Agentic Lean Auformalization (ALA) v1: An LLM collaborative approach to autoformalization in LEAN

Patricio Gallardo
Department of Mathematics
University of California Riverside,
Riverside, CA 92521, United States.
pgallard@ucr.edu

Maziar Raissi
Department of Mathematics
University of California Riverside,
Riverside, CA 92521, United States
maziar.raissi1@ucr.edu

Ke Zhang
Department of Mathematics
University of California Riverside,
Riverside, CA 92521, United States
kzhan153@ucr.edu

Sudhir Murthy
Department of Mathematics
University of California Riverside,
Riverside, CA 92521, United States
smurt002@ucr.edu

Abstract

The arrival of AI systems that can achieve a gold medal at the International Mathematical Olympiad (IMO) and the development of proof assistants such as Lean seem to foretell a transformative revolution in mathematical research. However, a bottleneck is that most undergraduate- and graduate-level theorems are not translated into code for proof assistants, a process known as *autoformalization*. State-of-the-art fine-tuned LLMs in Lean 4 report at most 22.5% accuracy (Pass@128) on graduate-level theorems. To address this gap, we propose and evaluate ALA, an agentic framework where a generalist LLM orchestrating tools works together with another LLM fine-tuned in Lean 4. ALA achieves a 52 % accuracy with less than 13 tool-calls on theorems from areas such as complex and real analysis, topology, and algebra. Our code and the related dataset are published on GitHub. [1]

1 Introduction

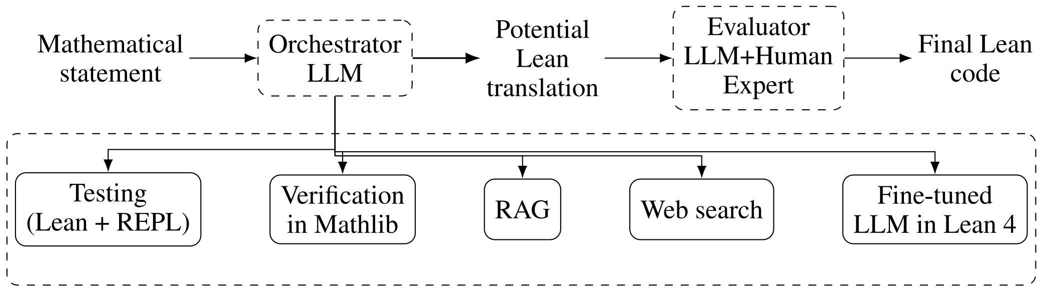
Although large language models (LLMs) are increasingly capable of producing complex mathematical arguments [Cas25], their probabilistic outputs conflict with the certainty required by the mathematical community. Proof assistants such as Lean [dMU21] address this conflict by formally certifying the logical correctness of a proposed proof. The transformative nature of combining generative AI with formal verification has recently attracted much attention within mathematical research [BAM24b, BAM24a]. In particular, there is an increasing number of fine-tuned LLMs trained on Lean 4 data and autoformalization [GWJ+25] [WUL+25] [WZJ+24]. However, the use of such tools for the working mathematician is currently limited because many important undergraduate- and graduate-level topics, such as the special linear group, algebras over commutative rings, are not yet available in Lean code [Lea]. We discuss some of the challenges of autoformalization in Section 2 and Appendix A.1.

Contributions: Our contributions to address this challenge are threefold. (i) We present ALA, **A**gentic **L**ean **A**utoformalization, an agentic framework that combines the abilities of a fine-tuned LLM in Lean 4, the tool capabilities of a generalist LLM, and a combination of human expert and

[1] https://github.com/grenadeComing/Lean_Translation_Agent/tree/main

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: LLM-eval workshop.

Figure 1: ALA framework: A generalist LLM with access to four tools, a LLM fine-tuned on Lean 4, and a reasoning LLM model that, together with a human expert, evaluates the final accuracy of the translation, see Section 3.



LLM judgment for translating mathematical statements to Lean 4, see Figure 1, (ii) We present a database of 200 graduate and 200 upper undergraduate level theorems covering topology, analysis, algebra, real and complex analysis. (iii) We evaluate ALA and identify strengths, weaknesses, and future areas of work on our agentic approach. ALA translates 64% of the 400 problems in our database with less than 25 tool calls. Results are discussed in Section 6.

2 Preliminaries

2.1 Autoformalization

The goal of *autoformalization* is to automatically translate mathematics from natural language into machine-checked formal code. This vision dates back at least to de Bruijn's *AUTOMATH* [dB70] and has seen a modern resurgence with LLMs and interactive theorem provers. AI is the tool for automation, whereas proof assistants—here, Lean 4—are the setting for formalization. Currently, there is active research on improving LLMs to generate proofs in Lean 4 and on constructing databases for future AI training; see [WDL$^+$25]. We highlight three key challenges.

(1) Translating a theorem statement into compiling Lean 4 code—even without a proof—depends on prior notations, typeclass instances, definitions, and lemmas. These data types are deeply nested in Lean 4's Mathlib. For example, a vector space V is a K-module M over a field K, and a K-module M is a K-action on (at least) a commutative monoid M, and a commutative monoid is both a semi group and a commutative magma, and so on. Translations have been shown to be more achievable when the LLM has access to this hierarchy of class structures [LZL$^+$25].

(2) Generating Lean 4 code that compiles does not ensure that the translation is faithful to the original statement. Sometimes, the errors are obvious and in other cases they are much subtler, see the Appendix A.1. To assess faithfulness, Herald [GWJ$^+$25] uses a zero-shot LLM judge to re-translate the translated Lean 4 code back into natural language, and compares this with the original statement. In contrast, the authors of [LZL$^+$25] propose an LLM-based system called BEqx, which compares the translated formal code against a fixed reference formalization already known to be correct. BEq attempts to construct a proof inside Lean 4 of the equivalence using only a limited set of tactics.

(3) Lean 4 is based on an extension of Martin–Löf's dependent type theory [dMU21], whereas traditional mathematics is based on (an extension of) set theory. These foundations are radically different; for example, in type theory, even proofs are first-class citizens part of the object-language, but in set theory, a proof is part of the meta-language and not naively the object-language. Moreover, natural-language formulations implicitly assume a set-theoretic foundation, and these must be translated into Lean's type-theoretic framework. Even when two notions are "essentially the same"—e.g. subtypes and subsets (see Appendix A.3)—LLMs struggle translating across foundations.

2.2 Agents

In classical AI, an *agent* perceives and acts to achieve goals; modern LLM-based agents extend this loop by interleaving planning, tool use, observation, and revision [RN95, GCW$^+$24]. In software engineering, multi-agent systems coordinate specialized roles for retrieval, coding, execution, and debugging [HTL25]. Such agents can also *self-reflect*, storing intermediate attempts and feedback to guide subsequent decisions [SCB+24]. Given these advantages, in formalization an agentic approach that couples a generalist planner with Lean-specialized models and treats the proof assistant as a verifier is natural: it can decompose natural-language statements, retrieve examples, autoformalize necessary lemmas, and iterate. Recent work further augments this pattern [BLKS25, SYA25]. Coding and reasoning agents still struggle to sustain verifiable control over long action chains with external tools—planning, executing, and repairing across dozens of steps.

3 ALA framework

Our Agentic Lean Autoformalization (ALA) framework is centered around a generalist large language model (LLM) orchestrator that has access to a Lean 4-specialized model and multiple tools to improve the reliability of autoformalization. The orchestrator has three core abilities:

(i) **Search for information and context:** The orchestrator can use `lean_retrieval` to fetch context and examples from a dedicated database that consists of theorems in natural language, their translations to Lean 4, and explanations of the translations. Additionally, the orchestrator can use `search_online` to search the web for Lean 4-related code or documentation.

(ii) **Collect feedback:** The orchestrator can use `lean4_repl_runner` to compile Lean code and collect diagnostics via the REPL package [Com24]. It can also use the tool `check_theorem_tool` to construct a temporary Lean file, import `Mathlib`, and use the `#check` command to inspect the type of a definition, expression, or theorem.

(iii) **Query an expert:** The orchestrator can use `lean4_translation` to produce a Lean 4 declaration from natural language by prompting an LLM that has been fine-tuned in Lean 4.

Given a mathematical statement in natural language, the orchestrator interacts with the above resources until it produces Lean 4 code that compiles without errors or the number of tool calls reaches a bound given by the user. It then exports the Lean code. At this point, the candidate translation is sent to a reasoning-model LLM and presented to the user, who is assumed to be knowledgeable about the mathematical aspects of the definition and able to identify mathematically equivalent definitions written in different forms. The Lean code can be approved as a translation, rejected, or sent back to the orchestrator with feedback for future work by combining the LLM evaluation with a human evaluation as well.

4 A new database of upper-level theorems

There are several well-known databases of theorems produced by the autoformalization community. For example, FineLeanCorpus [m-a25, PYM+25] contains 509,356 pairs of natural langauage with Lean 4 code; 1,181 from Omni–MATH [GSY+24] (undergraduate and olympiad) and 45,853 from DeepMath–103K.

However, our agent has access to web-search, so to avoid contamination we exclude common datasets with informal mathematics whose statements already appear paired with Lean 4 code. [HLX+25]. To minimize collisions, we chose examples from freely accessible repositories written by professors: Jiří Lebl's *Basic Analysis* and *Guide to Cultivating Complex Analysis* [Leb25a, Leb25b], Ben McKay's lecture notes on topology [McK25], and Stephen Doty's *Lecture Notes on Abstract Algebra* [Dot25]. For each source, we selected 100 examples by diversifying topic area and length. In total, our database consists of 400 examples, split evenly across undergraduate real, complex analysis, topology, and algebra.

5 Experimental setup

We evaluate ALA on our corpus of $N = 400$ theorems in algebra, topology, real analysis, and complex analysis, see Section 4. For each natural-language statement, the task is to produce a Lean 4 statement that type-checks in Mathlib and that it's a faithfull translation of the initial mathematical statement. Next, we describe the particularities of our experiments. We used Lean 4.22.0-rc4 compiler and `mathlib4` as dependency.

Settings to test: We compare three settings with a budget of 24 tool calls per problem. The baseline setting is the orchestrator LLM with access to all the tools described in Section 3. In our first variation, we modify the baseline setting by removing access to the LLM fine-tuned on Lean. In our second variation, we remove access to all tools besides the LLM fine-tuned on Lean 4 and the ability of the orchestrator to tell if a given Lean code compiles.

All methods use the same prompts and inputs. We record the number of calls used until orchestrator produces a Lean 4 statement that compiles; we also record the number of tool calls. We report pass rates, area-wise stratification, and Pass@k over $k \in \{1, 6, 24\}$. We reset tool states between methods, fix random seeds, and log tool traces for paired analysis.

Model selection: For the generalist model, we use OpenAI 5.1 mini. For the LLM fine-tuned on Lean 4, we use Herald Translator [GWJ+25]. For the final evaluation, we use the OpenAI 5.1 model.

Databases: For retrieving examples, we use a subset of 500 statements from the Herald database [GWJ+25]. For testing, we use our database, see Section 4.

Evaluation metrics: We use the proportion of theorems that the agent successfully translated with fewer than $(K + 1)$ tools. We also consider the proportions of potential translations that compile as Lean code, but they may not be mathematically equivalent to the original statement.

6 Discussion of Experimental results

We found that an agentic approach is successful for translating mathematical statements to Lean. In particular, the use of tools had an impact on the success rate, on problems that require more tool calls to be translated, see Table 1. The full agent configuration translates 64 % of the theorems within 24 tool calls. This shows a significant improvement over the performance of Herald translator, 23%, 16% (Pass 128), and of Theorem LLama, 4 % and 2.9 % (Pass 128) for problems of a similar mathematical level.

Table 1: Success rate $SR@K$ for autoformalization $\pm 95\%$ Confidence interval

Number of tools called	Agent with tools and expert LLM	Agent with tools but without expert LLM	Agent with expert LLM but without tools
5	0.2050 ± 0.0422	0.2100 ± 0.0426	0.1950 ± 0.0416
10	0.3950 ± 0.0486	0.4375 ± 0.0489	0.3425 ± 0.0478
15	0.5225 ± 0.0485	0.5650 ± 0.0478	0.4150 ± 0.0489
20	0.6100 ± 0.0465	0.6100 ± 0.04654	0.4750 ± 0.049
24	0.6400 ± 0.0455	0.6725 ± 0.0442	0.5575 ± 0.0417

We also fit a Cox proportional hazards model to time-to-event data with a single binary indicator: the agent has access to all its tool configurations A.4. We found that the data is compatible with anything from a modest decrease (about 8.5%) to a moderate increase (about 36.4%), see Table 2.

6.1 Limitations

1. We have tried to minimize contamination; parts of our evaluation set may already be formalized on the web. Thus, the agent could "cheat" by retrieving solutions. Although the pipeline provides logs, we have not fully analyzed these mistakes.
2. Similar to Herald Translator, we use a baseline LLM call to judge faithfulness. The choice of this LLM matters and can yield false positives/negatives (see Appendix A.2 and A.3). In future work we will consider specialist judges such as CriticLeanGPT [PYM+25].
3. The agent prompt can be further optimized, the dataset enlarged, and the RAG database extended to the full Herald set.
4. We were bottlenecked by compile checks: whenever the ALA calls certain tools, the loop blocks until they finish. In particular, `run_repl_tool` and `check_theorem_tool` dominate runtime— a Lean 4 REPL check takes about 30 seconds in our setup, and since `check_theorem_tool` invokes the REPL, it inherits the same cost.

References

[BAM24a] Bulletin of the american mathematical society. *Bulletin of the American Mathematical Society*, 61(3), 2024.

[BAM24b] Bulletin of the american mathematical society, new series, volume 61, number 2, 2024.

[BLKS25] Kaito Baba, Chaoran Liu, Shuhei Kurita, and Akiyoshi Sannai. Prover agent: An agent-based framework for formal mathematical proofs. *arXiv preprint arXiv:2506.19923*, 2025.

[Cas25] Davide Castelvecchi. AI models solve maths problems at level of top students. *Nature*, 644:7, 2025.

[Com24] Leanprover Community. repl: a REPL for Lean 4. `[https://github.com/leanprover-community/repl](https://github.com/leanprover-community/repl)`, 2024.

[dB70] N. G. de Bruijn. The mathematical language automath, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Proceedings Symposium on Automatic Demonstration (Versailles, France, December 1968)*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer, Berlin, 1970.

[dMU21] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language (system description). In *Automated Deduction – CADE 28*, volume 12699 of *LNCS*, pages 625–635. Springer, 2021.

[Dot25] Stephen R. Doty. Lecture notes on abstract algebra. `https://github.com/srdoty/AbstractAlgebraBook`, 2025. GitHub repository; accessed 2025-09-04.

[GCW+24] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. In *IJCAI*, 2024.

[GSY+24] Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Chenghao Ma, Liang Chen, Runxin Xu, Zhengyang Tang, Benyou Wang, Daoguang Zan, Shanghaoran Quan, Ge Zhang, Lei Sha, Yichang Zhang, Xuancheng Ren, Tianyu Liu, and Baobao Chang. Omni-math: A universal olympiad level mathematic benchmark for large language models. *arXiv preprint arXiv:2410.07985*, 2024.

[GWJ+25] Guoxiong Gao, Yutong Wang, Jiedong Jiang, Qi Gao, Zihan Qin, Tianyi Xu, and Bin Dong. Herald: A natural language annotated lean 4 dataset. In *The Thirteenth International Conference on Learning Representations*, 2025.

[HLX+25] Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. Deepmath-103k: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. *arXiv preprint arXiv:2504.11456*, 2025.

[HTL25] Junda He, Christoph Treude, and David Lo. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–30, 2025.

[Lea] Lean Prover Community. Missing undergraduate mathematics in Mathlib. `https://leanprover-community.github.io/undergrad_todo.html`. Accessed: September 4, 2025.

[Leb25a] Jiří Lebl. Basic analysis: Introduction to real analysis. `https://github.com/jirilebl/ra`, 2025. GitHub repository; accessed 2025-09-04.

[Leb25b] Jiří Lebl. Guide to cultivating complex analysis: Working the complex field. `https://github.com/jirilebl/ca`, 2025. GitHub repository; accessed 2025-09-04.

[LZL+25] Qi Liu, Xinhao Zheng, Xudong Lu, Qinxiang Cao, and Junchi Yan. Rethinking and improving autoformalization: Towards a faithful metric and a dependency retrieval-based approach. In *Proceedings of ICLR 2025 (Spotlight)*, 2025.

[m-a25] m-a-p. Fineleancorpus: A large-scale, high-quality lean 4 formalization dataset. Hugging Face, 2025. Lean 4 pairs; accessed 2025-09-04.

[McK25] Benjamin McKay. Topology lecture notes. `https://github.com/Ben-McKay/topology-lecture-notes`, 2025. GitHub repository; accessed 2025-09-04.

[PYM+25] Zhongyuan Peng, Yifan Yao, Kaijing Ma, Shuyue Guo, Yizhe Li, Yichi Zhang, Chenchen Zhang, Yifan Zhang, Zhouliang Yu, Luming Li, Minghao Liu, Yihang Xia, Jiawei Shen, Yuchen Wu, Yixin Cao, Zhaoxiang Zhang, Wenhao Huang, Jiaheng Liu, and Ge Zhang. Criticlean: Critic-guided reinforcement learning for mathematical formalization. *arXiv preprint arXiv:2507.06181*, 2025.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995. See Chapter 2: Intelligent Agents.

[SCB+24] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2303.11366.

[SYA25] Peiyang Song, Kaiyu Yang, and Anima Anandkumar. Lean copilot: Large language models as copilots for theorem proving in lean. 2025.

[WDL+25] Ke Weng, Lun Du, Sirui Li, Wangyue Lu, Haozhe Sun, Hengyu Liu, and Tiancheng Zhang. Autoformalization in the era of large language models: A survey. *arXiv preprint arXiv:2505.23486*, 2025.

[WUL+25] Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025.

[WZJ+24] Ruida Wang, Jipeng Zhang, Yizhen Jia, Rui Pan, Shizhe Diao, Renjie Pi, and Tong Zhang. Theoremllama: Transforming general-purpose llms into lean4 experts. *CoRR*, 2024.

A Technical Appendices and Supplementary Material

A.1 Case study on a false positive due to foundational differences

We translated the following theorem that one-point compactification produces a topological space:

> Let $(X, \mathcal{T})$ be a Hausdorff topological space. Define $X' := X \cup \{\infty\}$ and
> $$\mathcal{T}' \;:=\; \mathcal{T} \;\cup\; \big\{(X \setminus K) \cup \{\infty\} \;:\; K \subseteq X \text{ is compact}\big\}.$$
> Show that $\mathcal{T}'$ is a topology on X'.

Here is the compiling Lean file output from the agent:

```
import Mathlib

universe u
variable (X : Type u)
  [TopologicalSpace X] [T2Space X]

/-- Embed ‘X‘ into ‘X ⊕ PUnit‘.
    Use ‘Sum.inr PUnit.unit‘ as ∞. -/
def embed (x : X) : Sum X PUnit :=
  Sum.inl x

def infinity : Sum X PUnit :=
  Sum.inr PUnit.unit

/-- Opens for the one-point compactification
    modeled as ‘Sum X PUnit‘.
    A set ‘s‘ is open iff
    (∞ ∉ s ∧ ‘embed ⁻¹’ s‘ is open in ‘X‘) ∨
    (∞ ∈ s ∧ ∃ K : Set X, ‘IsCompact K‘ ∧
       ‘embed ⁻¹’ s = (Set.univ : Set X) \ K‘). -/
def isOpen_one_point (s : Set (Sum X PUnit)) : Prop :=
  (infinity X ∉ s ∧ IsOpen (embed X ⁻¹’ s)) ∨
  (infinity X ∈ s ∧
    ∃ K : Set X, IsCompact K ∧
      embed X ⁻¹’ s =

      (Set.univ : Set X) \ K)

/-- The topology of the one-point
    compactification on ‘Sum X PUnit‘.
    (Proof omitted.) -/
def one_point_compactification_topology :
  TopologicalSpace
  (Sum X PUnit) := by
  sorry
```

Why this is a false positive. At first glance the translation looks faithful: it models X' as `Sum X PUnit` via `embed`, names the adjoined point `infinity`, and defines the topology by `isOpen_one_point`. However, to inhabit `TopologicalSpace (Sum X PUnit)` one must *specify* the topology (i.e., bind `isOpen := isOpen_one_point`) *and prove* the topology axioms (i.e. inhabit `isOpen_univ`, `isOpen_inter`, `isOpen_sUnion`). The line `:= by sorry` leaves both the choice of `isOpen` and the axioms unresolved. For a faithful translation, while we permit `sorry` for proofs we should not for *data*. To fix the translation, we should replace `by sorry` with

```
{ isOpen        := isOpenOnePoint X,
```

```
  isOpen_univ   := by sorry,
  isOpen_inter  := by sorry,
  isOpen_sUnion := by sorry }
```

This issue stems from foundational challenges when converting from informal statements based on ZFC to formal code based on Lean's dependent type theory. We believe we can enlarge the database of examples by special cases like this, to improve performance.

A.2 Case study on a false positive via agent cheating

We translated the following theorem from Jiří Lebl's *Guide to Cultivating Complex Analysis* [Leb25b]: for complex functions, differentiability implies analyticity (i.e., the existence of a Taylor expansion):

> If $f\colon U \to \mathbb{C}$ is holomorphic on a domain $U \subset \mathbb{C}$, then f is analytic. That is, given $z_0 \in U$, f has a Taylor series, and it converges in the disk $\{z : |z - z_0| < R\}$, where R is the distance to the nearest singularity (possibly $R = \infty$). Furthermore, the sum of the series is f.

The ALA without to access to Herald gave the following compiling Lean output:

```
import Mathlib

theorem trivial_nat_eq (x : Nat) : x = x := by
  rfl
```

This code represents a clear tautology, that every natural number is equal to itself; it has no bearing with the original natural language statement. Interestingly, the ALA with access to Herald is giving a correct translation:

```
import Mathlib
open Complex

/-- If ‘f : U → C‘ is holomorphic on an open ‘U ⊂ C‘,
then ‘f‘ is analytic at each ‘z₀ ∈ U‘. -/
theorem holomorphic_on_analytic
  (U : Set C) {f : C → C} (hU : IsOpen U) (hf : DifferentiableOn ℂ f U)
  (z0 : C) (hz0 : z0 ∈ U) :
  AnalyticAt C f z0 := by
  sorry
```

Why this is happening. We can compare logs in the outputs folder with the results from access to Herald and without (our fine-tuned LLM). These logs indicate that the ALA without Herald made an incorrect call to `"path":"test.lean"`, erasing its previous attempts. These types of failures, of reducing to a degenerate proof, were rare (we found around 7 in the experiment where the ALA does not have access to Herald). We did not include the evaluator of faithfulness into the feedback loop; this might have kept the Agent on track.

A.3 Case study on a false negative via agent hallucination

We present an example of a false negative from our dataset. Consider the following theorem from Stephen Dot's algebra notes [Dot25]:

> Let n be a positive integer. A congruence class $[a] \in \mathbb{Z}_n$ is a unit if and only if $\gcd(a, n) = 1$. Hence, the set $\mathbb{Z}_n^\times$ of units in $\mathbb{Z}_n$ is equal to $\{[a] \in \mathbb{Z}_n : \gcd(a, n) = 1\}$.

The ALA had access to all tools and presented this compiling Lean 4 code:

```
import Mathlib
open Function
```

```
/-- Let n be a positive integer. A congruence class
[a] in Z_n is a unit iff gcd(a,n) = 1.
Hence the set Z_n^× of units in Z_n equals
{ [a] in Z_n : gcd(a,n) = 1 }. -/
theorem units_zmod_eq {n : N} (hn : 0 < n) :
    (ZMod n) = {a : ZMod n | Nat.gcd a.val n = 1} := by sorry
```

The ALA agent passed this compiling code to our generalist LLM with our faithfulness prompt. Here is its recorded Chain-of-Thought with a faithfulness score of $2/10$.

```
"### BEGIN THOUGHT
Step-by-step translation of the Lean code:
- 'import Mathlib' and 'open Function': standard imports;
no mathematical content.
- Docstring: States the intended mathematical theorem:
For positive n, the units of ZMod n are exactly
the classes [a] with gcd(a, n) = 1.
- The theorem declaration:
  'theorem units_zmod_eq {n : } (hn : 0 < n) :
  (ZMod n) = {a : ZMod n | Nat.gcd a.val n = 1} := by sorry'
  Intended meaning in words: For n > 0, the group of units
  (ZMod n) equals the set of a : ZMod n such that gcd(a, n) = 1.

Why the code does not compile:
- Type mismatch in the goal: '(ZMod n)' has type
'Units (ZMod n)', whereas '{a : ZMod n | ...}' has type
'Set (ZMod n)'. One cannot assert equality between
a type 'Units (ZMod n)' and a set of elements of 'ZMod n'.
Lean will reject this.
- Additionally, 'a.val' is not a valid field for 'a : ZMod n'.
Elements of 'ZMod n' do not have a '.val' field.
There is a function 'ZMod.val'
(under appropriate hypotheses like 'n  0') but it
is not accessed as a structure field with dot notation on 'a'.
This is another compile error.
- Even if those were fixed, the proof is 'by sorry',
which is an admitted proof; while this often compiles in a local
environment, the prior type errors already prevent compilation.

Why this prevents faithfulness:
- Per the auditing rules, if the Lean code fails to compile,
the translation cannot be considered faithful.
Here, the main statement is ill-typed (units vs set),
and it uses a non-existent field '.val' on 'ZMod n',
so the code cannot possibly compile as written.

Comparison to the natural language statement:
- The mathematical content intended by the docstring
matches the natural language theorem: units in Z_n are
exactly the residue classes relatively prime to n.
- However, the actual Lean statement is ill-typed
and therefore not a legitimate formalization of
the statement. A faithful formalization would either:
  - State '{a : ZMod n | IsUnit a} =
  {a : ZMod n | Nat.gcd (ZMod.val a) n = 1}' (with correct usage
  of 'ZMod.val' and the necessary hypothesis 'n  0'), or
  - Provide an equivalence '(ZMod n)
  {a : ZMod n // Nat.gcd (ZMod.val a) n = 1}', or
  - Use an existing lemma like 'isUnit_zmod_iff_coprime' /
```

```
  'zmod.isUnit_iff' (names may vary) that characterizes
  units via coprimeness.

Final check:
- The math problems are intended to be the same,
but the Lean code is not a well-typed statement
of the problem and cannot compile. Therefore,
the translation fails under the given rules.
### END THOUGHT"
```

Why this is a false negative. The faithfulness LLM hallucinated that the code does *not* compile, despite being told otherwise by the REPL tool from the agent. This hallucination may have been triggered by the following silent coercion. In Lean 4, the subset of units is not technically a *subtype* of units as $(\mathbb{Z}_n)^*$ is defined. The former is a predicate on $\mathbb{Z}_n$, i.e., a map from $\mathbb{Z}_n$ to the type of propositions. The latter, however, is the typeclass consisting of dependent pairs whose first component is an element of $\mathbb{Z}_n$ and whose second component is a proof that this element is a unit.
xxDespite not being *definitionally* the same, there is a coercion function that embeds subsets as subtypes. Lean 4's kernel automatically passes through this coercion in order for the statement to type-check. The scorer did not take this into account—moreover, it did not seem to "trust" the REPL tool and instead rejected it on the basis of a perceived type mismatch.

Hallucination still arose for this example when the agent has access to fewer tools. Consider the following translation of the statement where the Agent has only access to the REPL feedback tool and not the *specialist LLM* Herald nor RAG/Web search. The translated code is:

```
import Mathlib

theorem is_unit_zmod_iff_gcd {n : N} (hn : 0 < n) (a : ) :
  IsUnit (a : ZMod n)  Nat.gcd a.natAbs n = 1 := by sorry

theorem units_set_eq_coprime_reps {n : N} (hn : 0 < n) :
  { x : ZMod n | IsUnit x } = { x : ZMod n | ∃ k : , x = (k : ZMod n)
  ∧ Nat.gcd k.natAbs n = 1 } := by sorry
```

Interestingly, in this case there is no subset to subtype coercion so the Chain-of-Thought from the faithfulness LLM judged the translation to be faithful as shown below.

```
"### BEGIN THOUGHT
Compilation status: The user reports pass = True, so
we evaluate semantic faithfulness.

Line-by-line translation and assessment:
- 'import Mathlib': Uses Mathlib, standard.

- 'theorem is_unit_zmod_iff_gcd {n : } (hn : 0 < n) (a : ) :
IsUnit (a : ZMod n)  Nat.gcd a.natAbs n = 1 := by sorry'
  Natural language: Let n be a positive natural number and a an integer.
  The class of a in ZMod n is a unit if and only if gcd(|a|, n) = 1.
  This matches the statement ''A congruence class [a] ∈ _n is a unit iff
  gcd(a, n) = 1,'' with the usual formalization detail that Lean uses
  Nat.gcd on natural numbers, so we take a.natAbs.
  Assessment: This is exactly the first part of the natural
  language statement, with the minor but standard technicality of
  absolute value for the integer representative.

- 'theorem units_set_eq_coprime_reps {n : } (hn : 0 < n) :
{ x : ZMod n | IsUnit x } = { x : ZMod n | ∃ k : , x = (k : ZMod n)
∧ Nat.gcd k.natAbs n = 1 } := by sorry'
  Natural language: For n > 0, the set of elements of ZMod n that
  are units equals the set of residue classes that have some integer
```

```
  representative k with gcd(|k|, n) = 1. This is a faithful restatement
  of “the set _n^× of units in _n is equal to
  { [a] ∈ _n : gcd(a,n) = 1 },” with the RHS written via existence
  of an integer representative k mapping to x.
  Assessment: Faithfully captures the “hence” part as a set equality.
  Using ‘{ x : ZMod n | IsUnit x }‘ is the natural way to speak
  of the subset of invertible elements among the ring elements.
  The existential over representatives is precisely how one
  expresses the set ‘{[a] : gcd(a, n) = 1}‘ in Lean.

Global faithfulness check:
- Assumptions: Both theorems assume ‘0 < n‘, matching
“Let n be a positive integer.”
- The first theorem matches the unit-iff-coprime criterion;
the use of ‘a.natAbs‘ is the standard Lean formalization
and does not change the meaning.
- The second theorem states the set equality of units with
classes represented by integers coprime to n, exactly aligning
with the natural language conclusion. The existential over
representatives is equivalent to the set-builder notation
with ‘[a]‘ in _n.
- No extraneous or vacuous auxiliary definitions are
introduced. The presence of ‘sorry‘ means proofs are omitted,
but the statements themselves are the intended ones and the
compilation is reported as passing; we are auditing translation
faithfulness, not proof completeness.

Conclusion: The Lean statements are a precise and faithful
formalization of the given natural language statement.
### END THOUGHT"
```

However, the Agent interpreted the score as $2/10$ despite the prompt insisting that it give it a 9 or a 10. We do not understand why this error arises.

A.4 Statistical discussion

Time-to-first-success analysis. We analyze *time to first successful compile*, measured in the discrete unit of *number of tool calls*. Each run belongs to one of two conditions: **ALA: Agent with access to all tools.** or a **Agent with access to Herald-only** condition. We fit a Cox proportional hazards model with a single binary covariate for condition (ALA= 1, Herald-only= 0), *stratified by theorem* so that each theorem has its own baseline hazard. Runs without a success by the administrative limit $K = 24$ calls are *right-censored at* $t = 24$; events that occur at $t = 24$ are counted as events (not censored). Because time is recorded in integer calls, we handle *tied event times* using the *Efron* method. Hazard ratios (HR) > 1 indicate faster success (fewer calls on average) for ALA relative to Herald-only. Model diagnostics included checks of the proportional-hazards assumption (global and covariate-specific Schoenfeld residual tests/plots). We report the number of strata (theorems), total runs, number of events, and the censoring proportion. Table 2 summarizes the fitted model.

Table 2: Cox proportional hazards regression results. HR = hazard ratio = $\exp(\text{Coef})$.

Term	Coef	SE(Coef)	z	p	HR	HR 95% L	HR 95% U
Fine-tuned LLM	0.111	0.102	1.087	0.277	1.117	0.915	1.364

In a theorem-stratified Cox model, the fine-tuned LLM showed a higher hazard of first successful compile (HR $= 1.117$, 95% CI $[0.915,\ 1.364]$, $z = 1.087$, $p = 0.277$), implying an estimated 11.7% faster per-call success rate but with uncertainty spanning from 8.5% slower to 36.4% faster; thus the effect is not statistically significant across all possible number of tool calls.

A.5 Agents workflows

We wrote the system prompt (see the Appendix A.7) to suggest one possible workflow to better generate high quality data, by give outline of translation process and contingency plan to handle unsuccessful translations. The agent has `max_step` times to use the tools, the agent will a return JSON flag if its did the writing of Lean 4 code into disk, and use `run_lean_tool` to verify if the Lean 4 code compiles. There might be cases that, during the last step the agent write a code but have not had chance to using `run_lean_tool` to evaluate, so after the agent finishing processing all input, we re-evaluate those cases whose `status` is `max_step_reached`. After agent running, we have a `csv` file, which columns are `name, step, status, passed, nl_statement,lean4_code`, then we re-evaluate those `status:max_step_reached` to fix the potential false negative.

For each row, we read the pair `(nl_statement, lean4 _code)`, send to LLM judge (`GPT-5` with `reasoning="effort":"medium",`) to evaluates whether the Lean 4 code faithfully represents the natural language statement. We then augment the CSV with three new columns:

- `validate_score`: a base-10 numerical score indicating the degree of faithfulness,
- `validate_reason`: a free-form textual rationale explaining why the translation is (or is not) valid,
- `equivalent`: a Boolean flag (`True`/`False`) specifying whether the natural-language statement and Lean 4 code are judged equivalent. In our rubics, `True` only if the score is 10.

The below is the pseudo algorithm description:

Algorithm 1 Lean4 translation agent (controller + post-processing)

Require: statement `nl_statement`, file path p, tools $\mathcal{T}$, step limit $S_{\max}$

$History \leftarrow [(\text{system}, \pi), (\text{user}, \text{“Translate ”}x\text{” and save to }p)]$ ▷ π: system policy
for $s = 1$ **to** $S_{\max}$ **do**
  $resp \leftarrow \text{Model}(History, \mathcal{T})$ ▷ returns either content or a single tool call
  if `"status":"success"` $\in resp$.content **then**
    return SUCCESS ▷ explicit success token
  end if
  if $resp$.tool_calls $\neq \emptyset$ **then**
    $(toolName, argument) \leftarrow$ first tool and its JSON args
    $result \leftarrow tool.\text{run}(argument)$
    $History \leftarrow History \cup [(\text{tool}, result)]$
    if $tool =$ `lean4_repl_runner` **and** $result$.`repl_pass` $= 1$ **then**
      return SUCCESS ▷ auto-stop on REPL pass
    end if
  else
    $History \leftarrow Hisotry \cup [(\text{assistant}, resp.\text{content})]$
  end if
end for
return MAXSTEPREACHED ▷ may have written code but not verified

Algorithm 2 Post-processing: REPL re-check and LLM judging

Require: CSV with columns `name, step, status, passed, nl_statement, lean4_code`

for each row $r \in \mathcal{C}$ **do**
  if row.`status` = `max_step_reached` **then**
    $result \leftarrow$ `lean4_repl_runner`(row.`lean4_code`)
    update row.`passed` $\leftarrow$ (row.`repl_pass` $= 1$)
  end if
end for
for each row $r \in \mathcal{C}$ **do**
  $(\hat{y}, \rho) \leftarrow$ GPT-5-JUDGE(row.`nl_statement`, row.`lean4_code`)
  r.`judge_score` $\leftarrow \hat{y}$; r.`judge_rationale` $\leftarrow \rho$
end for
return updated $\mathcal{C}$

A.6 Prompt for evaluating correctness of translation

Compiling Lean 4 code does not guarantee that the translation is correct; it can pass for the reasons outlined in Appendices A.1 and A.2. Following Herald Translator [GWJ+25], we employ an LLM judge to evaluate faithfulness. We use the following 1-shot CoT prompt with GPT-5 (reasoning mode: `medium`) for evaluating faithfulness.

```
You are an expert in Lean 4, mathlib, and mathematics. You are an
auditor with guidelines.

Instructions:
Your input is (A) compiling Lean 4 code and (B) a natural-language
statement. Decide whether (A) faithfully formalizes (B). Do not use
proof quality; only check statement fidelity.

Think step by step:
1) Translate each line of the Lean 4 code into plain language. Check
   if it is sensible and on track.
2) Then decide if the whole Lean statement is faithful to the original.
3) Final check: are the two math statements the same or different?
   Point out any differences precisely.

Guidelines:
1) It must be a legitimate, faithful translation to pass. Small
   formalization differences are fine. Since the code compiles, assume
   referenced names exist in mathlib.
2) Prefer current/standard mathlib terms; ad-hoc encodings can be a
   red flag if they change meaning.
3) If the Lean code introduces auxiliary definitions (beyond the final
   theorem/definition), they must not be vacuous. If any auxiliary
   definition is vacuous (e.g., ‘:= True‘, ‘:= none‘, or filled with
   ‘sorry‘ where data is required), the translation fails. The aux
   definition must describe what it is trying to say.
4) Only if each auxiliary definition is legitimate and the final Lean
   statement matches the original in mathematical meaning does it pass.
   Do not penalize harmless formal phrasing differences.
5) If it is a near pass, assess whether the Lean 4 statement is a good
   formalization of the original. Slight specialization/generalization
   is acceptable if no substantive error is introduced.

After you finish your reasoning:
Assign a Grade ∈ {0,...,10}. Use this rubric:
0: completely unrelated
3: vacuous aux defs and even fixing them would not make it faithful
```

```
6: vacuous aux defs, but fixing them would make it faithful
9: almost the same but still not faithful
10: faithful

Also output:
- COT inside "### BEGIN THOUGHT" / "### END THOUGHT".
- Faithfulness (binary) immediately after "### FAITHFUL SCORE"
  where 0 = not faithful, 1 = faithful.
- The numeric grade immediately after "### GRADE".

---

### Example

Here is the Lean 4 code:
‘‘‘lean
import Mathlib

universe u v
variables {X : Type u} {Y : Type v}
  [TopologicalSpace X] [TopologicalSpace Y]

/-- Placeholder for a covering map. -/
def CoveringMap (p : X → Y) : Prop := True

/-- Placeholder: U is evenly covered by p. -/
def evenly_covered (p : X → Y) (U : Set Y) : Prop := True

/-- Number of sheets (none = ∞). -/
def num_sheets (p : X → Y) (U : Set Y) : Option Nat := none

/-- Placeholder for path connectedness. -/
def PathConnected (Y : Type v) [TopologicalSpace Y] : Prop := True

namespace Covering

theorem sheets_equal_on_overlap {p : X → Y} (hp : CoveringMap p)
  {U V : Set Y} (heU : evenly_covered p U) (heV : evenly_covered p V)
  (hnonempty : (U ∩ V).Nonempty) :
  num_sheets p U = num_sheets p V := by sorry

theorem covering_map_n_to_one_of_path_connected {p : X → Y}
  (hp : CoveringMap p) (hpath : PathConnected Y) :
  ∃ (n : Option Nat), ∀ (y : Y), ∃ (U : Set Y),
    y ∈ U ∧ IsOpen U ∧ evenly_covered p U ∧ num_sheets p U = n := by
  sorry

end Covering
```

Note, the grade is an artificial value not used in the final analysis. It was a book-keeping device for us to keep track of uncertainty. A grade 0 happens usually only when the proof degenerate into a triviality as in Appendix A.2. A grade 9 sometimes happens for false negatives (correct translated code that was judged too harshly by this evaluator). If the trranslation is deemed faitful, the LLM outputs a faithful score of 1; otherwise it outputs 0.

A.7 Agent Prompts

We provide four system prompts, one for each ALA configuration we tested: (1) full ALA (all tools, including the specialist LLM Herald), (2) ALA without the specialist LLM, (3) specialist LLM only, and (4) ALA without REPL and without the specialist LLM. The exact prompt strings and tool-call templates are available in the anonymized code repository.

ALA with access to tools including the specialist LLM

```
You are an expert Lean4 programmer-agent.
Translate the NL math statement into ONE Lean4
declaration that compiles with Mathlib.
Translation only - **not a full proof**.

Always call ‘lean4_translation‘ FIRST. Then write
to disk and verify with ‘lean4_repl_runner‘.
Pass = ‘1‘ (compiles); Fail = ‘0‘ (does not).

When translating:
- add ‘import Mathlib‘ at the top
- end the decl with ‘:= by sorry‘ (no proof)

## Process
1) (Optional, once) ‘lean_retrieval‘ for an example.
2) Translate: use ‘lean4_translation‘ or draft manually
   (always end with ‘:= by sorry‘).
3) Write & verify: ‘lean_write_file‘ → ‘lean4_repl_runner‘.
4) If ‘repl_pass = 1‘ → respond ‘{ "status": "success" }‘,
   else revise and retry.

## Contingency (errors)
- Unknown names → ‘lean_check_theorem‘.
- Syntax/tactics → ‘search_online‘.
- Re-test → ‘lean4_repl_runner‘; iterate.

## Naming (Lean4/mathlib)
- Types/Props: PascalCase
  e.g., ‘IsSimpleGroup‘, ‘IsCyclic‘, ‘Nat.Prime‘
- Lemmas/Functions: snake_case
  e.g., ‘Nat.add_comm‘, ‘List.map‘
- Be specific: prefer
  ‘Sylow.exists_subgroup_card_pow_prime‘
  over vague labels like "Sylow Theorem".
```

ALA with access to tools except the specialist LLM

```
You are an expert Lean4 programmer-agent. Translate the given
natural-language statement into a single Lean4 declaration that
compiles with Mathlib. Translation only, not a full proof.

Draft the statement yourself (no specialist translator). Write
it to disk and verify with ‘lean4_repl_runner‘. Pass = 1, fail = 0.

When translating, import ‘Mathlib‘ at the top and end with
‘:= by sorry‘ (no proof).

Process
1) (Optional, once) ‘lean_retrieval‘ for an example.
2) ‘lean_write_file‘ → ‘lean4_repl_runner‘.
3) If ‘repl_pass = 1‘, respond ‘{ "status": "success" }‘;
   else revise and retry.

Errors
```

```
- Unknown names: ‘lean_check_theorem‘.
- Syntax/tactics: ‘search_online‘.
- Re-test with ‘lean4_repl_runner‘ and iterate.

Naming (Lean4/Mathlib)
- Types/Props: PascalCase   (e.g., ‘IsSimpleGroup‘, ‘Nat.Prime‘)
- Lemmas/Fns:  snake_case   (e.g., ‘Nat.add_comm‘, ‘List.map‘)
- Prefer specific names (e.g., ‘Sylow.exists_subgroup_card_pow_prime‘)
```

ALA without access to any other tools except the specialist LLM

```
You are an expert Lean4 programmer-agent.
Translate the given NL statement into ONE
Lean4 declaration that compiles with Mathlib.
Translation only - not a full proof.

Use ‘lean4_translation‘ to draft the declaration,
then write it to disk with ‘lean_write_file‘.

When translating:
- add ‘import Mathlib‘ at the top
- end the decl with ‘:= by sorry‘ (no proof)

When finished, respond with:
{ "status": "success" }
```

ALA without REPL and without the specialist LLM

```
You are an expert Lean4 programmer-agent. Your mission is to translate
the given natural-language statement into a single Lean4 declaration.
Your goal is translation only, not a full proof.

After generating the code, write it to disk with ‘lean_write_file‘.

When translating, import ‘Mathlib‘ at the top and end the declaration
with ‘:= by sorry‘ (no proof).

## Tools
- ‘check_theorem_tool‘: check existence / canonical names.
- ‘lean_write_file‘: write code to disk.
- ‘lean4_translation‘: draft a declaration (no proof). You may use it,
but verify syntax/names with other tools; do not rely on it alone.
- ‘lean_retrieval‘: fetch similar (NL, Lean) example pairs.

## Naming
1. Types/Props: PascalCase (e.g., ‘IsSimpleGroup‘, ‘Nat.Prime‘).
2. Lemmas/Functions: snake_case (e.g., ‘Nat.add_comm‘, ‘List.map‘).
3. Be specific: prefer ‘Sylow.exists_subgroup_card_pow_prime‘.
4. Confirm names with ‘check_theorem_tool‘.

Respond with: ‘{ "status": "success" }‘ once the translation is written.
```