Lean Meets Theoretical Computer Science: Scalable Synthesis of Theorem Proving Challenges in Formal-Informal Pairs

Anonymous Authors[1]

Abstract

Formal theorem proving (FTP) stands at the forefront of LLM reasoning, yet existing datasets are largely limited in scope due to their dependence on manual curation. We identify theoretical computer science (TCS) as a novel testbed with promising potential, as many TCS problems can be algorithmically modeled and automatically verified, thereby enabling scalable formalization with rigorous formal-informal alignment guarantees. We showcase this approach via two TCS problem modules: the Busy Beaver (BB) challenge and the Mixed BooleanArithmetic (MBA) challenge. We evaluate leading reasoning models and finetuned theorem provers on our dataset, where the best-performing DeepSeekProver-v2-671B attains only 57.5% on our BB challenge and no more than 12% on our MBA challenge. These results reveal substantial reasoning gaps beyond conventional static benchmarks. We call on the Lean community to further study the potential of TCS problems in the formal reasoning domain where our approach enables fully automated synthesis of arbitrarily many problems in their strictly aligned formal-informal pairs that are universally **easy to verify** yet systematically **hard to prove**.

1. Introduction

While Large Reasoning Models (LRMs) (Besta et al., 2025) are evolving at an unprecedented speed, this promising progress is increasingly shadowed by a critical asymmetry: the advancing momentum of frontier models is quickly outpacing our ability to accurately gauge the boundary of their reasoning capability, as evidenced by the rapid saturation of legacy benchmarks such as MATH (Hendrycks et al., 2021) and GPQA (Rein et al., 2023).

Beyond conventional static benchmarks that evaluate on a final answer only, a natural next stage is to examine detailed machine proof to see if models are genuinely capable of making meaningful logical derivations. Formal Theorem Proving (FTP) (Chen et al., 2025) is emerging as a defining frontier for automating such proof validation by harnessing the power of interactive theorem provers such as Lean (de Moura & Ullrich, 2021) and Isabelle (Nipkow et al., 2002).

As a result, there's a critical need for large-scale, high-quality datasets that bridge theorem proving problems between their formal-informal forms. Previous work such as MiniF2F (Zheng et al., 2022), ProofNet (Azerbayev et al., 2023), PutnamBench (Tsoukalas et al., 2024), and CombiBench (Liu et al., 2025) leveraged labor-intensive expert curation to formalize problems from math competitions and college courses, yet such public data sources may suffer from inter-annotator inconsistency and data contamination of the training corpora (Magar & Schwartz, 2022; Dong et al., 2024), where even slight data leakage can inflate model performance by up to 30% points (Dekoninck et al., 2024).

In light of these concerns, Theoretical Computer Science (TCS) emerges as a promising new testbed where many problems can be **algorithmically modeled and verified**, which enables scalable fully automated problem formalization using a modularized TCS problem such as the Halting Problem of Turing Machines. Furthermore, we can adjust the difficulty of synthesized problems by tuning complexity parameters such as the state count of a Turing Machine or the variable count in an algebraic expression. This parameterized setting also enables us to synthesize fresh problems from a infinitely vast problem space, which acts as a robust mitigation against data contamination. It's particularly worth noting that our synthesis does not rely on any LLM-generated content in any steps, which would be unavoidably impacted by language model hallucination and potential regurgitation of training data.

In light of these concerns, our main contribution in this paper is threefold:

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

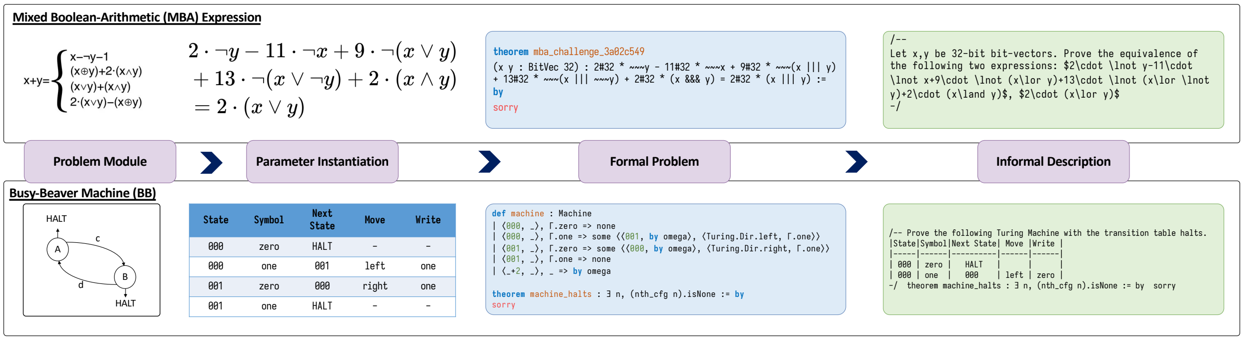

Figure 1. Complete Pipeline of our Synthesis Framework: We start with a problem module that can be instantiated by a rule-based system to generate theorem proving problem description in Lean code shown in the blue block and the same description in Markdown shown in the green blocks.

- We propose **a contamination-resistant synthesis framework with adjustable difficulty** that generates arbitrarily many theorem proving challenges in their rigorously aligned formal-informal pairs by leveraging TCS-inspired problem modules as shown in Figure 1.
- We evaluate a variety of frontier LRMs and LLM-based theorem provers. The best-performing DeepSeek-Prover-v2-671B attained best performance of 57.5% on our BB challenge and only 12% on our MBA challenge respectively, yet other theorem provers such as Kimina-Prover-Preview (Wang et al., 2025) and Goedel-Prover-SFT (Lin et al., 2025) struggle to resolve any of our synthesized problems, which may hint at how base model scaling could enhance capability of LLM-based theorem provers.
- We design **a novel step-level task decomposition setting** to gauge if models can effectively use a customized **Out-Of-Distribution** library of Lean lemma to resolve unseen problems, where best-performing model, OpenAI-o4-mini, showed 98.88% success rate. This reveals the critical bottleneck of the proof task lies in whole-proof level understanding and strategic planning as opposed to step-level problem-solving.

2. Related Work

Formal Theorem Proving (FTP). LLM has demonstrated promising potential in advancing the field of mathematics (Polu & Sutskever, 2020) with the help of interactive theorem provers as endorsed by many world-leading mathematicians such as Peter Scholze (Scholze, 2022) and Terence Tao (Tao, 2025). By integrating FTP in various contexts such as LLM-based Agents (Li et al., 2024a) and Retrieval-Augmented Generation (RAG) (Zayyad & Adi, 2024), many downstream applications can be empowered such as code copilot (Murphy et al., 2024) and operations research (Yang et al., 2024).

Autoformalization. Autoformalization refers to the process of translating natural language into formal language, which is inherently challenging for human due to the complex syntax of FTP languages. While the synthetic data approach (Wu et al., 2021; Huang et al., 2024; Ying et al., 2024) attempted to leverage LLMs for this process (Li et al., 2024b; Wu et al., 2022; Lu et al., 2024), current progress is largely limited by expert annotation to proofread and correct LLM-generated formal contents, which is labor-intensive and exposed to inter-annotator inconsistencies. Therefore, we aim to automate the entire process of (in)formalization via a template-based approach to ensure rigorous alignment, rendering manual curation *an optional quality check* as opposed to *an expensive necessity*.

Benchmark Contamination. In light of rising concerns about whether our benchmarks reflect genuine reasoning capability of LLMs (Magar & Schwartz, 2022; Dong et al., 2024), various attempts were made to detect and mitigate data contamination (Dekoninck et al., 2024). For instance, LiveBench (White et al., 2024) aims to mitigate contamination by refreshing benchmark questions every 6 months, and MixEval (Ni et al., 2024) attempted to remove preference bias by adopting a mixture of benchmarks. Nevertheless, data contamination remains a persistent issue so long as the questions are sourced from the public Internet at large, including synthetic data reliant on LLM generation.

3. Methodology

3.1. The Busy Beaver (BB) Challenge

We showcase our approach based on the halting problem of the famous Busy Beaver Machine. Busy Beaver machines are simple yet powerful computation models as any Turing-complete problems can be reduced to the Busy Beaver problem. Many research questions in modern mathematics, such as the Collatz Conjecture (Michel, 1993) and the famous Goldbach Conjecture (Aaronson, 2020) have been proven to be reducible to BB(6) and BB(27) respectively.

The Problem Module. A busy beaver machine is an N-state, 2-symbol Turing machine, where the Busy Beaver function $\textbf{BB}(N)$ (Harland, 2022) is defined as the maximum number of steps an N-state halting machine can execute before halt. In this challenge we instantiate with $N \leq 5$ as BB(5) is currently the limit of frontier mathematics research (Aaronson, 2020), we note that this parameter N can be any natural number in principle. By setting this parameter N we enable adjustable difficulty of synthesized problems and the number of machines available grows exponentially with the N as detailed in Table 1, thereby ensuring the infinite scalability of our synthesis.

n	Number of Machines
1	64
2	20,736
3	16,777,216
4	4,294,967,296

Table 1. Number of Busy Beaver (BB) Machines for Increasing Values of N.

Challenge definition. We ask models to prove whether a given BB machine described by a state transition table halts or not.

Generate easy-to-verify ground-truth. In order to know exactly whether a given Busy Beaver machine halts, we execute the machine for $BB(N) + 1$ steps, where $BB(N)$ is the Busy Beaver number for N-state machines, defined as the maximum steps for any N-state Turing Machine to run without halting, The numerical value of $\textbf{BB}(N)$ ($N \leq 5$) are proven results in mathematics (Xu, 2024). BB Machines that reaches halt state within $BB(N) + 1$ steps are halting, otherwise non-halting by the definition of $BB(N)$.

Synthesize hard-to-prove challenge. We instantiate a Busy Beaver machine with a state transition table, which is then used to generate both Lean and Markdown statements using an expert-defined template. This synthetic framework ensures scalable generation of arbitrarily many proof questions in rigorously aligned formal-informal pairs as illustrated in Figure 1. Specifically, each BB machine is uniquely defined by a State Transition Table containing tuples in the form State × Symbol × State × Move × Symbol, where State $\in \{0, 1, \ldots, N-1, \text{HALT}\}$, Symbol $\in \{\texttt{zero}, \texttt{one}\}$, Move $\in \{\texttt{left}, \texttt{right}\}$, indicating a BB machine at a certain state with head pointing to certain symbol needs to change to another state, write a symbol to the original head position and move to left or right. To generate a N state BB machine, we randomly sample the remaining elements for all $2N$ transition tuples.

With the state transition table, we apply an expert-defined template to generate the proof challenge in Lean. The template contains all necessary definitions and two blanks to fill in for the machine definition and the halting/non-halting theorem, a `sorry` at the end indicates where models should fill in their proof. We also generate a Markdown description based on a rigorously aligned template as detailed in Appendix D. We provide a complete example of our BB Challenge with problem descriptions in both Lean-Markdown and model responses in Appendix C.

3.2. The Mixed Boolean-Arithmetic (MBA) Challenge

The Problem Module Mixed Boolean-Arithmetic (MBA) expressions combine arithmetic operators $(+, -, \times)$ with bitwise operations ($\wedge$ (AND), $\vee$ (OR), $\oplus$ (XOR), $\neg$ (NOT)) over integer variables, which form universal representations of polynomial functions over bitvectors (Reichenwallner & Meerwald-Stadler, 2022) used in cryptographic obfuscation (Liu et al., 2021).

MBA equations can be categorized into two types according to their construction:

Linear MBA equations. Equations of the form

$$\underbrace{\sum_k a_k \times e_k(x, y)}_{\text{Linear MBA expression A}} = \underbrace{\sum_k b_k \times e'_k(x, y)}_{\text{Linear MBA expression B}},$$

where each coefficient $a_k,\ b_k \in \{\pm 1, \pm 2, ..., \pm 11\}$. and each atomic boolean expression $e_k(x, y)$, $e'_k(x, y)$ have 16 options as below .

$$\left\{ \begin{array}{cccc} x & x \vee y & x \wedge y & x \oplus y \\ \neg x & x \vee \neg y & x \wedge \neg y & \neg(x \oplus y) \\ y & \neg(x \vee y) & x \wedge \neg x & \neg(x \wedge y) \\ \neg y & \neg(x \vee \neg y) & \neg(x \wedge \neg y) & \neg(x \wedge \neg x) \end{array} \right\}$$

Nonlinear MBA equations. Equations in which each side may include products of MBA sub-expressions $\big(e_{k_1}(x, y) \times e_{k_2}(x, y)\big)$ and arbitrarily nested bitwise operations, e.g. $10 \times \neg\Big(1 \times (x \wedge y)\ \vee\ \neg\big((-1) \times (x \vee \neg y)\big)\Big)$.

Dataset Generation In the first stage, we generate two-variable MBA equations following the MBA obfuscator protocol (Liu et al., 2021). Specifically, each equation is constructed by first generating a linear MBA expression that is identically zero, and then randomly moving a subset of terms to the right-hand side with their signs flipped. We then translate each equation into a Lean theorem paired with an informal Markdown description as illustrated in

Figure 1. Every variable and constant coefficient are declared as `BitVec 32` so that all arithmetic and bitwise operations are on 32-bit bitvectors modulo 2^{32}. We ensure infinite scalability by incrementing the number of variables in the MBA expression, which guarantees a sufficiently large problem space to mitigate contamination via combinatorial explosion.

Challenge definition. We ask models to prove the equivalence of two MBA expressions. Proofs must be constructed step by step using lemmas for arithmetic, bitwise, and distributive identities to ensure genuine symbolic reasoning. We explicitly discourage tactics such as bv_decide (which invokes a SAT solver) and other automatic approaches.

SAT solver. Lean provides a built-in tactic called bv_decide for automatically proving or disproving equalities over fixed-width bitvectors (BitVec n). Internally, bv_decide reduces the goal to a Boolean satisfiability problem (SAT) by applying bit-blasting, a process that translates bitvector operations into equivalent propositional logic formulas over individual bits. The resulting Boolean formula is then handed to an external SAT solver to check satisfiability. By default, bv_decide uses a 10-second timeout for the solver.

Generate easy-to-verify ground-truth. For any linear MBA equation

$$E_1(x, y) \;=\; E_2(x, y),$$

we normalize both sides into their *Weighted 2-DNF* (W2DNF):

$$\mathrm{W2DNF}(E) \;=\; \sum_{i,j\in\{0,1\}} c_{ij}(E)\,\bigl(\ell_i(x)\wedge \ell_j(y)\bigr),$$

where $\ell_0(z) = \neg z,\ \ell_1(z) = z$ and each $c_{ij}(E) \in \mathbb{Z}_{32}$ Thus the original equality $E_1 = E_2$ holds if and only if

$$c_{ij}(E1) = c_{ij}(E2) \quad \text{for all } i, j \in \{0, 1\}.$$

Normalization steps are directly translated into a sequence of Lean tactics that Lean can verify as the ground-truth solution for the MBA challenge.

Customized lemma library. We curated a custom library containing all necessary lemmas to reduce any 2-variable linear MBA expression into weighted 2-DNF form without relying on any external theorem or lemma, which enabled us to automatically generate and verify all ground-truth proofs for our MBA challenge. The complete library is provided in Appendix F.

Step-level task decomposition. In order to test if models perform differently in handling step-level task comparing to entire proof, we decompose each ground-truth proof into a sequence of step-level tasks. After each tactic is applied, Lean reports an unsolved goal. Whenever the left-hand side (LHS) or right-hand side (RHS) of the goal changes, we extract a step lemma according to the following rules:

- **For LHS changes:** The lemma's left-hand side matches the *previous* goal's LHS, and its right-hand side matches the *current* goal's LHS.
- **For RHS changes:** The lemma's left-hand side matches the *previous* goal's RHS, and its right-hand side matches the *current* goal's RHS.

For each problem, the final step (and *only* the final step) is always `simp`. All non-final tactics are exclusively one of three types: `simp only`, `rw`, or `nth_rewrite`, and crucially, each must use a lemma from our custom library. We provide a complete example of the step lemma challenge with model responses in Appendix C and results for this setting in 5.

4. Experimental

We report Pass@16 on various frontier models, which counts a case as successful if and only if at least one of the 16 attempts could pass Lean compiler verification, that is, no errors are reported and no 'sorry' is used in the proof.

While it's possible to set other n for Pass@n, we note that $n = 16$ is the common best practice for existing formal theorem proving benchmarks (Zheng et al., 2022; Tsoukalas et al., 2024; Liu et al., 2025), which strikes a balance of reasonable budget and non-deterministic inference. For unexpected failures such as Timeout, Out-Of-Memory and API error, we re-run each trial without counting the failed case into $n = 16$.

4.1. Evaluation Setup

Evaluated models. We evaluated both frontier general-purpose reasoning models and LLM-based theorem provers. Dedicated formal theorem provers are fine-tuned using FTP data on base models such as DeepSeekMath (Shao et al., 2024). For general-purpose LRMs, we chose a mixture of advanced open-source and proprietary models: DeepSeek-R1-671B (DeepSeek-AI et al., 2025), QwQ-32B (Qwen, 2025), OpenAI-o3 and o4-mini (OpenAI, 2025). Both OpenAI models carry default settings of 200K context window, 100K max output tokens, and a knowledge cut-off date on June 1, 2024. For LLM-based theorem provers, we evaluated various frontier models including Goedel-Prover-SFT (Lin et al., 2025), Leanabelle-Prover (Zhang et al., 2025), Kimina-Prover-Preview-7B (Wang et al., 2025), DeepSeek-Prover-v2-7B, and DeepSeek-Prover-v2-671B (Ren et al., 2025).

Model	BB(1)	BB(2)	BB(3)	BB(4)	Total
General LRMs					
DeepSeek-R1	21/50	12/50	10/50	12/50	55/200 (27.5%)
QwQ-32B-Preview	26/50	21/50	26/50	25/50	98/200 (49.0%)
OpenAI-o3	15/50	17/50	19/50	17/50	68/200 (34.0%)
OpenAI-o4-mini	19/50	10/50	12/50	13/50	54/200 (27.0%)
Theorem Provers					
Goedel-Prover-SFT	0/50	0/50	0/50	0/50	0/200 (0.0%)
Leanabell-Prover-GD-RL	0/50	0/50	0/50	0/50	0/200 (0.0%)
Kimina-Prover-Preview-7B	0/50	0/50	0/50	0/50	0/200 (0.0%)
DeepSeek-Prover-v2-7B	9/50	9/50	10/50	10/50	38/200 (19.0%)
DeepSeek-Prover-v2-671B	34/50	30/50	26/50	25/50	115/200 (57.5%)

Table 2. Pass@16 Results on BB Challenge, where each column represents an incremental setting for state count N as a complexity parameter.

Inference setting. For open-source models, we use greedy sampling to aid reproducibility, which is equivalent to fixing the sampling temperature to $T = 0.0$ as recommended by the model developers (DeepSeek-AI et al., 2025; Qwen, 2025) for math and coding tasks. Furthermore, We enable thinking mode for all models whenever possible and follow default settings on Reasoning Efforts, Context Length, Max Tokens, and System Prompts in model metadata.

We implemented our code under a Linux environment with `Python 3.10`. All open-source models are deployed using `transformer` library and `PyTorch` on a 4xGH200 server, with open-source models downloaded via `huggingface-cli`. The proprietary models are used directly via the OpenAI API. We prompted models to follow detailed instruction as in Appendix C.

After generation, model outputs are parsed to extract Lean proof, which is validated using a regular expression. The proof is then verified using `Kimina-Lean-Server` (Santos et al., 2025) built with `Lean and Mathlib v4.19.0` via a RESTful API.

4.2. Results

BB challenge. We report Pass@16 in Table 2 and analyze detailed failure modes in Section 4.4. Frontier LRMs OpenAI-o3 and o4-mini have struggled to score past 35% on our BB challenge, marking a critical gap in formal reasoning capability despite rapid saturation on static benchmarks from public sources.

Among LLM-based theorem provers, best-performing DeepSeek-Prover-v2-671B (Ren et al., 2025) scored 57.5%, but we notice rapid performance degradation as models scale down to 7B, where advanced theorem provers such as Goedel-Prover-SFT (Lin et al., 2025), Leanabelle-Prover-GD-RL (Zhang et al., 2025) and Kimina-Prover-Preview-7B (Wang et al., 2025) struggled to solve even 1 instance of our challenges.

Paradoxically, QwQ-32B (Qwen, 2025) attained best performance among general LRMs with only 32B parameters, which raises an intriguing question as to how scaling affects the performance of general reasoning models vs. dedicated theorem provers in a potentially different manner.

MBA challenge. We report Pass@16 in Table 5, revealing even more pronounced difficulties than BB challenge. Further, we decompose 10 randomly selected MBA challenge into 365 steps according to Section 3.2 and report step-level performance in Table 4.

Frontier LRMs including DeepSeek-R1 (DeepSeek-AI et al., 2025) and QwQ-32B (Qwen, 2025), struggled to solve any of our challenges, while OpenAI-o4-mini (OpenAI, 2025) solved only 6 problems.

Specialized theorem provers exhibited extreme stratification: DeepSeek-Prover-v2-671B (Ren et al., 2025) solved 12% of linear problems with lemmas, but failed all problems without lemmas and all non-linear problems. Smaller provers showed near-universal failure, with only Kimina-Prover-7B (Wang et al., 2025) solving a single instance. This demonstrates that even state-of-the-art formal systems struggle with fundamental algebraic reasoning when deprived of lemma support or faced with non-linear complexity.

4.3. Success Cases

We present a successful case of a proof of non-halting BB(4) machine and more cases in Appendix E. The model DeepSeekProver-v2-671B leveraged proof by induction and attempted to use various automated tactics, such as `aesop`

Model	Linear w/ Lemma	Linear w/o Lemma	Non-Linear
Reasoning LLMs			
DeepSeek-R1	0/100	0/100	0/40
OpenAI-o4-mini	2/100	4/100	0/40
QwQ-32B-Preview	0/100	0/100	0/40
Theorem Provers			
DeepSeek-Prover-v2-7B	0/100	0/100	0/40
Goedel-Prover	0/100	0/100	0/40
Kimina-Prover-7B	1/100	0/100	0/40
Leanabelle-Prover	0/100	0/100	0/40
DeepSeek-Prover-v2-671B	12/100	0/100	0/40

Table 3. Pass@16 Performance on MBA challenge. We evaluated all models using 2 parallel settings where models are tested with vs. without our customized lemma library as detailed in Section 3.2. This parallel setting aims to gauge their capability of in-context learning unseen lemma to tackle novel problems out of their training corpora.

Model	#Correct Steps
OpenAI-o4-mini	352/356
DeepSeek-Prover-v2-671B	258/356

Table 4. Step-level task performance on linear MBA where models are asked to choose an appropriate lemma from our customized lemma library for each step of the proof.

and `omega`, showing that a considerable manipulation of the automated tactic taxonomy.

```
DeepSeek-Prover-v2-671B:
theorem machine_never_halts : ∀ n, (nth_cfg n).isSome
    ↪:= by
  intro n
  have h : ∀ n, (nth_cfg n).isSome := by
    intro n
    induction n with
    | zero => simp [nth_cfg, init]
    | succ n ih =>
      cases h1 : nth_cfg n with
      | none => simp_all [nth_cfg, step, machine, Cfg,
          ↪Stmt, Γ, Fin.forall_fin_succ, Fin.
          ↪forall_fin_succ']
      | some cfg =>
        simp_all [nth_cfg, step, machine, Cfg, Stmt, Γ,
            ↪ Fin.forall_fin_succ, Fin.forall_fin_succ
            ↪']
        <;>
        (try cases cfg <;> simp_all [nth_cfg, step,
            ↪machine, Cfg, Stmt, Γ, Fin.
            ↪forall_fin_succ, Fin.forall_fin_succ'])
        <;>
        (try aesop)
        <;>
        (try omega)
        <;>
        (try aesop)
  exact h n
```

In this successful MBA case, the problem is simple enough that `bv_decide` is able to solve it within its default 10-second timeout by leveraging a SAT solver.

```
o4-mini:
theorem mba_challenge_88282d89 (x y : BitVec 32) : 1#32
    ↪ * ~~~(x &&& ~~~y) + 1#32 * ~~~(x ^^^ y) - 3#32 *
    ↪ (x ||| ~~~y) + 1#32 * ~~~(x ||| y) + 3#32 * (x
    ↪&&& ~~~y) + 1#32 * (x &&& y) = 1#32 * ~~~(x |||
    ↪~~~y) := by
  bv_decide
```

This case is more complex such that `bv_decide` alone cannot solve. However, the combination of `norm_num` and `bv_decide` is sufficient to complete the proof. Notably, the sequence of have statements introducing previously proven lemmas is entirely unused, and the tactics following the first `bv_decide` are never executed, as the goal is already solved at that point. This suggests that the model remains heavily reliant on automated tactics and lacks a clear understanding of what these tactics actually do.

```
DeepSeek-Prover-v2-671B:
theorem mba_challenge_3165c251 (x y : BitVec 32) :
  -3#32 * (x ||| ~~~y) - 1#32 * x - 2#32 * y + 1#32 *
      ↪~~~(x ||| y) + 2#32 * ~~~(x ||| ~~~y) + 2#32 *
      ↪(x &&& ~~~y) + 6#32 * (x &&& y) = -2#32 * ~~~y
      ↪:= by
  have h1 := bv32_not_xor_eq_or x y
  ...
  have h21 := bv32_sum_all y x
  norm_num [Nat.add_assoc, Nat.add_comm, Nat.
      ↪add_left_comm, Nat.mul_assoc, Nat.mul_comm, Nat
      ↪.mul_left_comm] at *
  <;> simp_all (config := {decide := true})
  <;> bv_decide
  <;> simp_all (config := {decide := true})
  <;> bv_decide
```

4.4. Failure Cases

Our experiments reveal several key failure modes of frontier models. These failure can be categorized into 4 major types with signature patterns as shown in Table 5. More cases in

Appendix E.

Error Message Type	%(#Errors)
Irrelevant Hallucination	67.27% (122404)
Tactic Misuse	23.22% (42247)
Voluntary Give Up	4.88% (8871)
Type Mismatch	4.47% (8136)
Others	0.16% (300)

Table 5. Summary of error type distribution for both BB and MBA Challenge.

Hallucination. Hallucination consistently appears within 7B theorem provers, where they often tried to use non-existent theorems or tactics. DeepSeek-Prover-v2-671B has exhibited much stronger anti-hallucination capability thanks to its large base model.

Here we show 2 cases where Goedel-Prover-SFT attempted to prove by induction on a completely unrelated problem. Similarly, Kimina-Prover-Preview tried to rename the given theorem head `nth_cfg` to `nth_configuration`, which led to an unsolved goal at the end of the proof.

```
Goedel-Prover-SFT:
theorem sequence_formula {a : ℕ → ℕ} (h1 : a 1 = 1)
    (h_rec : ∀ n : ℕ, n ≥ 1 → a (n + 1) = 2 * a n + 1)
        ↪:
    ∀ n : ℕ, n ≥ 1 → a n = 2^n - 1 := by
```

```
Kimina-Prover-Preview:
theorem machine_never_halts : ∀ n, (nth cfigurations n)
    ↪.isSome := by
  intro n
  induction n with
  | zero =>
    simp [nth configurations]
  | succ n ih =>
    simp [nth configurations, ih]
    <;> aesop
```

For the MBA challenge, we show another failure case of OpenAI-o4-mini where model hallucinates a non-existent tactic `bv_simp`.

```
o4-mini:
theorem mba_challenge_14b066fa (x y : BitVec 32) :
    ↪-1#32 * ~~~(x ^^^ y) + 2#32 * ~~~x + 2#32 * ~~~(x
    ↪ &&& y) + 1#32 * (x ||| y) - 3#32 * x + 3#32 * (x
    ↪ ||| ~~~y) - 2#32 * (x ^^^ y) + 1#32 * ~~~(x &&&
    ↪~~~x) - 5#32 * y - 8#32 * ~~~(x ||| y) - 2#32 * (
    ↪x &&& ~~~y) + 3#32 * (x &&& y) = -1#32 * ~~~(x
    ↪&&& ~~~y) := by
  bv_simp
  ring
```

Tactic misuse. Models showed a dangerous tendency to blindly apply automated proof tactics such as aesop (which automatically looks for search tactics recursively in a proof tree) without understanding the necessary conditions for aesop to be applicable, resulting in the failure of exhaustive search.

```
DeepSeek-Prover-v2-671B:
theorem machine_never_halts : ∀ n, (nth configurations
    ↪n).isSome := by
  intro n
  ...
    <;> aesop
```

In this MBA challenge example, this tendency is even more pronounced. The model gathers many lemmas into one `simp_only` call without checking their relevance and then blindly invokes ring. The proof is bloated with useless rewrites and heavy automation, yet still failsdemonstrating a failure to reason about lemma and tactic applicability.

```
o4-mini with lemmas:
theorem mba_challenge_080db093 (x y : BitVec 32) :
  -5#32 * ~~~y + 1#32 * ~~~(x &&& ~~~x) + 4#32 * (x |||
      ↪ y) - 6#32 * (x ^^^ y) - 6#32 * ~~~(x &&& ~~~y)
      ↪ - 1#32 * ~~~(x &&& y) - 3#32 * y - 11#32 * ~~~
      ↪x + 24#32 * ~~~(x ||| y) + 22#32 * ~~~(x |||
      ↪~~~y) + 13#32 * (x &&& ~~~y) +  6#32 * (x &&& y
      ↪) = -4#32 * ~~~(x ^^^ y) + 6#32 * (x ||| ~~~y)
      ↪:= by
  simp only [bv32_and_not_self, bv32_not_and_self,
      ↪bv32_or_not_self, bv32_not_or_self] at *
  simp only [bv32_x_distr, bv32_y_distr, ...] at *
  simp only [bv32_sum_all] at *
  ring
```

This sample illustrates a tactic misuse where`bv_decide` fails to solve the goal within the 10-second timeout. The failure highlights the limitations of relying solely on automated tactics for large or intricate goals.

```
o4-mini:
import Std.Tactic.BVDecide
/--
Let x,y be 32-bit bit-vectors. Prove the equivalence of
    ↪ the following two expressions: 7·x - 5·y - 2·(x⊕
    ↪y) - 6·¬(x ∧ ¬x) - 5·(x∨y) - 2·¬(x∧y) - (x∨¬y) +
    ↪4·¬y - 7·¬(x⊕y) + 13·¬(x∨y) + 21·¬(x∨¬y) + 17·(x∧
    ↪y) and -5·(x∧¬y) + 1·¬x
-/
theorem mba_challenge_6f99807f (x y : BitVec 32) :
    ↪((7#32 * x) - (5#32 * y) - (2#32 * (x ^^^ y)) -
    ↪(6#32 * (~~~(x &&& (~~~x)))) - (5#32 * (x ||| y))
    ↪ - (2#32 * (~~~(x &&& y))) - (1#32 * (x ||| (~~~y
    ↪))) + (4#32 * (~~~y)) - (7#32 * (~~~(x ^^^ y))) +
    ↪ (13#32 * (~~~(x ||| y))) + (21#32 * (~~~(x |||
    ↪(~~~y)))) + (17#32 * (x &&& y))) = ((-5#32) * (x
    ↪&&& (~~~y))) + (1#32 * (~~~x)) := by
  bv_decide
```

Type mismatch. Type mismatch is a fundamental syntactic error arising from the type-dependent nature of Lean system, which directly reflects models' failure to understand and correctly apply respective Lean tactics in question. In this case, the 7B theorem-proving models failed to interpret and adapt to Leans inherent type rules, producing expressions whose inferred types did not match the expected ones.

Fortunately, this type of error occurs rarely at 4.47% in the big picture, which speaks to the promising progress that frontier models have a considerably correct understanding of the Lean tactic taxonomy.

```
Kimina-Prover-Preview:
theorem machine_never_halts : ∀ n, (nth_cfg n).isSome
    ↪:= by
  intro n
  induction n with
    ...
  | succ n ih =>
    simp [nth_cfg, step, machine]
    exact ih
```

Voluntary give-up. In this case, the model voluntarily chose to give up by leaving `sorry` under the proof goal without even attempting to solve the problem with any tactics at all. This type of error makes up 4.88% of total errors, which could be seen as a failure of instruction-following often depicted by reasoning models comparing to general-purpose LLMs.

```
Kimina-Prover-Preview:
theorem machine_never_halts : ∀ n, (nth_cfg n).isSome
    ↪:= by
  sorry
```

5. Discussion

The Automation Trap in Theorem Proving Our empirical analysis identifies a systematic overreliance on automated tactics as primary proof mechanisms across frontier models. Models predominantly deploy tactics such as `aesop` for recursive proof search and `bv_decide` for SAT-based solving as black-box oracles, demonstrating minimal comprehension of their formal operational boundaries. This dependency manifests in two critical failure modalities: (1) proofs that circumvent genuine deductive understanding when tactics succeed coincidentally, and (2) complete proof collapse when tactics fail without fallback mechanisms. Notably, models frequently generate elaborate lemma declarations while defaulting to brute-force tactic sequencinga pattern indicating a fundamental disconnect between syntactic manipulation and strategic reasoning that fundamentally undermines verification integrity.

The Reasoning Gap between Step-Level vs. Holistic Proof Task Our step-level task decomposition setting reveals a significant discrepancy between step-level performance and holistic proof synthesis with a simple overhead proof goal. Models demonstrate near-perfect capability in choosing unseen out-of-distribution lemma for atomic inference tasks yet exhibit catastrophic failure rates in composing these operations into complete proofs. This divergence indicates that the primary bottleneck resides not in local operations but in global proof planning and strategic integration. The persistent inability to reconcile stepwise correctness with end-to-end whole proof generation suggests reasoning capabilities of frontier models remain constrained in long contexts by insufficient strategic planning for orchestrating atomic transformations into coherent proof strategies when challenged with lengthy proofs containing multiple steps.

Contamination-Resistant Evaluation Framework Our methodology establishes a novel evaluation paradigm through the integration of Lean with Theoretical Computer Science. Our framework achieves infinite scalability via algorithmic problem generation from parameterized TCS modules, with granular difficulty modulation through computational parameters (e.g., Turing Machine state complexity, MBA expression depth). Automated verifiability provides ground-truth validation without human intervention, while dynamically generated problem spaces ensure intrinsic resistance to dataset contamination. By enabling rigorous formal-informal alignment absent expert curation, this TCS-inspired synthesis creates a sustainable evaluation ecosystem where benchmark freshness and complexity scales along with the progress of frontier theoretical computer science research.

6. Conclusion

We propose Theoretical Computer Science as a promising testbed for formal theorem proving, which by design is capable of utilizing modularized, fully automatic problem generation to systematically overcome contamination and scalability limitations in legacy benchmarks. Our TCS-inspired modular synthesis paradigm enables the creation of infinitely scalable, formally rigorous problem spaces with adjustable complexity through computational parameters such as Turing Machine state configurations and MBA expression depth. Our experimental results demonstrate a profound reasoning gap: while models achieve near-perfect scores of 98.88% on atomic step-level tasks involving out-of-distribution lemma selection, they collapse to a mere 12% success rate when synthesizing complete lengthy proofs. This drastic performance degradation highlights the devastating role of hallucination during long-context reasoning sequences, which may act as a roadblock for models to form any systematic strategies for tasks requiring lengthy whole proofs. We also reveal the dangerous tendency of frontier models' over-reliance on automated tactics like `aesop` and `bv_decide` without understanding their applicability with limitations. To tackle these challenges, we call on the Lean community to further explore the massive potential of theoretical computer science and its interplay with the realm of formal theorem proving, thereby leveraging the joint effort to advance both domains synergistically.

References

Aaronson, S. The busy beaver frontier. *SIGACT News*, 51(3):32–54, 2020. doi: 10.1145/3427361.3427369. URL https://doi.org/10.1145/3427361.3427369.

Anthropic. Claude 3.7 sonnet system card. https://assets.anthropic.com/m/785e231869ea8b3b/original/claude-3-7-sonnet-system-card.pdf, 2025. Accessed: 2025-05-10.

Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., and Avigad, J. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *CoRR*, abs/2302.12433, 2023. doi: 10.48550/ARXIV.2302.12433. URL https://doi.org/10.48550/arXiv.2302.12433.

Besta, M., Barth, J., Schreiber, E., Kubicek, A., Catarino, A., Gerstenberger, R., Nyczyk, P., Iff, P., Li, Y., Houliston, S., et al. Reasoning language models: A blueprint. *arXiv preprint arXiv:2501.11223*, 2025.

Chen, S., Yu, W., Dou, G., and Zhang, Q. A review on mechanical proving and formalization of mathematical theorems. *IEEE Access*, 13:50672–50686, 2025. doi: 10.1109/ACCESS.2025.3552634. URL https://doi.org/10.1109/ACCESS.2025.3552634.

de Moura, L. and Ullrich, S. The lean 4 theorem prover and programming language. In Platzer, A. and Sutcliffe, G. (eds.), *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37. URL https://doi.org/10.1007/978-3-030-79876-5_37.

DeepSeek-AI, Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., Zhang, X., Yu, X., Wu, Y., Wu, Z. F., Gou, Z., Shao, Z., Li, Z., Gao, Z., Liu, A., Xue, B., Wang, B., Wu, B., Feng, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Bao, H., Xu, H., Wang, H., Ding, H., Xin, H., Gao, H., Qu, H., Li, H., Guo, J., Li, J., Wang, J., Chen, J., Yuan, J., Qiu, J., Li, J., Cai, J. L., Ni, J., Liang, J., Chen, J., Dong, K., Hu, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Zhao, L., Wang, L., Zhang, L., Xu, L., Xia, L., Zhang, M., Zhang, M., Tang, M., Li, M., Wang, M., Li, M., Tian, N., Huang, P., Zhang, P., Wang, Q., Chen, Q., Du, Q., Ge, R., Zhang, R., Pan, R., Wang, R., Chen, R. J., Jin, R. L., Chen, R., Lu, S., Zhou, S., Chen, S., Ye, S., Wang, S., Yu, S., Zhou, S., Pan, S., and Li, S. S. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/2501.12948, 2025. doi: 10.48550/ARXIV.2501.12948. URL https://doi.org/10.48550/arXiv.2501.12948.

Dekoninck, J., Müller, M. N., and Vechev, M. T. Constat: Performance-based contamination detection in large language models. In Globersons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J. M., and Zhang, C. (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/a7f89793b9e6f8c6568dbbb6ff727b9b-Abstract-Conference.html.

Dong, Y., Jiang, X., Liu, H., Jin, Z., Gu, B., Yang, M., and Li, G. Generalization or memorization: Data contamination and trustworthy evaluation for large language models. In Ku, L., Martins, A., and Srikumar, V. (eds.), *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pp. 12039–12050. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.FINDINGS-ACL.716. URL https://doi.org/10.18653/v1/2024.findings-acl.716.

Google DeepMind. Gemini 2.5: Our most intelligent ai model. Google AI Blog, Mar 2025; https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/, 2025.

Harland, J. Generating candidate busy beaver machines (or how to build the zany zoo). *Theor. Comput. Sci.*, 922:368–394, 2022. doi: 10.1016/J.TCS.2022.04.040. URL https://doi.org/10.1016/j.tcs.2022.04.040.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the MATH dataset. In Vanschoren, J. and Yeung, S. (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/be83ab3ecd0db773eb2dc1b0a17836a1-Abstract-round2.html.

Huang, Y., Lin, X., Liu, Z., Cao, Q., Xin, H., Wang, H., Li, Z., Song, L., and Liang, X. MUSTARD: mastering uniform synthesis of theorem and proof data. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=8xliOUg9EW.

Li, Z., Hua, W., Wang, H., Zhu, H., and Zhang, Y. Formal-llm: Integrating formal language and natural language for controllable llm-based agents, 2024a. URL https://arxiv.org/abs/2402.00798.

Li, Z., Wu, Y., Li, Z., Wei, X., Zhang, X., Yang, F., and Ma, X. Autoformalize mathematical statements by symbolic equivalence and semantic consistency. In Globersons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J. M., and Zhang, C. (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024b. URL http://papers.nips.cc/paper_files/paper/2024/hash/6034a661584af6c28fd97a6f23e56c0a-Abstract-Conference.html.

Lin, Y., Tang, S., Lyu, B., Wu, J., Lin, H., Yang, K., Li, J., Xia, M., Chen, D., Arora, S., and Jin, C. Goedel-prover: A frontier model for open-source automated theorem proving, 2025. URL https://arxiv.org/abs/2502.07640.

Liu, B., Feng, W., Zheng, Q., Li, J., and Xu, D. Software obfuscation with non-linear mixed boolean-arithmetic expressions. In Gao, D., Li, Q., Guan, X., and Liao, X. (eds.), *Information and Communications Security - 23rd International Conference, ICICS 2021, Chongqing, China, November 19-21, 2021, Proceedings, Part I*, volume 12918 of *Lecture Notes in Computer Science*, pp. 276–292. Springer, 2021. doi: 10.1007/978-3-030-86890-1_16. URL https://doi.org/10.1007/978-3-030-86890-1_16.

Liu, J., Lin, X., Bayer, J., Dillies, Y., Jiang, W., Liang, X., Soletskyi, R., Wang, H., Xie, Y., Xiong, B., Yang, Z., Zhang, J., Zhi, L., Li, J., and Liu, Z. Combibench: Benchmarking llm capability for combinatorial mathematics, 2025. URL https://arxiv.org/abs/2505.03171.

Lu, J., Liu, Z., Wan, Y., Huang, Y., Wang, H., Yang, Z., Tang, J., and Guo, Z. Process-driven autoformalization in lean 4. *CoRR*, abs/2406.01940, 2024. doi: 10.48550/ARXIV.2406.01940. URL https://doi.org/10.48550/arXiv.2406.01940.

Magar, I. and Schwartz, R. Data contamination: From memorization to exploitation. In Muresan, S., Nakov, P., and Villavicencio, A. (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pp. 157–165. Association for Computational Linguistics, 2022. doi: 10.18653/V1/2022.ACL-SHORT.18. URL https://doi.org/10.18653/v1/2022.acl-short.18.

Michel, P. Busy beaver competition and collatz-like problems. *Arch. Math. Log.*, 32(5):351–367, 1993. doi: 10.1007/BF01409968. URL https://doi.org/10.1007/BF01409968.

Murphy, W., Holzer, N., Qiao, F., Cui, L., Rothkopf, R., Koenig, N., and Santolucito, M. Combining LLM code generation with formal specifications and reactive program synthesis. *CoRR*, abs/2410.19736, 2024. doi: 10.48550/ARXIV.2410.19736. URL https://doi.org/10.48550/arXiv.2410.19736.

Ni, J., Xue, F., Yue, X., Deng, Y., Shah, M., Jain, K., Neubig, G., and You, Y. Mixeval: Deriving wisdom of the crowd from LLM benchmark mixtures. In Globersons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J. M., and Zhang, C. (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/b1f34d7b4a03a3d80be8e72eb430dd81-Abstract-Conference.html.

Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 3-540-43376-7. doi: 10.1007/3-540-45949-9. URL https://doi.org/10.1007/3-540-45949-9.

OpenAI. Openai o3 and o4-mini system card. https://cdn.openai.com/pdf/2221c875-02dc-4789-800b-e7758f3722c1/o3-and-o4-mini-system-card.pdf, 2025.

Polu, S. and Sutskever, I. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020. URL https://arxiv.org/abs/2009.03393.

Qwen. Qwq - 32b: The power of scaling rl, March 2025. URL https://qwenlm.github.io/blog/qwq-32b/. Accessed: 2025 - 05 - 14.

Reichenwallner, B. and Meerwald-Stadler, P. Efficient deobfuscation of linear mixed boolean-arithmetic expressions. In III, G. G. R. and Blazytko, T. (eds.), *Proceedings of the 2022 ACM Workshop on Research on* *offensive and defensive techniques in the context of Man At The End (MATE) attacks, Los Angeles, CA, USA, 11 November 2022*, pp. 19–28. ACM, 2022. doi: 10.1145/3560831.3564256. URL https://doi.org/10.1145/3560831.3564256.

Rein, D., Hou, B. L., Stickland, A. C., Petty, J., Pang, R. Y., Dirani, J., Michael, J., and Bowman, S. R. GPQA: A graduate-level google-proof q&a benchmark. *CoRR*, abs/2311.12022, 2023. doi: 10.48550/ARXIV.2311.12022. URL https://doi.org/10.48550/arXiv.2311.12022.

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., and Ruan, C. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition, 2025. URL https://arxiv.org/abs/2504.21801.

Santos, M. D., Wang, H., de Saxcé, H., Wang, R., Baksys, M., Unsal, M., Liu, J., Liu, Z., and Li, J. Kimina lean server: Technical report. *arXiv preprint arXiv:2504.21230*, 2025.

Scholze, P. Liquid tensor experiment. *Exp. Math.*, 31(2):349–354, 2022. doi: 10.1080/10586458.2021.1926016. URL https://doi.org/10.1080/10586458.2021.1926016.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Tao, T. ams.org. https://www.ams.org/notices/202501/rnoti-p6.pdf, 2025. [Accessed 10-05-2025].

Tsoukalas, G., Lee, J., Jennings, J., Xin, J., Ding, M., Jennings, M., Thakur, A., and Chaudhuri, S. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. In Globersons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J. M., and Zhang, C. (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/1582eaf9e0cf349e1e5a6ee453100aa1-Abstract-Datasets_and_Benchmarks_Track.html.

Wang, H., Unsal, M., Lin, X., Baksys, M., Liu, J., Santos, M. D., Sung, F., Vinyes, M., Ying, Z., Zhu, Z., Lu, J., de Saxc, H., 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., and Li, J. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning, 2025. URL https://arxiv.org/abs/2504.11354.

White, C., Dooley, S., Roberts, M., Pal, A., Feuer, B., Jain, S., Shwartz-Ziv, R., Jain, N., Saifullah, K., Naidu, S., Hegde, C., LeCun, Y., Goldstein, T., Neiswanger, W., and Goldblum, M. Livebench: A challenging, contamination-free LLM benchmark. *CoRR*, abs/2406.19314, 2024. doi: 10.48550/ARXIV.2406.19314. URL https://doi.org/10.48550/arXiv.2406.19314.

Wu, Y., Jiang, A., Ba, J., and Grosse, R. B. {INT}: An inequality benchmark for evaluating generalization in theorem proving. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=O6LPudowNQm.

Wu, Y., Jiang, A. Q., Li, W., Rabe, M. N., Staats, C., Jamnik, M., and Szegedy, C. Autoformalization with large language models. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/d0c6bc641a56bebee9d985b937307367-Abstract-Conference.html.

Xu, C. Skelet 17 and the fifth busy beaver number, 2024. URL https://arxiv.org/abs/2407.02426.

Yang, Z., Wang, Y., Huang, Y., Guo, Z., Shi, W., Han, X., Feng, L., Song, L., Liang, X., and Tang, J. Optibench meets resocratic: Measure and improve llms for optimization modeling, 2024. URL https://arxiv.org/abs/2407.09887.

Ying, H., Wu, Z., Geng, Y., Wang, J., Lin, D., and Chen, K. Lean workbook: A large-scale lean problem set formalized from natural language math problems. In Globersons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J. M., and Zhang, C. (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.

nips.cc/paper_files/paper/2024/hash/bf236666a2cc5f3ae05d2e08485efc4c-Abstract-Datasets_and_Benchmarks_Track.html.

Zayyad, M. and Adi, Y. Formal language knowledge corpus for retrieval augmented generation, 2024. URL https://arxiv.org/abs/2412.16689.

Zhang, J., Wang, Q., Ji, X., Liu, Y., Yue, Y., Zhang, F., Zhang, D., Zhou, G., and Gai, K. Leanabell-prover: Posttraining scaling in formal reasoning, 2025. URL https://arxiv.org/abs/2504.06122.

Zheng, K., Han, J. M., and Polu, S. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

A. Limitations

While this work has been conducted according to best practices of previous work on formal theorem proving, there are several potential limitations:

Firstly, our evaluation for proprietary OpenAI models are conducted with their default configuration, combined with our zero-temperature setting for open-sourced models to enhance reproducibility, all evaluated models may not have shared completely identical settings due to the undisclosed parameter settings of proprietary models.

Meanwhile, in order to strike a balance between budgetary restraints and performance representativeness, we only selected best-performing models on mainstream formal theorem proving benchmarks such as PutnamBench (Tsoukalas et al., 2024) and miniF2F (Zheng et al., 2022) for evaluation. There may be merit in exploring how other frontier proprietary models such as Gemini-2.5 (Google DeepMind, 2025) and Claude-3.7-Sonnet (Anthropic, 2025) would perform on our dataset. Therefore, we plan to open-source our codebase and dataset to the community for researchers with ample resources to explore more models.

Lastly, while we leverage Lean compiler to automate proof validation, the proof is graded on a pass/fail basis as a whole. However, we note it's possible to implement a stepwise grading metric by dynamically masking each step in a whole proof and ask the model to fill in each step respectively, thereby gauging models' capability to implement key intermediate steps.

B. Automated Proof Validation

In our validation pipeline, we adopt the open-source Lean 4 server implementation from Kimina Lean Server (Santos et al., 2025). The Kimina Lean Server provides a Python interface allowing real-time feedback for generated proofs. With a Python function call `verify`, the server receives a list of Lean proofs and returns validation results for each proof. The server itself handles multiple requests efficiently by spreading verification across multiple Lean REPL processes.

C. Complete Example with Our Prompting Strategy

````
```lean4
{lean_code}
```
You can make your own auxiliary corollaries and theorems to support the proof, instead of only completing the part
    ↪with the sorry. Please output the entire program and not just the last part. Please output only the program
    ↪and add no other comment, such that your answer is a compilable lean code. Make sure to reason enough to make
    ↪ your code correct.
"""
````

System Prompt We used default system prompts as in the evaluated models' metadata.

A complete BB example. Below we demonstrate a complete record of evaluation, including the problem statement in Lean 4 and Markdown, our evaluation prompt and the model response:

```
import Mathlib.Computability.TuringMachine

inductive Γ
  | zero
  | one
   deriving DecidableEq

instance : Inhabited Γ := ⟨ Γ.zero ⟩

structure Stmt where
  move : Turing.Dir
  write : Γ

def Machine := Fin 2 → Γ → Option (Fin 2    Stmt)

structure Cfg where
  q : Fin 2
  Tape : Turing.Tape Γ
```

```
def init (l : List Γ) : Cfg := ⟨⟨0, by omega⟩, Turing.Tape.mk₁ l⟩

def step (M : Machine) : Cfg → Option Cfg :=
  fun ⟨q, T⟩     (M q T.head).map fun ⟨q', a⟩     ⟨q', (T.write a.write).move a.move⟩

def machine : Machine
| ⟨000, _⟩, Γ.zero  => some ⟨⟨000, by omega⟩, ⟨Turing.Dir.left, Γ.zero⟩⟩
| ⟨000, _⟩, Γ.one  => none
| ⟨001, _⟩, Γ.zero  => none
| ⟨001, _⟩, Γ.one  => some ⟨⟨000, by omega⟩, ⟨Turing.Dir.left, Γ.one⟩⟩
| ⟨_+2, _⟩, _ => by omega

def nth_cfg : (n : Nat) -> Option Cfg
| 0 => init []
| Nat.succ n => match (nth_cfg n) with
                | none => none
                | some cfg =>  step machine cfg

/--
Prove the following Turing Machine with the transition table never halts.
| State | Symbol | Next State | Move | Write |
|------|------|------|------|------|
| 000 | zero | 000 | left | zero |
| 000 | one | HALT | - | - |
| 001 | zero | HALT | - | - |
| 001 | one | 000 | left | one |
-/
theorem machine_never_halts : ∀ n, (nth_cfg n).isSome := by
  sorry
```

You can make your own auxiliary corollaries and theorems to support the proof, instead of only completing the part ↪with the sorry. Please output the entire program and not just the last part. Please output only the program ↪and add no other comment, such that your answer is a compilable lean code. Make sure to reason enough to make ↪ your code correct.
Response:

```lean4
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat

/-- Given a sequence { a   } with a₁ = 1 and  a    Ł ₁ = 2 a    + 1,
    prove that  a    = 2^n - 1 for all n ≥ 1 -/
theorem sequence_formula {a : ℕ → ℕ} (h1 : a 1 = 1)
    (h_rec : ∀ n : ℕ, n ≥ 1 → a (n + 1) = 2 * a n + 1) :
    ∀ n : ℕ, n ≥ 1 → a n = 2^n - 1 := by
  /-
  We need to prove that for a sequence \( \{a_n\} \) defined by \( a_1 = 1 \) and \( a_{n+1} = 2a_n + 1 \), the
      ↪general term \( a_n \) is given by \( a_n = 2^n - 1 \) for all \( n \geq 1 \).
  1. **Base Case**: For \( n = 1 \), we have \( a_1 = 1 \). This matches \( 2^1 - 1 = 1 \), so the base case holds.
  2. **Inductive Step**: Assume \( a_n = 2^n - 1 \) holds for some \( n \geq 1 \). We need to show it holds for \( n
      ↪ + 1 \).
     - By the recurrence relation, \( a_{n+1} = 2a_n + 1 \).
     - Using the inductive hypothesis \( a_n = 2^n - 1 \), we get:
       \[
       a_{n+1} = 2(2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1
       \]
     - Thus, \( a_{n+1} = 2^{n+1} - 1 \), completing the inductive step.
  By induction, \( a_n = 2^n - 1 \) holds for all \( n \geq 1 \).
  -/
  intro n hn
  induction' hn with n hn
  -- Base case: n = 1
  -- We know a 1 = 1, which matches 2^1 - 1 = 1
  simp_all
  -- Inductive step: Assume a n = 2^n - 1, prove a (n + 1) = 2^(n + 1) - 1
  cases n with
  | zero =>
```

```
    -- When n = 0, the induction hypothesis does not apply, so we directly use the initial condition
    simp_all
  | succ n =>
    -- For n = succ n, use the recurrence relation and the inductive hypothesis
    simp_all [Nat.succ_eq_add_one, Nat.pow_succ, Nat.mul_succ]
    -- Simplify the expression to match the form 2^(n + 1) - 1
    ring
    <;> omega
```

A complete step lemma challenge example. Below we demonstrate a complete record of evaluation, including the problem statement in Lean 4 and Markdown, our evaluation prompt and the model response:

````
Theorem library:

```lean4
import Mathlib.Tactic.Lemma
import Mathlib.Tactic.NthRewrite

@[simp]
theorem bv32_and_not_self(x : BitVec 32) :
  x &&& ~~~x = 0 := by
  simp

@[simp]
theorem bv32_not_not(x : BitVec 32) :
  ~~~~~~x = x := by
  simp

@[simp]
theorem bv32_or_not_self(x : BitVec 32) :
  x ||| ~~~x = BitVec.allOnes 32 := by
  simp

@[simp]
theorem bv32_not_or_self(x : BitVec 32) :
  ~~~x ||| x = BitVec.allOnes 32 := by
  simp

@[simp]
theorem bv32_neg_mul (x y : BitVec 32) :
  -x * y = -(x * y) := by
  simp

theorem bv32_not_and (x y : BitVec 32) :
  ~~~(x &&& y) = ~~~x ||| ~~~y := by
  rw [BitVec.not_and]

theorem bv32_not_or (x y : BitVec 32) :
  ~~~(x ||| y) = ~~~x &&& ~~~y := by
  rw [BitVec.not_or]

theorem bv32_not_xor_eq_or (x y : (BitVec 32)) :
  ~~~(x ^^^ y) = (~~~x &&& ~~~y) ||| (x &&& y) := by
  ext i
  simp
  cases h₁ : x[i] <;> cases h₂ : y[i]
  simp
  simp
  simp
  simp

theorem bv32_xor_eq_or (x y : (BitVec 32)) :
  (x ^^^ y) = (~~~x &&& y) ||| (x &&& ~~~y) := by
  ext i
  simp
  cases h₁ : x[i] <;> cases h₂ : y[i]
  simp
  simp
  simp
  simp
````
  ~~~

```
theorem bv32_x_distr (x y: BitVec 32) :
  x = (x &&& y) ||| (x &&& ~~~y) := by
  ext i
  simp
  simp [    Bool.and_or_distrib_left]

theorem bv32_y_distr (x y: BitVec 32) :
  y = (x &&& y ||| ~~~x &&& y) := by
  ext i
  simp
  simp [    Bool.and_or_distrib_right]


theorem bv32_add_assoc (x y z : BitVec 32) :
  x + y + z = x + (y + z) := by
  rw [BitVec.add_assoc]

theorem bv32_add_comm(x y : BitVec 32) :
  x + y = y + x := by
  rw [BitVec.add_comm]

theorem bv32_add_neg_eq_sub {x y : BitVec 32} :
  x + -y = x - y := by
  rw [BitVec.add_neg_eq_sub]

theorem bv32_mul_comm (x y : BitVec 32) :
  x * y = y * x := by
  rw [BitVec.mul_comm]

theorem bv32_var_mul_comm (x y z: BitVec 32) :
  (x &&& y) * z = z * (x &&& y) := by
  rw [BitVec.mul_comm]

theorem bv32_mul_add (x y z : BitVec 32) :
  x * (y + z) = x * y + x * z := by
  rw [BitVec.mul_add]

theorem bv32_neg_eq_mul (x : BitVec 32) :
  -x = x *  (BitVec.allOnes 32) := by
  rw [    BitVec.neg_one_eq_allOnes]
  rw [BitVec.mul_neg]
  rw [BitVec.mul_one]

theorem bv32_add_mul_one (x y : BitVec 32) :
  x + x * y = x * (1#32 + y) := by
  rw [BitVec.mul_add]
  rw [BitVec.mul_one]

/--
1: x &&& y
2: ~~~x &&& y
3: x &&& ~~~y
4: ~~~x &&& ~~~y
-/
theorem bv32_or_eq_add12 (x y : BitVec 32) :
  (x &&& y) ||| (~~~x &&& y) = (x &&& y) + (~~~x &&& y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ (~~~x)]
  simp [    BitVec.and_assoc]

theorem bv32_or_eq_add13 (x y : BitVec 32) :
  (x &&& y) ||| (x &&& ~~~y) = (x &&& y) + (x &&& ~~~y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ x]
  simp [BitVec.and_assoc]

theorem bv32_or_eq_add14 (x y : BitVec 32) :
```

```
  (x &&& y) ||| (~~~x &&& ~~~y) = (x &&& y) + (~~~x &&& ~~~y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ (~~~x)]
  simp [    BitVec.and_assoc]

theorem bv32_or_eq_add21 (x y : BitVec 32) :
  (~~~x &&& y) ||| (x &&& y) = (~~~x &&& y) + (x &&& y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ x]
  simp [    BitVec.and_assoc]

theorem bv32_or_eq_add23 (x y : BitVec 32) :
  (~~~x &&& y) ||| (x &&& ~~~y) = (~~~x &&& y) + (x &&& ~~~y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ x]
  simp [    BitVec.and_assoc]

theorem bv32_or_eq_add31 (x y : BitVec 32) :
  (x &&& ~~~y) ||| (x &&& y) = (x &&& ~~~y) + (x &&& y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ x]
  simp [    BitVec.and_assoc]
  simp [BitVec.and_assoc]

theorem bv32_or_eq_add32 (x y : BitVec 32) :
  (x &&& ~~~y) ||| (~~~x &&& y) = (x &&& ~~~y) + (~~~x &&& y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ (~~~x)]
  simp [    BitVec.and_assoc]

theorem bv32_or_eq_add41 (x y : BitVec 32) :
  (~~~x &&& ~~~y) ||| (x &&& y) = (~~~x &&& ~~~y) + (x &&& y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ x]
  simp [    BitVec.and_assoc]

theorem bv32_or_eq_add_three (x y : BitVec 32) :
  (x ||| y) = (x &&& ~~~y) + (x &&& y) + (~~~x &&& y) := by
  nth_rw 1 [bv32_y_distr x y]
  nth_rw 1 [bv32_x_distr x y]
  simp [    BitVec.or_assoc]
  simp [BitVec.or_comm _ (x &&& y)]
  simp [    BitVec.or_assoc]
  rw [BitVec.or_comm (x &&& y)]
  apply Eq.symm
  rw [BitVec.add_eq_or_of_and_eq_zero]
  rw [BitVec.add_eq_or_of_and_eq_zero]
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ x]
  simp [BitVec.and_assoc]
  rw [BitVec.add_comm]
  rw [    bv32_or_eq_add13]
  rw [    bv32_x_distr x y]
  simp [    BitVec.and_assoc]

theorem bv32_sum_all (x y : BitVec 32) :
  (~~~x &&& ~~~y) + (~~~x &&& y) + (x &&& y) + (x &&& ~~~y) = BitVec.allOnes 32 := by
  simp [BitVec.add_comm _ (~~~x &&& y)]
  simp [BitVec.add_comm _ (x &&& _)]
  simp [    BitVec.add_assoc]
```

```
  rw [BitVec.add_eq_or_of_and_eq_zero]
  rw [    bv32_or_eq_add_three x y]
  nth_rw 1 [bv32_x_distr x y]
  simp [BitVec.or_comm _ y]
  nth_rw 1 [bv32_y_distr x y]
  simp [    BitVec.or_assoc]
  simp [BitVec.or_comm _ (x &&& y)]
  simp [    BitVec.or_assoc]
  simp [BitVec.or_comm _ (~~~x &&& _)]
  simp [    BitVec.or_assoc]
  simp [BitVec.or_comm _ (~~~x &&& y)]
  rw [    bv32_x_distr (~~~x) y]
  rw [BitVec.or_assoc]
  rw [    bv32_x_distr x y]
  simp
  rw [    bv32_or_eq_add_three x y]
  simp [    BitVec.not_or]

theorem bv32_self_eq_neg_mul (x: BitVec 32):
  x = -x * (BitVec.allOnes 32) := by
  rw [BitVec.neg_mul]
  rw [BitVec.mul_comm]
  rw [    BitVec.neg_mul]
  simp [    BitVec.neg_one_eq_allOnes]

theorem bv32_not_self_and_not (x y : BitVec 32) :
  ~~~(x &&& ~~~x) = (~~~x &&& ~~~y) + (~~~x &&& y) + (x &&& y) + (x &&& ~~~y) := by
  rw [bv32_not_and]
  rw [BitVec.not_not]
  rw [bv32_not_or_self]
  rw [bv32_sum_all]

/--
Let x,y be 32-bit bit-vectors. Prove the equivalence of the following two expressions: $2\cdot \lnot (x\land \lnot y
    ↪)-1\cdot (x\lor \lnot y)-7\cdot \lnot (x\oplus y)+11\cdot \lnot (x\land y)-5\cdot \lnot (x\lor y)-13\cdot \
    ↪lnot (x\lor \lnot y)+6\cdot (x\land y)$, $10\cdot (x\land \lnot y)$
-/
theorem mba_challenge_02a2f35e (x y : BitVec 32) :  2#32 * ~~~(x &&& ~~~y) - 1#32 * (x ||| ~~~y) - 7#32 * ~~~(x ^^^
    ↪y) + 11#32 * ~~~(x &&& y) - 5#32 * ~~~(x ||| y) - 13#32 * ~~~(x ||| ~~~y) + 6#32 * (x &&& y) = 10#32 * (x &&&
    ↪ ~~~y) := by
  simp only [    bv32_add_neg_eq_sub] /- step 1 -/
  simp only [    bv32_neg_mul] /- step 2 -/
  simp only [bv32_not_and] /- step 3 -/
  simp only [bv32_not_or] /- step 4 -/
  sorry

lemma mba_challenge_02a2f35e_lhs_step_1 (x y : BitVec 32) : 2#32 * ~~~(x &&& ~~~y) - 1#32 * (x ||| ~~~y) - 7#32 *
    ↪~~~(x ^^^ y) + 11#32 * ~~~(x &&& y) - 5#32 * ~~~(x ||| y) - 13#32 * ~~~(x ||| ~~~y) + 6#32 * (x &&& y) = 2#32
    ↪ * ~~~(x &&& ~~~y) + -(1#32 * (x ||| ~~~y)) + -(7#32 * ~~~(x ^^^ y)) + 11#32 * ~~~(x &&& y) + -(5#32 * ~~~(x
    ↪||| y)) + -(13#32 * ~~~(x ||| ~~~y)) + 6#32 * (x &&& y) := by
  simp only [    bv32_add_neg_eq_sub]

lemma mba_challenge_02a2f35e_lhs_step_2 (x y : BitVec 32) : 2#32 * ~~~(x &&& ~~~y) + -(1#32 * (x ||| ~~~y)) + -(7#32
    ↪ * ~~~(x ^^^ y)) + 11#32 * ~~~(x &&& y) + -(5#32 * ~~~(x ||| y)) + -(13#32 * ~~~(x ||| ~~~y)) + 6#32 * (x &&&
    ↪ y) = 2#32 * ~~~(x &&& ~~~y) + -1#32 * (x ||| ~~~y) + -7#32 * ~~~(x ^^^ y) + 11#32 * ~~~(x &&& y) + -5#32 *
    ↪~~~(x ||| y) + -13#32 * ~~~(x ||| ~~~y) + 6#32 * (x &&& y) := by
  simp only [    bv32_neg_mul]

lemma mba_challenge_02a2f35e_lhs_step_3 (x y : BitVec 32) : 2#32 * ~~~(x &&& ~~~y) + -1#32 * (x ||| ~~~y) + -7#32 *
    ↪~~~(x ^^^ y) + 11#32 * ~~~(x &&& y) + -5#32 * ~~~(x ||| y) + -13#32 * ~~~(x ||| ~~~y) + 6#32 * (x &&& y) =
    ↪2#32 * (~~~x ||| ~~~~~~y) + -1#32 * (x ||| ~~~y) + -7#32 * ~~~(x ^^^ y) + 11#32 * (~~~x ||| ~~~y) + -5#32 *
    ↪~~~(x ||| y) + -13#32 * ~~~(x ||| ~~~y) + 6#32 * (x &&& y) := by
  simp only [bv32_not_and]

lemma mba_challenge_02a2f35e_lhs_step_4 (x y : BitVec 32) : 2#32 * (~~~x ||| ~~~~~~y) + -1#32 * (x ||| ~~~y) + -7#32
    ↪ * ~~~(x ^^^ y) + 11#32 * (~~~x ||| ~~~y) + -5#32 * ~~~(x ||| y) + -13#32 * ~~~(x ||| ~~~y) + 6#32 * (x &&& y
    ↪) = 2#32 * (~~~x ||| ~~~~~~y) + -1#32 * (x ||| ~~~y) + -7#32 * ~~~(x ^^^ y) + 11#32 * (~~~x ||| ~~~y) + -5#32
    ↪ * (~~~x &&& ~~~y) + -13#32 * (~~~x &&& ~~~~~~y) + 6#32 * (x &&& y) := by
  simp only [bv32_not_or]

lemma mba_challenge_02a2f35e_lhs_step_5 (x y : BitVec 32) : 2#32 * (~~~x ||| ~~~~~~y) + -1#32 * (x ||| ~~~y) + -7#32
    ↪ * ~~~(x ^^^ y) + 11#32 * (~~~x ||| ~~~y) + -5#32 * (~~~x &&& ~~~y) + -13#32 * (~~~x &&& ~~~~~~y) + 6#32 * (x
```

````
    ↪ &&& y) = 2#32 * (~~~x ||| ~~~~~~y) + -1#32 * (x ||| ~~~y) + -7#32 * (~~~x &&& ~~~y ||| x &&& y) + 11#32 *
    ↪(~~~x ||| ~~~y) + -5#32 * (~~~x &&& ~~~y) + -13#32 * (~~~x &&& ~~~~~~y) + 6#32 * (x &&& y) := by
  simp only [<Theorem here>]
```

You are proving a single step of theorem 'mba_challenge_02a2f35e'. Based on the previous steps, select exactly one
    ↪appropriate theorem from the Theorem library above and insert it in place of <Theorem here> to complete the
    ↪proof of lemma 'mba_challenge_02a2f35e_lhs_step_5'. Do not modify any code from previous theorems or lemmas.
    ↪You are also not allowed to add any other tactics to the proof body.
### Response:
You need the lemma that rewrites ~~~(x ^^^ y) into (~~~x &&& ~~~y) ||| (x &&& y). That is
bv32_not_xor_eq_or
So your step 5 becomes:
lemma mba_challenge_02a2f35e_lhs_step_5 (x y : BitVec 32) :
  2#32 * (~~~x ||| ~~~~~~y) + -1#32 * (x ||| ~~~y) + -7#32 * ~~~(x ^^^ y) + 11#32 * (~~~x ||| ~~~y) + -5#32 * (~~~x
      ↪&&& ~~~y) + -13#32 * (~~~x &&& ~~~~~~y) + 6#32 * (x &&& y) = 2#32 * (~~~x ||| ~~~~~~y) + -1#32 * (x ||| ~~~
      ↪y) + -7#32 * (~~~x &&& ~~~y ||| x &&& y) + 11#32 * (~~~x ||| ~~~y) + -5#32 * (~~~x &&& ~~~y) + -13#32 *
      ↪(~~~x &&& ~~~~~~y) + 6#32 * (x &&& y) := by
  simp only [bv32_not_xor_eq_or]
This completes the requested rewrite.
````

D. Template-Based Formal-Informal Translation

We use the following Python program to produce problem descriptions in rigorously aligned Lean-Markdown pair.

````python
import srsly
from tqdm import tqdm

output_dir = "busy_beaver/prompts_to_run"

tables = list(srsly.read_jsonl("busy_beaver/bb_with_table.jsonl"))

template = """Complete the following Lean 4 code:

```lean4
{lean_code}
```

You can make your own auxiliary corollaries and theorems to support the proof, instead of only completing the part
    ↪with the sorry. Please output the entire program and not just the last part. Please output only the program
    ↪and add no other comment, such that your answer is a compilable lean code. Make sure to reason enough to make
    ↪ your code correct.
"""

informal_tmplate = """
/--
{informal}
-/
"""

for problem in tables:
    lean_path = problem["file_name"]
    file_name = lean_path.split("/")[-1]
    out_path = f"{output_dir}/{file_name.removesuffix('.lean')}.txt"
    is_halting = True
    if "nonhalting" in file_name:
        is_halting = False
    informal_head = "Prove the following Turing Machine with the transition table " + (
        "halts." if is_halting else "never halts."
    )
    # print(out_path)
    lean_code = None
    with open(lean_path, "r") as f:
        lean_code = f.read().removeprefix("\n")
    assert lean_code != None
    informal = "/-- \n" + informal_head + "\n" + problem["table"] + "-/\n"
    with open(out_path, "w") as f:
        f.write(
            template.format(
                lean_code=lean_code.replace(
````
````

```
                    "theorem machine_", informal + "theorem machine_"
                ).replace("\n\ndef nth_cfg", "\ndef nth_cfg")
                .replace("\n\n/--", "\n/--")
                .replace("sorry\n", "sorry")
            )
        )
        pass
```

## E. More Success and Failure Cases

We demonstrate additional full record of evaluation, including problem statements in their Lean-Markdown pair and model responses for both successful and failed evaluation cases. The comprehensive record of model responses for cases can be found in the results folder of our dataset.

### E.1. Success Case

#### E.1.1. DEEPSEEK-PROVER-V2-671B/BB-4STATE-CASE107-NONHALTING

This is a trivial non-halting case, meaning there is no "HALT" entry in the transition table. It can therefore be resolved via a single structural induction. Although the proof completes, warning messages appear, such as "aesop: failed to prove the goal after exhaustive search", "(try aesop) tactic does nothing", and "this tactic is never executed". These suggest the model does not fully reason about the proof and instead relies on automated tactics, hoping some may resolve the goal.

```
import Mathlib.Computability.TuringMachine

inductive Γ
  | zero
  | one
   deriving DecidableEq

instance : Inhabited Γ := ⟨ Γ.zero ⟩

structure Stmt where
  move : Turing.Dir
  write : Γ

def Machine := Fin 4 → Γ → Option (Fin 4    Stmt)

structure Cfg where
  q : Fin 4
  Tape : Turing.Tape Γ

def init (l : List Γ) : Cfg := ⟨⟨0, by omega⟩, Turing.Tape.mk₁ l⟩

def step (M : Machine) : Cfg → Option Cfg :=
  fun ⟨q, T⟩     (M q T.head).map fun ⟨q', a⟩     ⟨q', (T.write a.write).move a.move⟩

def machine : Machine
| ⟨000, _⟩, Γ.zero  => some ⟨⟨001, by omega⟩, ⟨Turing.Dir.left, Γ.one⟩⟩
| ⟨000, _⟩, Γ.one  => some ⟨⟨001, by omega⟩, ⟨Turing.Dir.right, Γ.one⟩⟩
| ⟨001, _⟩, Γ.zero  => some ⟨⟨000, by omega⟩, ⟨Turing.Dir.right, Γ.one⟩⟩
| ⟨001, _⟩, Γ.one  => some ⟨⟨003, by omega⟩, ⟨Turing.Dir.left, Γ.one⟩⟩
| ⟨002, _⟩, Γ.zero  => some ⟨⟨001, by omega⟩, ⟨Turing.Dir.right, Γ.zero⟩⟩
| ⟨002, _⟩, Γ.one  => some ⟨⟨000, by omega⟩, ⟨Turing.Dir.right, Γ.zero⟩⟩
| ⟨003, _⟩, Γ.zero  => some ⟨⟨000, by omega⟩, ⟨Turing.Dir.left, Γ.zero⟩⟩
| ⟨003, _⟩, Γ.one  => some ⟨⟨001, by omega⟩, ⟨Turing.Dir.left, Γ.zero⟩⟩
| ⟨_+4, _⟩, _ => by omega

def nth_cfg : (n : Nat) -> Option Cfg
| 0 => init []
| Nat.succ n => match (nth_cfg n) with
                | none => none
                | some cfg =>  step machine cfg

/--
Prove the following Turing Machine with the transition table never halts.
| State | Symbol | Next State | Move | Write |
```

```
|-----|-----|-----|-----|-----|
| 000 | zero | 001 | left | one |
| 000 | one | 001 | right | one |
| 001 | zero | 000 | right | one |
| 001 | one | 003 | left | one |
| 002 | zero | 001 | right | zero |
| 002 | one | 000 | right | zero |
| 003 | zero | 000 | left | zero |
| 003 | one | 001 | left | zero |
-/
theorem machine_never_halts : ∀ n, (nth_cfg n).isSome := by
  intro n
  have h : ∀ n, (nth_cfg n).isSome := by
    intro n
    induction n with
    | zero => simp [nth_cfg, init]
    | succ n ih =>
      cases h₁ : nth_cfg n with
      | none => simp_all [nth_cfg, step, machine, Cfg, Stmt, Γ, Fin.forall_fin_succ, Fin.forall_fin_succ']
      | some cfg =>
        simp_all [nth_cfg, step, machine, Cfg, Stmt, Γ, Fin.forall_fin_succ, Fin.forall_fin_succ']
        <;>
        (try cases cfg <;> simp_all [nth_cfg, step, machine, Cfg, Stmt, Γ, Fin.forall_fin_succ, Fin.forall_fin_succ
            ↪'])
        <;>
        (try aesop)
        <;>
        (try omega)
        <;>
        (try aesop)
  exact h n
```

## E.2. Failure Case

### E.2.1. DEEPSEEK-PROVER-V2-671B/BB-2STATE-CASE102-NONHALTING

This is a case where the Turing machines state remains fixed at 000. A correct proof requires showing that the machine consistently executes the first rule of the transition table. The model attempts a single structural induction; however, in the inductive case $succ\ n\ ih$, it leaves two unsolved goals $False$. These cannot be resolved via the induction hypothesis, suggesting that the model fails to generalize inductive reasoning in this setting.

```
import Mathlib.Computability.TuringMachine

inductive Γ
  | zero
  | one
   deriving DecidableEq

instance : Inhabited Γ := ⟨ Γ.zero ⟩

structure Stmt where
  move : Turing.Dir
  write : Γ

def Machine := Fin 2 → Γ → Option (Fin 2    Stmt)

structure Cfg where
  q : Fin 2
  Tape : Turing.Tape Γ

def init (l : List Γ) : Cfg := ⟨⟨0, by omega⟩, Turing.Tape.mk₁ l⟩

def step (M : Machine) : Cfg → Option Cfg :=
  fun ⟨q, T⟩     (M q T.head).map fun ⟨q', a⟩     ⟨q', (T.write a.write).move a.move⟩

def machine : Machine
| ⟨000, _⟩, Γ.zero  => some ⟨⟨000, by omega⟩, ⟨Turing.Dir.left, Γ.zero⟩⟩
| ⟨000, _⟩, Γ.one  => none
| ⟨001, _⟩, Γ.zero  => none
```

```
| ⟨001, _⟩, Γ.one  => some ⟨⟨000, by omega⟩, ⟨Turing.Dir.left, Γ.one⟩⟩
| ⟨_+2, _⟩, _ => by omega

def nth_cfg : (n : Nat) -> Option Cfg
| 0 => init []
| Nat.succ n => match (nth_cfg n) with
                | none => none
                | some cfg =>  step machine cfg

/--
Prove the following Turing Machine with the transition table never halts.
| State | Symbol | Next State | Move | Write |
|------|------|------|------|------|
| 000 | zero | 000 | left | zero |
| 000 | one | HALT | - | - |
| 001 | zero | HALT | - | - |
| 001 | one | 000 | left | one |
-/
theorem machine_never_halts : ∀ n, (nth_cfg n).isSome := by
  intro n
  have h : ∀ n, (nth_cfg n).isSome := by
    intro n
    induction n with
    | zero =>
      simp [nth_cfg, init]
    | succ n ih =>
      simp_all [nth_cfg, step, machine, Cfg.q, Cfg.Tape, Turing.Tape.move, Option.isSome]
      <;> aesop
  exact h n
```

### E.2.2. GOEDEL-PROVER/BB-2STATE-CASE102-NONHALTING

The model produced proof of a completely unrelated theorem about sequences. This seems to be a hallucination, where the model matched a common pattern like *theorem...by sorry* and filled in a generic proof without understanding the actual task. This shows a key limitation of small provers: Without symbolic grounding or verification, they can only match surface-level patterns rather than truly reasoning.

```
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat

/-- Given a sequence { a   } with a₁ = 1 and  a    Ł ₁ = 2 a    + 1,
    prove that  a    = 2^n - 1 for all n ≥ 1 -/
theorem sequence_formula {a : ℕ → ℕ} (h1 : a 1 = 1)
    (h_rec : ∀ n : ℕ, n ≥ 1 → a (n + 1) = 2 * a n + 1) :
    ∀ n : ℕ, n ≥ 1 → a n = 2^n - 1 := by
  /-
  We need to prove that for a sequence \( \{a_n\} \) defined by \( a_1 = 1 \) and \( a_{n+1} = 2a_n + 1 \), the
      ↪general term \( a_n \) is given by \( a_n = 2^n - 1 \) for all \( n \geq 1 \).
  1. **Base Case**: For \( n = 1 \), we have \( a_1 = 1 \). This matches \( 2^1 - 1 = 1 \), so the base case holds.
  2. **Inductive Step**: Assume \( a_n = 2^n - 1 \) holds for some \( n \geq 1 \). We need to show it holds for \( n
      ↪ + 1 \).
     - By the recurrence relation, \( a_{n+1} = 2a_n + 1 \).
     - Using the inductive hypothesis \( a_n = 2^n - 1 \), we get:
       \[
       a_{n+1} = 2(2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1
       \]
     - Thus, \( a_{n+1} = 2^{n+1} - 1 \), completing the inductive step.
  By induction, \( a_n = 2^n - 1 \) holds for all \( n \geq 1 \).
  -/
  intro n hn
  induction' hn with n hn
  -- Base case: n = 1
  -- We know a 1 = 1, which matches 2^1 - 1 = 1
  simp_all
  -- Inductive step: Assume a n = 2^n - 1, prove a (n + 1) = 2^(n + 1) - 1
  cases n with
```

```
  | zero =>
    -- When n = 0, the induction hypothesis does not apply, so we directly use the initial condition
    simp_all
  | succ n =>
    -- For n = succ n, use the recurrence relation and the inductive hypothesis
    simp_all [Nat.succ_eq_add_one, Nat.pow_succ, Nat.mul_succ]
    -- Simplify the expression to match the form 2^(n + 1) - 1
    ring
    <;> omega
```

### E.2.3. Kimini-Prover-7B/bb-4state-case122-nonhalting

The model just outputs *sorry*, which means it didnt complete the proof. This might be because it couldnt figure out how to reason about the problem, or because it has seen *sorry* used as a placeholder in Lean and assumed that was enough to count as a valid answer.

```
import Mathlib.Computability.TuringMachine

inductive Γ
  | zero
  | one
   deriving DecidableEq

instance : Inhabited Γ := ⟨ Γ.zero ⟩

structure Statement where
  move : Turing Dir
  write : Γ

def Machine := Fin 4 → Γ → Option (Fin 4    Statement)

structure Cfg where
  q : Fin 4
  Tape : Turing Tape Γ

def init (l : List Γ) : Cfg := ⟨⟨0, by omega⟩, ⟨Turing Dir.right, Γ.zero⟩⟩

def step (M : Machine) : Cfg → Option Cfg :=
  fun ⟨q, T⟩     (M q T.head).map fun ⟨q', a⟩     ⟨q', (T.map a).move a⟩

def machine : Machine
| ⟨000, _⟩, Γ.zero  => some ⟨⟨001, by omega⟩, ⟨Turing Dir.right, Γ.zero⟩⟩
| ⟨000, _⟩, Γ.one  => some ⟨⟨003, by omega⟩, ⟨Turing Dir.left, Γ.zero⟩⟩
| ⟨001, _⟩, Γ.zero  => some ⟨⟨003, by omega⟩, ⟨Turing Dir.left, Γ.one⟩⟩
| ⟨001, _⟩, Γ.one  => some ⟨⟨002, by omega⟩, ⟨Turing Dir.right, Γ.zero⟩⟩
| ⟨002, _⟩, Γ.zero  => some ⟨⟨003, by omega⟩, ⟨Turing Dir.left, Γ.zero⟩⟩
| ⟨002, _⟩, Γ.one  => some ⟨⟨001, by omega⟩, ⟨Turing Dir.left, Γ.one⟩⟩
| ⟨003, _⟩, Γ.zero  => some ⟨⟨001, by omega⟩, ⟨Turing Dir.left, Γ.one⟩⟩
| ⟨003, _⟩, Γ.one  => some ⟨⟨000, by omega⟩, ⟨Turing Dir.right, Γ.zero⟩⟩
| ⟨_+4, _⟩, _ => by omega

def nth configurations (n : Nat) : Option Cfg :=
  | 0 => init []
  | Nat.succ n => match (nth configurations n) with
              | none => none
              | some cfg =>  step machine cfg

/--
Prove the following Turing Machine with the transition table never halts.
| State | Symbol | Next State | Move | Write |
|-------|--------|------------|------|
| 000 | zero | 001 | right | zero |
| 000 | one | 003 | left | zero |
| 001 | zero | 003 | left | one |
| 001 | one | 002 | right | zero |
| 002 | zero | 003 | left | zero |
| 002 | one | 001 | left | one |
| 003 | zero | 001 | left | one |
| 003 | one | 000 | right | zero |
-/
```

```
theorem machine_never_halts : ∀ n, (nth configurations n).isSome := by
  sorry
```

# F. Customized Lemma Library.

```
import Mathlib.Tactic.NthRewrite

@[simp]
theorem bv32_and_not_self(x : BitVec 32) :
  x &&& ~~~x = 0 := by
  simp

@[simp]
theorem bv32_not_not(x : BitVec 32) :
  ~~~~~~x = x := by
  simp

@[simp]
theorem bv32_or_not_self(x : BitVec 32) :
  x ||| ~~~x = BitVec.allOnes 32 := by
  simp

@[simp]
theorem bv32_not_or_self(x : BitVec 32) :
  ~~~x ||| x = BitVec.allOnes 32 := by
  simp

@[simp]
theorem bv32_neg_mul (x y : BitVec 32) :
  -x * y = -(x * y) := by
  simp

theorem bv32_not_and (x y : BitVec 32) :
  ~~~(x &&& y) = ~~~x ||| ~~~y := by
  rw [BitVec.not_and]

theorem bv32_not_or (x y : BitVec 32) :
  ~~~(x ||| y) = ~~~x &&& ~~~y := by
  rw [BitVec.not_or]

theorem bv32_not_xor_eq_or (x y : (BitVec 32)) :
  ~~~(x ^^^ y) = (~~~x &&& ~~~y) ||| (x &&& y) := by
  ext i
  simp
  cases h₁ : x[i] <;> cases h₂ : y[i]
  simp
  simp
  simp
  simp

theorem bv32_xor_eq_or (x y : (BitVec 32)) :
  (x ^^^ y) = (~~~x &&& y) ||| (x &&& ~~~y) := by
  ext i
  simp
  cases h₁ : x[i] <;> cases h₂ : y[i]
  simp
  simp
  simp
  simp

theorem bv32_x_distr (x y: BitVec 32) :
  x = (x &&& y) ||| (x &&& ~~~y) := by
  ext i
  simp
  simp [    Bool.and_or_distrib_left]

theorem bv32_y_distr (x y: BitVec 32) :
  y = (x &&& y ||| ~~~x &&& y) := by
  ext i
  simp
```

```
  simp [    Bool.and_or_distrib_right]


theorem bv32_add_assoc (x y z : BitVec 32) :
  x + y + z = x + (y + z) := by
  rw [BitVec.add_assoc]

theorem bv32_add_comm(x y : BitVec 32) :
  x + y = y + x := by
  rw [BitVec.add_comm]

theorem bv32_add_neg_eq_sub {x y : BitVec 32} :
  x + -y = x - y := by
  rw [BitVec.add_neg_eq_sub]

theorem bv32_mul_comm (x y : BitVec 32) :
  x * y = y * x := by
  rw [BitVec.mul_comm]

theorem bv32_var_mul_comm (x y z: BitVec 32) :
  (x &&& y) * z = z * (x &&& y) := by
  rw [BitVec.mul_comm]

theorem bv32_mul_add (x y z : BitVec 32) :
  x * (y + z) = x * y + x * z := by
  rw [BitVec.mul_add]

theorem bv32_neg_eq_mul (x : BitVec 32) :
  -x = x *  (BitVec.allOnes 32) := by
  rw [    BitVec.neg_one_eq_allOnes]
  rw [BitVec.mul_neg]
  rw [BitVec.mul_one]

theorem bv32_add_mul_one (x y : BitVec 32) :
  x + x * y = x * (1#32 + y) := by
  rw [BitVec.mul_add]
  rw [BitVec.mul_one]

/--
1: x &&& y
2: ~~~x &&& y
3: x &&& ~~~y
4: ~~~x &&& ~~~y
-/
theorem bv32_or_eq_add12 (x y : BitVec 32) :
  (x &&& y) ||| (~~~x &&& y) = (x &&& y) + (~~~x &&& y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ (~~~x)]
  simp [    BitVec.and_assoc]

theorem bv32_or_eq_add13 (x y : BitVec 32) :
  (x &&& y) ||| (x &&& ~~~y) = (x &&& y) + (x &&& ~~~y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ x]
  simp [BitVec.and_assoc]

theorem bv32_or_eq_add14 (x y : BitVec 32) :
  (x &&& y) ||| (~~~x &&& ~~~y) = (x &&& y) + (~~~x &&& ~~~y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ (~~~x)]
  simp [    BitVec.and_assoc]

theorem bv32_or_eq_add21 (x y : BitVec 32) :
  (~~~x &&& y) ||| (x &&& y) = (~~~x &&& y) + (x &&& y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
```

```
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ x]
  simp [    BitVec.and_assoc]

theorem bv32_or_eq_add23 (x y : BitVec 32) :
  (~~~x &&& y) ||| (x &&& ~~~y) = (~~~x &&& y) + (x &&& ~~~y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ x]
  simp [    BitVec.and_assoc]

theorem bv32_or_eq_add31 (x y : BitVec 32) :
  (x &&& ~~~y) ||| (x &&& y) = (x &&& ~~~y) + (x &&& y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ x]
  simp [    BitVec.and_assoc]
  simp [BitVec.and_assoc]

theorem bv32_or_eq_add32 (x y : BitVec 32) :
  (x &&& ~~~y) ||| (~~~x &&& y) = (x &&& ~~~y) + (~~~x &&& y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ (~~~x)]
  simp [    BitVec.and_assoc]

theorem bv32_or_eq_add41 (x y : BitVec 32) :
  (~~~x &&& ~~~y) ||| (x &&& y) = (~~~x &&& ~~~y) + (x &&& y) := by
  apply Eq.symm
  apply BitVec.add_eq_or_of_and_eq_zero
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ x]
  simp [    BitVec.and_assoc]

theorem bv32_or_eq_add_three (x y : BitVec 32) :
  (x ||| y) = (x &&& ~~~y) + (x &&& y) + (~~~x &&& y) := by
  nth_rw 1 [bv32_y_distr x y]
  nth_rw 1 [bv32_x_distr x y]
  simp [    BitVec.or_assoc]
  simp [BitVec.or_comm _ (x &&& y)]
  simp [    BitVec.or_assoc]
  rw [BitVec.or_comm (x &&& y)]
  apply Eq.symm
  rw [BitVec.add_eq_or_of_and_eq_zero]
  rw [BitVec.add_eq_or_of_and_eq_zero]
  simp [    BitVec.and_assoc]
  simp [BitVec.and_comm _ x]
  simp [BitVec.and_assoc]
  rw [BitVec.add_comm]
  rw [    bv32_or_eq_add13]
  rw [    bv32_x_distr x y]
  simp [    BitVec.and_assoc]

theorem bv32_sum_all (x y : BitVec 32) :
  (~~~x &&& ~~~y) + (~~~x &&& y) + (x &&& y) + (x &&& ~~~y) = BitVec.allOnes 32 := by
  simp [BitVec.add_comm _ (~~~x &&& y)]
  simp [BitVec.add_comm _ (x &&& _)]
  simp [    BitVec.add_assoc]
  rw [BitVec.add_eq_or_of_and_eq_zero]
  rw [    bv32_or_eq_add_three x y]
  nth_rw 1 [bv32_x_distr x y]
  simp [BitVec.or_comm _ y]
  nth_rw 1 [bv32_y_distr x y]
  simp [    BitVec.or_assoc]
  simp [BitVec.or_comm _ (x &&& y)]
  simp [    BitVec.or_assoc]
  simp [BitVec.or_comm _ (~~~x &&& _)]
  simp [    BitVec.or_assoc]
  simp [BitVec.or_comm _ (~~~x &&& y)]
```

```
  rw [    bv32_x_distr (~~~x) y]
  rw [BitVec.or_assoc]
  rw [    bv32_x_distr x y]
  simp
  rw [    bv32_or_eq_add_three x y]
  simp [    BitVec.not_or]

theorem bv32_self_eq_neg_mul (x: BitVec 32):
  x = -x * (BitVec.allOnes 32) := by
  rw [BitVec.neg_mul]
  rw [BitVec.mul_comm]
  rw [    BitVec.neg_mul]
  simp [    BitVec.neg_one_eq_allOnes]

theorem bv32_not_self_and_not (x y : BitVec 32) :
  ~~~(x &&& ~~~x) = (~~~x &&& ~~~y) + (~~~x &&& y) + (x &&& y) + (x &&& ~~~y) := by
  rw [bv32_not_and]
  rw [BitVec.not_not]
  rw [bv32_not_or_self]
  rw [bv32_sum_all]
```