gao, W., Guo, D., Ruan, C. (2025). DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. *arXiv preprint arXiv:2504.21801*. Available at: https://arxiv.org/abs/2504.21801

[2] Xin, H., Ren, Z. Z., Song, J., Shao, Z., Zhao, W., Wang, H., Liu, B., Zhang, L., Lu, X., Du, Q., Gao, W., Zhu, Q., Yang, D., Gou, Z., Wu, Z. F., Luo, F., Ruan, C. (2024). DeepSeek-Prover-V1.5: Harnessing Proof Assistant Feedback for Reinforcement Learning and Monte-Carlo Tree Search. *arXiv preprint arXiv:2408.08152*. Available at: https://arxiv.org/abs/2408.08152

[3] Lin, Y., Tang, S., Lyu, B., Wu, J., Lin, H., Yang, K., Li, J., Xia, M., Chen, D., Arora, S., Jin, C. (2025). Goedel-Prover: A Frontier Model for Open-Source Automated Theorem Proving. *arXiv preprint arXiv:2502.07640*. Available at: https://arxiv.org/abs/2502.07640

[4] Yu, A., et al. (2025). FormalMATH: A Large-Scale Dataset for Formal Mathematical Reasoning. *arXiv preprint*.

[5] Xin, B., Zhang, C., Wang, Y., et al. (2025). BFS-Prover: Mastering MiniF2F through Iterative Reinforcement Learning with Backward-Forward Search. *arXiv preprint arXiv:2509.06493*. Available at: https://arxiv.org/abs/2509.06493

[6] Zheng, K., Chen, Z., Han, J., Wu, Y. (2021). MiniF2F: A cross-system benchmark for formal theorem proving. *Advances in Neural Information Processing Systems (NeurIPS)*.

[7] Azerbayev, Z., Polu, S., Selsam, D., et al. (2023). ProofNet: Autoformalizing and Formally Proving Undergraduate-Level Mathematics. *International Conference on Learning Representations (ICLR)*.

[8] Jiang, A.Q., Cui, C., Zhang, Z., Shao, Z., Abhyankar, A., Chi, Y., Srinivasan, S., Wang, W., Chen, W., Yang, Y., Li, C., Dai, Y., Wang, Z., Lin, J., Liu, J., Xiong, C., Yasunaga, M., Le, Q., Liang, P., Zhou, D. (2023). Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. *arXiv preprint arXiv:2305.18058*. Available at: https://arxiv.org/abs/2305.18058

[9] Miranda, B., Zhou, Z., Nie, A., Obbad, E., Aniva, L., Fronsdal, K., Kirk, W., Soylu, D., Yu, A., Li, Y., Koyejo, S. (2025). VeriBench: End-to-End Formal Verification Benchmark for AI Code Generation in Lean 4. In *Workshop: AI for Math @ ICML 2025*.

[10] Anthropic. (2023). Claude: A family of large language models. Available at https://www.anthropic.com

[11] DeepSeek-AI. (2025). DeepSeek-V3: Scaling Open-Source Language Models with Mixture-of-Experts. *arXiv preprint arXiv:2412.19437*. Available at: https://arxiv.org/abs/2412.19437

[12] Miranda, B. (2025). PyPantograph: A Python Library for Evaluating Language Models on Lean 4 Proof Synthesis. *GitHub repository*. Available at: https://github.com/brando90/pypantograph

[13] Dong, K., Ma, T. (2025). STP: Self-play LLM Theorem Provers with Iterative Conjecturing and Proving. *arXiv preprint arXiv:2502.00212*. Available at: https://arxiv.org/abs/2502.00212

[14] Zheng, K., Chen, Z., Han, J., Wu, Y. (2021). MiniF2F: A cross-system benchmark for formal theorem proving. *Advances in Neural Information Processing Systems (NeurIPS)*.

[15] Liu, J., et al. (2023). FIMO: A Challenge Formal Dataset of Mathematical Olympiad Problems. *arXiv preprint*.

[16] Azerbayev, Z., Polu, S., Selsam, D., et al. (2023). ProofNet: Autoformalizing and Formally Proving Undergraduate-Level Mathematics. *International Conference on Learning Representations (ICLR)*.

[17] Tsoukalas, S., et al. (2024). PutnamBench: Evaluating Neural Theorem-Provers on the Putnam Mathematical Competition. *arXiv preprint*.

[18] Yang, K., et al. (2023). LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. *arXiv preprint*.

[19] Mathlib Community. (2020). The Lean mathematical library. Available at: https://leanprover-community.github.io/

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: An example of theorem from 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 -- Program not whitelisted
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: An example of theorem from Security set.

```
def _siftdown (heap : Array Int) (startpos pos : Nat) : Array Int :=
  let newitem := heap[pos]!
  let rec loop (h : Array Int) (pos : Nat) : Array Int :=
    if pos > startpos then
      let parentpos := (pos - 1) >>> 1
      let parent := h[parentpos]!
      if newitem < parent then
        let h' := h.set! pos parent
        loop h' parentpos
      else
        h.set! pos newitem
    else
      h.set! pos newitem
  loop heap pos
def heappush (heap : Array Int) (item : Int) : Array Int :=
  let heap1 := heap.push item
  _siftdown heap1 0 (heap1.size - 1)
def checkInvariant (h : Array Int) : Bool :=
  let n := h.size
  let rec go (i : Nat) : Bool :=
    if i ≥n then
      true
    else if i = 0 then
      go (i + 1)
    else
      let parentpos := (i - 1) >>> 1
      if h[parentpos]! ≤h[i]! then
        go (i + 1)
      else
        false
  go 0
def Pre (heap : Array Int) : Prop :=
  checkInvariant heap = true
def prop_invariant (heap : Array Int) (item : Int) : Prop :=
  checkInvariant (heappush heap item) = true

theorem invariant_thm (heap : Array Int) (item : Int) (hPre : Pre heap) :
  prop_invariant heap item := sorry
```

Listing 5: An example of theorem from Real code.

B 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 6: Prompt for evaluating LLM provers with chat templates.

```
Complete the following Lean 4 code :

```lean4
```

Listing 7: 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 8: 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 9: Prompt template for sketch generation using the DSP framework.