LeanTutor: A Lean-Verified Tutor for Mathematical Proofs

Manooshree Patel [1] **Rayna Bhattacharyya** [*1] **Thomas Lu** [*1] **Arnav Mehta** [*1] **Niels Voss** [*1]
Narges Norouzi [1] **Gireeja Ranade** [1]

Abstract

This paper presents a proof-of-concept version for LeanTutor, a tutoring system for mathematical proofs that combines Large Language Models (LLMs) with the Lean proof assistant. LeanTutor interacts with the student in natural language, formally verifies auto-formalized-versions of the student-written proofs in Lean, generates correct next steps, and provides the appropriate instructional guidance. LeanTutor is composed of three modules: (i) an autoformalizer/proof-checker, (ii) a next-step generator, and (iii) a natural language feedback generator. To evaluate the system, we introduce PeanoBench, a dataset of 371 Peano Arithmetic proofs in human-written natural language and formal language, derived from the Natural Numbers Game. Each natural language proof step is paired with the corresponding logically equivalent tactic in Lean. The autoformalizer correctly formalizes 57% of tactics in correct proofs and accurately identifies the incorrect step in 30% of incorrect proofs. In generating natural language hints for erroneous proofs, LeanTutor outperforms a simple baseline on accuracy and relevance metrics.

1. Introduction

College students use LLMs such as ChatGPT and Claude to start projects, create practice questions, and generate solutions to academic assignments (OpenAI, 2025; Anthropic, 2025). State-of-the-art LLMs are easy to access and perform well on material from undergraduate courses (Scarfe et al., 2024). However, LLM usage can be detrimental to student learning (Goetze, 2025), because these systems are not designed from a pedagogical perspective. Specifically, (1) most models are designed to be maximally "helpful" (Askell et al., 2021) to a user, which often leads to them directly giving away the answer to a student,instead of helping them come up with it on their own (Sonkar et al., 2024), (2) even state-of-the-art models are prone to hallucinations and generate convincing wrong answers (Maurya et al., 2024; Balunović et al., 2025; Gupta et al., 2025), (3) models struggle to identify mistakes in reasoning (Tyen et al., 2024; Miller & DiCerbo, 2024), and (4) even if models can produce the correct answer, they cannot necessarily produce correct reasoning to guide the student (Gupta et al., 2025). Even more alarmingly, students have admitted that LLM usage on educational assignments has led them to feeling that they are "getting dumber" (Goetze, 2025).

However, educational technology can have an immense positive impact when used appropriately. For instance, autograders have revolutionized the student experience in introductory programming classes (DeNero & Martinis, 2014; Mitra, 2023; Hecht et al., 2023; Messer et al., 2024). Autograders, along with feedback from a programming language compiler, encourage self-correction and allow students to rapidly test solutions and learn from their mistakes, empowering them to explore new ideas through private, low-stakes failure (Aziz et al., 2015).

Mathematical proofs have long been a "stumbling block" for undergraduates (Iannone & Thoma, 2024), and for decades, math educators have been trying to build an autograder and/or tutor for math proofs (Bundy et al., 2000; Lodder et al., 2021; Barnes & Stamper, 2008; Park & Manley, 2024; Sufrin & Bornat, 1997; Zhao et al., 2024a; Sieg, 2007; Wemmenhove et al., 2022). Educators have developed intelligent tutoring systems (ITS) (Lodder et al., 2021; Bundy et al., 2000) to teach math proofs or utilized theorem provers as teaching tools (Avigad, 2019; Wemmenhove et al., 2022). While these systems provide students with similar benefits to autograders (such as immediate feedback), they can be tedious to create (Dermeval et al., 2018) or require an understanding of complex formal language syntax that students find difficult to learn (Thoma & Iannone, 2022).

LLMs, theorem provers, and ITS all have unique complementary strengths, and we aim to develop a proof tutoring system that leverages all of them. We propose **LeanTutor, a Lean-verified tutoring system for undergraduate mathe-**

[*]Equal contribution [1]University of California, Berkeley, United States. Correspondence to: Manooshree Patel <manooshreepatel@berkeley.edu>, Gireeja Ranade <ranade@eecs.berkeley.edu>.

The second AI for MATH Workshop at the 42nd International Conference on Machine Learning, Vancouver, Canada.

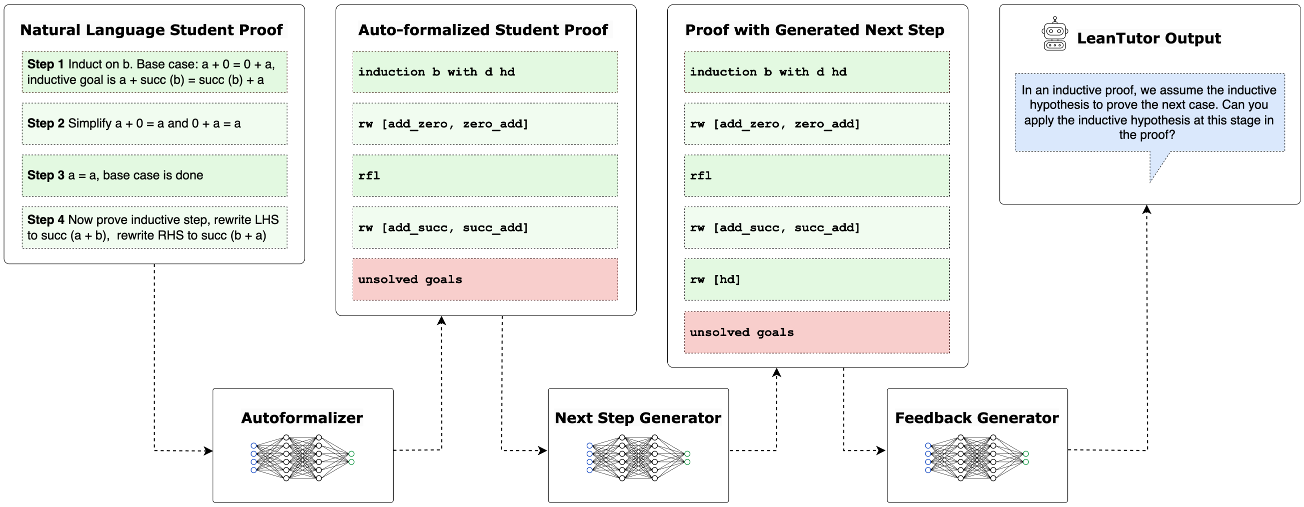

Figure 1. LeanTutor is comprised of three modules: an autoformalizer that automatically formalizes an NL student proof into Lean step-by-step; a next step generator that generates a next feasible tactic for the student proof; and a natural language feedback generator that generates guiding feedback to help the student progress towards a correct proof.

matics proofs. LeanTutor interacts with students in natural language (NL), while using the formal language (FL) Lean to evaluate proof correctness and generate correct next steps on the backend. Specifically, LeanTutor can:

- Accept complete/partial/correct/incorrect student-written natural language proofs
- Verify if the student work is correct or incorrect
- Identify the student error, if applicable, and provide guidance towards a correct proof, without giving away the complete answer

The LeanTutor design assumes a small self-contained dataset, as used by Murphy et al. (2024); Cunningham et al. (2023) and one known proof per theorem with a Lean formalization. These limit generalizability, but are reasonable assumptions for the tutoring setting (solutions are readily available, and there is a limited set of previous theorems that students will use). We attempt *faithful autoformalization*, (autoformalization focusing on preserving the semantic meaning of the natural language (Murphy et al., 2024)) of natural language statements, when one complete proof of the theorem is known in natural and formal language. We similarly explore next-step-generation where the explorable space of theorems is small (relative to Mathlib which has an extremely large theorem space (mathlib Community, 2020)). Additionally, the educational application of LeanTutor introduces the following novel challenges in AI for Math:

- We must be able to formalize, not only complete and correct, but incomplete and incorrect proofs into Lean. Previous work on autoformalization focuses on whole proof and theorem statement autoformalization of correct proofs and statements (Yang et al., 2024).
- In our paradigm, at least one correct proof (and a semantically equivalent Lean formalization) for all theorems is known. Hence, the key challenge is not to prove a new theorem or formalize mathematics, but to identify which proof approach a student is taking and pinpoint error locations in the proof.

Contributions We present three main contributions in this work. First, we propose a framework and implementation for LeanTutor (Fig. 1) comprised of three modules: (1) an autoformalizer and proof checker, (2) a next-step generator, and (3) a natural language feedback generator. Second, we introduce the new problem of autoformalization of correct and incorrect proofs in the presence of a reference proof. We propose an autoformalizer that proceeds tactic-by-tactic, and propose a metric to evaluate faithful autoformalization in the presence of a reference proof. Third, we construct the PeanoBench dataset, comprising of 371 correct and incorrect Lean proofs, with rule-based human-written NL annotations. We evaluate LeanTutor's ability to autoformalize PeanoBench proofs and generate feedback for a subset of incorrect and incomplete proofs.

2. Related Work

2.1. Autoformalization via Language Models

A large body of recent work has focused on autoformalizing, translating NL theorem statements into formal math languages, using deep learning methods (Gadgil et al., 2022; Ying et al., 2024a; Gao et al., 2024; Shao et al., 2024; Wu et al., 2022; Jiang et al., 2022a; Azerbayev et al., 2023a; Lin et al., 2025; Zhou et al., 2024; Lin et al., 2025). The more difficult task of autoformalizing whole proofs from NL to FL has been explored in fewer works (Jiang et al., 2022b; Murphy et al., 2024; Wang et al., 2024b; Cao et al., 2025; Tarrach et al., 2024; Huang et al., 2024). State-of-the-art (SOTA) LLMs, without any specific formal language training, have shown strong performance on the task of autoformalization (Wu et al., 2022; Chen et al., 2021) and motivate our development of an LLM-agnostic framework for autoformalization.

In a classroom setting, autoformalization can support tutoring (as in LeanTutor) or auto-grading. Both applications require *faithful autoformalization* Murphy et al. (2024). We take a similar approach to Kulal et al. (2019) method of translating pseudocode to code, line-by-line, in a C++ program generation task. Faithful autoformalization metrics are discussed in Section 5.1. We make the reasonable assumptions for the classroom setting that all proofs come from a small dataset and at least one valid proof per theorem is known (in both NL and FL). Murphy et al. (2024); Cunningham et al. (2023) successfully formalize proofs in a small dataset where all feasible theorems/tactics are known.

2.2. Neural Theorem Proving

Neural theorem proving reframes theorem proving as a language modeling task (Li et al., 2024a). An abundance of prior work has made progress towards training language models for theorem proving (Jiang et al., 2022a; Wu et al., 2024; Polu & Sutskever, 2020; Polu et al., 2022; Jiang et al., 2021; Yeh et al., 2023; Wang et al., 2023b; Gloeckle et al., 2023; Wang et al., 2024a; Szegedy et al., 2021; Welleck et al., 2022; Ying et al., 2024b; Azerbayev et al., 2023b; Thakur et al., 2025; Poesia et al., 2024; Ren et al., 2025; Lin et al., 2025; Yang et al., 2023). Additionally, prior work has explored theorem proving frameworks with SOTA LLMs (Jiang et al., 2022b; Zhao et al., 2024b; Zheng et al., 2023; Wang et al., 2023a; Huang et al., 2024; Thakur et al., 2023; DeepMind, 2024; Trinh et al., 2024). Our next-step generation approach is largely inspired by the COPRA agent (Thakur et al., 2023). The COPRA agent performs a GPT-4 directed depth-first search over sequences of possible tactics, to complete a formal theorem proof. The agent additionally implements a "progress check", which assesses if generated tactics progress the proof.

2.3. Automated Feedback Generation for Programming Assignments

We draw inspiration for LeanTutor's feedback generation module from automated feedback generation in programming classes (Suraweera & Mitrovic, 2002; D'antoni et al., 2015; Singh et al., 2013; Suzuki et al., 2017; Head et al., 2017; Alur et al., 2013). Since students write their code in a programming environment where compilers enforce formal correctness and autograders ensure that the code passes test cases or return appropriate errors, autonomous tutors can leverage the resulting error messages and metadata to generate high-quality feedback. We build on the five hint types identified by Suzuki et al. (2017) (transformation, location, data, behavior, and example) that can be generated via program synthesis to provide students feedback in an introductory coding class.

Autoinformalization, translating formal statements into informal ones (Li et al., 2024a), is a parallel task to feedback generation. LLM-based autoinformalization has been explored with success (Wu et al., 2022; Jiang et al., 2023; Huang et al., 2024; Azerbayev et al., 2023a; Lu et al., 2024a).

2.4. Math Proof Tutors

We identify three categories of existing math proof tutors—intelligent tutoring systems, LLM-based tutors, and theorem prover-based tutors. Researchers have made attempts to develop (Autexier et al., 2012; Briggle et al., 2008) or developed intelligent tutoring systems (ITS) for math proofs (Barnes & Stamper, 2008; Lodder et al., 2021; Bundy et al., 2000). ITS require expert authoring of solutions or feedback, making them difficult to develop and scale (Dermeval et al., 2018). LLM-based math tutors have demonstrated benefits such as learning gains (Pardos & Bhandari, 2023) and can maintain conversations with no harmful content (Levonian et al., 2025). However, these LLMs fail as tutors, for the reasons outlined in Section 1. Math educators have used theorem provers, such as Lean, Coq (Huet et al., 1997), and Isabelle (Paulson, 1994), to teach proofs (Avigad, 2019; Villadsen & Jacobsen, 2021; Boldo et al., 2024; Kerjean et al., 2024). These tools have led to unique benefits in students' learning of proofs (Thoma & Iannone, 2022), but students struggle to learn the complex syntax required to interact with most (Avigad, 2019; Buzzard, 2022; Villadsen & Jacobsen, 2021; Karsten et al., 2023).

A more extensive review of these three categories of tutors can be found in Appendix A.1.

3. PeanoBench Dataset

To develop and evaluate LeanTutor, we created the PeanoBench dataset, which contains 371 total proofs. Each proof has a human-written natural language proof and a semantically equivalent formal language proof in Lean. PeanoBench is derived from the original 80 Peano Arithmetic proofs in the Natural Number Game 4 (NNG4) (Buzzard et al., 2023) (Apache-2.0 license). NNG4 organizes proofs into "worlds", or topic categories, such as "Addition World", "Multiplication World", and so on. Worlds generally increase in difficulty. In PeanoBench, we keep proofs organized by the original NNG4 world designations.

Unlike other datasets with NL and FL proofs (Lu et al., 2024b; Wang et al., 2024b), PeanoBench's informalizations are human-written[1].

To construct the dataset, we begin with a subset of 75 of the original NNG4 proofs (we remove the attempted proof of Fermat's Last Theorem and proofs which contain the `simp` tactic). A categorization of selected proofs by world can be found in Appendix A.3. We annotate these 75 proofs tactic-by-tactic, such that each Lean tactic has a corresponding semantically equivalent NL back-translation (example proofs in Figure 3 and Figure 4 in the Appendix). The **one-to-one correspondence between NL proof steps and individual FL tactics** differentiates PeanoBench from prior datasets for Lean autoformalization that pair whole Lean proofs with their whole NL counterpart (Lu et al., 2024b; Wang et al., 2024b; Gao et al., 2024).

Proof annotators followed two rules while annotating. (1) Natural language annotations are free of Lean-specific syntax, premises, or tactics. (2) Natural language annotations are written to function as standalone proofs independent of the Lean code.

PeanoBench is comprised of three groups of proofs. The first set of 75 proofs, derived directly from NNG4, is annotated by two paper authors and annotations are very descriptive. We call this first set of proofs our *staff solutions*. To mimic student proofs, we write two variations of each *staff solution* proof, to create the second group of proofs. When possible, we varied the proof's Lean code (whether this be a major logical difference or rearranging commutative tactics). We then annotated the proof in either the (1) *equation-based* persona or the (2) *justification-based* persona (we borrow the idea of persona-based annotations from user interface design (Cooper, 1999)). Each proof was annotated by one of five annotators and proofread by a different annotator. In total, we end with 75 *staff solution* proofs, 75 *equation-based* proofs, and 75 *justification-based* proofs. An example of one theorem with three proofs in three personas can be found in Figure 3.

Incorrect proofs, the third group of proofs, are derived from the set of *equation-based* and *justification-based* proofs. We mimic "incorrectness" by randomly skipping a step from the last three lines of the proof (step-skipping algorithm pseudocode is in Algorithm 1). Proofs that are only one line are removed from the incorrect set. The "incorrect" step in the proof is then marked; this is the step that causes the first Lean compiler error in the proof. In total, we end with 73 incorrect *equation-based* proofs and 73 incorrect *justification-based* proofs.

Staff solutions proofs are only offered as context to the model. System performance is evaluated on the correct and incorrect *equation-based* and *justification-based* proofs.

4. System Design

LeanTutor has three modules: an autoformalizer, a next-step generator, and an automatic feedback generator, which are illustrated in Fig. 1.

4.1. Autoformalizer and Proof Checker

The tutoring application offers a new frame for approaching autoformalization of NL proofs into Lean, where the autoformalizer will have access to a reference/staff solution. Since we anticipate implementing this tutor in an undergraduate classroom, all theorems have at least one correct proof (staff solutions for assignments). Furthermore, the space of all feasible definitions and theorems is known and relatively small. However, the tutoring setting offers novel challenges as compared to other autoformalization settings. Specifically, tutoring requires *faithful autoformalization*, i.e. one that retains the meaning of the student's input. Even mistakes must be correctly reflected in the formalization, since student-written proofs may be both incomplete and incorrect. Finally, student NL will have lots of variation and may not have the polish of professional mathematical writing.

Practically, we want both the semantic meaning and granularity of the student's proof step to be reflected by the generated Lean code. We note that the student written NL proofs may or may not follow the proof path taken by the staff solution. We include examples from PeanoBench of faithful formalization of the desired style in Appendix A.2.

We autoformalize student proofs one step at a time, and check for compliation at each step. This approach is similar in spirit to previous works breaking autoformalization into subtasks (Patel et al., 2023; Jiang et al., 2022b), but closest to the work of Kulal et al. (2019). Figure 2 illustrates this process of translating a single student proof step into Lean, and repeating the process until the student is finished with their proof or the student makes an error in their proof. To support the autoformalization task, we add several key pieces of information in-context of our model:

- *Staff Solution*: We provide in context one correct proof

[1]To support the laborious task of human-written informalizations, we build upon the tooling released by Welleck & Saha (2023) and develop a `suggest` tactic, which displays an LLM-generated NL informalization of the selected Lean tactic in the Lean Infoview. Human annotators then appropriately edited these informalizations.

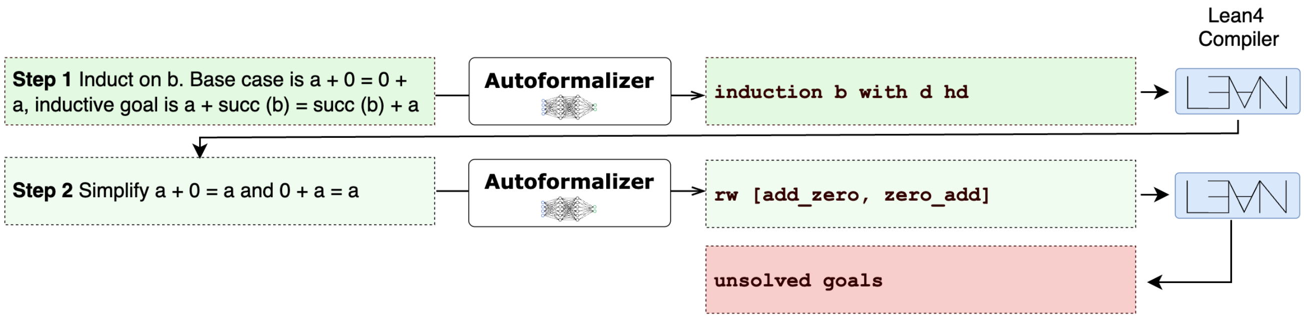

Figure 2. Autoformalizer architecture. The natural langauge student step is provided to the autoformalizer, and the output is checked by the Lean compiler. The formalization of each step is appended to formalizations of previous steps to check for correctness.

in both natural language and formal language (valid assumptions in the tutoring paradigm). The input student proof may or may not align with the staff solution. We do not leverage this staff solution beyond providing it in the context, but aim to do so in future work.

- *Theorem and Tactic Dictionary*: We organize all of the tactics and theorems in our dataset into a dictionary, where the keys are the formal Lean names of the theorems and tactics, and the values are natural language descriptions of each. All tactics and theorems are equivalent (specifically a subset, as we remove a few tactics such as `simp`) to those originally defined in NNG4[2]; we do not introduce new theorems or tactics. All definitions for these theorems and tactics are written by paper authors, based on the instructional content in NNG4.

- *5-shot examples*: We include five examples of translations of a natural language proof step and corresponding Lean formalization following Murphy et al. (2024). These five examples were selected from our existing dataset.

Proof Checker. The input to autoformalizer module will include both correct and incorrect proofs.

As shown in Figure 2, each autoformalizing student proof step is appended to the Lean theorem statement and previously formalized steps. The proof is compiled, via Lean-Interact (Poiroux et al., 2025). If the compiler output indicates only `unsolved goals`, we assume the student step is correct and proceed with autoformalizing remaining steps. For any other error message (`unknown tactic`, `error:unexpected identifier`, etc.), we assume the student step is incorrect and mark this proof step as erroneous. (Note: We end the autoformalization process once the first error is located.) Furthermore, a compiler error can indicate either an incorrect student proof step, or an autoformalization error. This is a limitation of the system that we intend to address in future work.

4.2. Next Step Generator

The Next Step Generator (NSG) (See Appendix A.5) is launched when the student proof is not identified as complete and correct by the autoformalizer/proof checker. The NSG takes as input the formalized partial student proof (with the incorrect step removed). It aims to output a Lean tactic that can lead to a complete proof. Similar to (Thakur et al., 2023), the module performs an LLM-directed depth-first proof search. An LLM is instructed to generate 12 candidate tactics with a rank-ordering of their likelihood of being a correct next step. The prompt includes a list of all tactics/premises used in the NNG4 world of that theorem.

The 12 generated tactic candidates are appended to the existing proof and run through the Lean compiler (via Lean-Interact (Poiroux et al., 2025)). Compiling tactics are then filtered through a *progress check*, which follows Thakur et al. (2023) and Sanchez-Stern et al. (2020). In the progress check, we (1) ensure we are not using any theorems on a list of forbidden theorems (we define this list to include the theorem we are currently trying to prove and theorems that are introduced after the theorem being proven in the order defined by NNG4) and (2) avoid cyclic tactics that would cause the proof-tree to revisit a goal state (Thakur et al., 2023). We build a proof-search tree using all tactics that fulfill the compilation and progress check and do a depth-first search until a complete proof is found. We bound the tree depth to eight, which is sufficient for most of the proofs in our case. If a proof cannot be found, we report to the following module that the NSG could not find an appropriate next tactic.

[2]For pedagogical purposes, tactics behave slightly differently in NNG4 compared to Mathlib. Operations on the natural numbers are defined axiomatically rather than recursively. We preserve these changes in PeanoBench.

4.3. Natural Language Feedback Generator

The feedback-generation module combines information from previous modules to provide natural language feedback to the student. Specifically, the feedback generator takes as input the student's autoformalized proof, the Lean compiler error message (if present), and the next Lean tactic generated from the NSG module. To aid in error identification, we include six common errors students have made in inductive proofs (Baker, 1996) in our prompt (prompt found in Appendix A.10.2).

We use this information to automatically generate three types of feedback common in ITS (VanLehn, 2006). Similar to the automatic feedback generated by D'antoni et al. (2015), we (1) identify the student error and (2) generate a hint or question that guides the student to the next step. We also generate (3) an explicit next step the student could take, following the paradigm of *bottom-out hints* (Suzuki et al., 2017). This third part of our feedback is very similar to the autoinformalization task in automated theorem proving (Li et al., 2024a).

5. Experiments

We evaluate the end-to-end LeanTutor system on incorrect proofs. In this experiment, a baseline model and LeanTutor are both given incorrect proofs as input and generate NL feedback as output. Human evaluators then assess the generated feedback across four axes: Accuracy, Relevance, Readability, and Answer Leakage, on a 5-point scale. These experiments are detailed in section 5.4. To understand the impact of key innovations in our autoformalizer, namely the presence of staff solutions and the step-by-step autoformalization approach, we perform ablations on our Autoformalizer. To assess our model's performance at the faithful autoformalization task, we present a novel metric. These experiments are explained in section 5.2. All experiments cost less than \$4.00 to run on `gpt-4o-mini-2024-07-18`. We expect that the autoformalization performance can be boosted by using other more powerful LLMs. Since the focus of our paper is presenting a framework for the LeanTutor model, we did not optimize over different LLMs.

5.1. Metric for Faithful Autoformalization

A few metrics have been developed to assess faithful autoformalization (Murphy et al., 2024; Liu et al., 2025; Lin et al., 2025; Li et al., 2024b). Li et al. (2024b) verify Isabelle formalizations and rely on Sledgehammer, Lin et al. (2025) use an LLM-as-a-Judge, Murphy et al. (2024) use an SMT solver to prove equivalence between two statements, and Liu et al. (2025) define a new equivalence relation: bidirectional extended definitional equivalence (BEq). We prefer not to use the LLM-as-a-judge paradigm (Lin et al., 2025) due to the potential for hallucinations. Both the measures proposed by Murphy et al. (2024); Liu et al. (2025) are too coarse for our use case.

We develop a metric that performs *relaxed exact matching*. Our metric has two phases. Firstly, exact *tactic-matching* is attempted in which the generated tactic string is matched with the ground truth tactic string, similar to the variable transformations implemented by Jain et al. (2022) for program synthesis. If string matching fails, we move to the second phase: *state-matching*. In *state-matching* we compare the two tactics by checking if the proof states (the proof state rendered once the predicted and ground truth tactics have been appended to the existing predicted and ground truth proofs respectively) are syntactically identical up to variable naming. We call our metric *relaxed*, because we accommodate differing variable names between the input and ground truth proofs. To do this, proof states are segmented by goal and/or casework and we locate all variables through a custom Python implementation of Lean Identifiers (Lean Community, 2024). Variables in all goal state segments are standardized and string matching can ensue. If this check fails as well, we deem the predicted tactic as not a faithful autoformalization of the input NL proof stem. More details on metric implementation and pseudocode can be found in Appendix A.7.

5.2. Autoformalizer Evaluation

For our *baseline model*, we adapt the autoformalization prompt proposed by Murphy et al. (2024) to our dataset. Their autoformalization prompt was designed for a small dataset use case in which all tactics/premises can be provided in-context; this is appropriate for PeanoBench. Our baseline prompt contains the theorem statement in both NL and FL, the tactic and theorem dictionaries, five examples of the formalization task, and the student input that needs to be formalized.

For correct proof formalizations, accuracies at both the tactic and proof levels were measured. Tactic-level accuracies were determined using the metric described above. Proof-level accuracy was measured by verifying all tactics in a given proof were correctly autoformalized. For incorrect proof formalizations, we report only proof-level accuracy. Thus, for incorrect proofs a formalization is considered successful if (1) all correct proof steps until the first incorrect step were formalized correctly and (2) formalization of the incorrect proof step leads to a Lean compiler error.

We report results in Table 1. Tactic-level results are out of 900 total tactics, correct proofs results are out of 150 total proofs, and incorrect proof results are out of 146 proofs. The Baseline + Staff Solution model displays superior performance in all categories compared to the Baseline model.

Table 1. Autoformalization performance per experiment across correct and incorrect proofs. Autoformalization is done step-by-step in the Baseline and Baseline + Staff Solution experiments. In the experiments labeled with (whole proof), the whole NL proof is autoformalized into Lean code at once. Binomial error bars were computed using Jeffreys prior with a 95% confidence interval.

Experiment	Correct Tactics	Correct Proofs	Incorrect Proofs
Baseline	$32.9\% \pm 3.1\%$	$6.7\% \pm 4.0\%$	$14.4\% \pm 5.7\%$
Baseline + Staff Solution	$\mathbf{56.8\% \pm 3.2\%}$	$18.0\% \pm 6.1\%$	$\mathbf{30.1\% \pm 7.4\%}$
Baseline (whole proof)	$28.2\% \pm 2.9\%$	$10.7\% \pm 4.9\%$	$13.0\% \pm 5.4\%$
Baseline + Staff Solution (whole proof)	$51.8\% \pm 3.3\%$	$\mathbf{26.7\% \pm 7.0\%}$	$21.9\% \pm 6.7\%$

We find that the model relies heavily on the staff solution in its context in generating the formalization. In 89% of correct tactic formalizations, the tactic predicted was in the staff solution (meaning that the tactic was present somewhere in the staff solution provided in context, but possibly in a different position than the predicted tactic). When the autoformalization was wrong, the model copied a tactic in the staff-solution, instead of predicting the real tactic corresponding to the NL input, in 51% of cases. When the expected formalization was not in the staff solution, LeanTutor was able to correctly formalize the NL proof step in 32% of cases. These results demonstrate a limitation of the current approach's ability to autoformalize NL proofs whose formalization does not correspond to the formalized staff solution provided in-context.

We compare our autoformalizer model to one ablation: generating whole proofs all at once instead of step-by-step generations (experiments labeled with (whole proof) in Table 1). With this approach, the autoformalized whole proof does not necessarily contain the same number of tactics as our ground truth whole proof. We truncate proof lengths to `min(len(generated proof), len(ground truth proof))` (the length of a proof referring to the number of tactics in the proof) and align both proofs to each other tactic-by-tactic. We compute tactic-level and proof-level accuracy in the same manner described above[3]. Considering the models with staff solutions, the step-by-step autoformalization approach has comparable performance to the whole proof autoformalization on correct proofs. However, the step-by-step autoformalization outperforms the whole proof approach on incorrect proofs, by 8%. As many incoming proofs to a tutoring system will be incorrect, better performance on incorrect proofs vs. correct proofs is advantageous.

[3]Our metric is imperfect for evaluating generated whole proofs. Thus, we also evaluate how many generated whole proofs (in the correct proof experiments) also completed successfully, with the Lean compiler displaying `no goals`. The Baseline (whole proof) model produced 28 compiling proofs and the Baseline + Staff Solution model (whole proof) produced 50 compiling proofs. Note, that a complete Lean proof doesn't serve as an appropriate measure for faithful autoformalization.

Prompts for step-by-step and whole proof generation can be found in Appendix A.10.1. We performed additional experiments, evaluating the impact of adding the student's natural language proof and Lean goal state information in-context of the autoformalizer. These results can be found in Appendix A.6.

5.3. Metric for LeanTutor Feedback

In the system-level evaluation of LeanTutor, a student NL proof is input and NL feedback is generated as output. We qualitatively evaluate the generated outputs on four axes: *Accuracy*, *Relevance*, *Readability*, and *Answer Leakage*, motivated by the metrics used in Mitra et al. (2024); Mozafari et al. (2025); Phung et al. (2024). We evaluate each of our three categories of feedback (error identification, hint/question generation and explicit next step) along each axis using a 5-point scale.

We define what it means to receive the highest rating of 5 for each axis below. A score of 1 indicates complete disagreement with the following definitions.

- **Accuracy**: The generated error/hint/next-step is correctly and accurately identified (similar to Factuality axis of Mitra et al. (2024) and *HCorrect* of Phung et al. (2024).)
- **Relevance**: The generated error/hint/next step is relevant to the error/proof following Mitra et al. (2024); Mozafari et al. (2025).
- **Readability**: The generated feedback is coherent (Mitra et al., 2024; Phung et al., 2024).
- **Answer Leakage**: The generated feedback does not disclose the answer in any way (Mozafari et al., 2025; Phung et al., 2024).

5.4. LeanTutor Evaluation

We evaluate our full system on incorrect proofs and "cold-start" proofs, a proof in which the student does not know how to start the proof. Results for the "cold-start" proofs can be found in Appendix A.9. Across our experiments, we

Feedback Type	Accuracy	Relevance	Readability	Answer Leakage
Baseline Error Identification	2.6	2.7	**4.8**	4.7
LeanTutor Error Identification	**3.7**	**3.6**	4.7	**4.9**
Baseline Hint/Question	2.9	2.8	**4.8**	**4.6**
LeanTutor Hint/Question	**4.0**	**4.1**	4.5	4.4
Baseline Next Step	2.8	2.8	4.6	**1.6**
LeanTutor Next Step	**3.9**	**3.9**	**4.7**	1.1

Table 2. Average (across all proofs) scores of generated feedback from baseline and LeanTutor experiments on 21 incorrect proofs. Only proofs that were correctly autoformalized were selected for this evaluation. The generated feedback (error identification, hint/question, next step) was all scored on four qualitative axes on a scale of 1-5 in which a score closer to 5 indicates desired performance.

use `gpt-4o-mini-2024-07-18` (temperature = 0.0). Feedback evaluation was conducted by three of the paper authors, all of whom are undergraduate students with prior teaching experience. All evaluators discussed and came to an agreement on the scores for several proofs. After jointly calibrating scores on several proofs, output for proof was evaluated by a single author.

Incorrect Proofs We evaluate our end-to-end system on a subset of incorrect proofs from PeanoBench. We only consider incorrect proofs that were "successfully autoformalized" by the LeanTutor autoformalizer. Of the 44 proofs (results in Table 1), we randomly selected one to three proofs per world, totaling 21 proofs. We exclude proofs for evaluation which did not contain a Lean compiler error, but were simply incomplete proofs. These proofs are passed through our Next Step Generator and Feedback Generator modules. All three types of generated feedback are evaluated by paper authors. We compare to a simple baseline, providing the LLM with the erroneous student proof and prompting the model to generate the three feedback types. An example of LeanTutor's generated hints (and their evaluations) for one incorrect proof can be found in Appendix A.8. The prompts for LeanTutor's feedback generation module and the baseline model can be found in Appendices A.10.2 and A.10.3 respectively.

Our system-level evaluation (Table 2) indicates LeanTutor outperforms the baseline model on the *Accuracy* and *Relevance* metrics. Performance on the *Readability* and *Answer Leakage* metrics are comparable for both models. (Note: We expect high answer leakage in the scores for "next step" feedback; a score of 1 is expected).

6. Limitations

LeanTutor presents a proof-of-concept design for a formally-verified mathematical proof tutoring system. Our work is a first prototype for such a system and leaves many open questions for future research. Our autoformalization and proof-search strategies are focused on a small dataset, which is acceptable for our use case, but as a result, some aspects of our approach do not easily generalize.

First, we are limited because of two major assumptions in our system design. The first is assuming a one-to-one correspondence between NL proof steps and FL tactics, which does not scale to more complicated proofs as the granularity of informal and formal mathematics is generally quite different. The second assumption is the presence of an already formalized staff solution, which could be a significant burden on an instructor in the absence of a good autoformalizer. Similarly, our metric for faithful autoformalization applies only when ground truth formalizations exist. We aim to explore approaches that only use an informal staff solution in future work.

A second set of limitations comes from our dataset construction. (1) While students commonly miss steps in writing proofs, there are several other types of errors that are not captured in the incorrect proofs dataset. (2) All of the natural language in PeanoBench has been written by paper authors, as opposed to non-author students (the varied personas are an earnest effort to incorporate realistic natural language variations).

Finally, a critical limitation in our system design is that if autoformalization fails, we cannot proceed in responding to the student. As a result, in our evaluation, we did not evaluate end-to-end system performance on proofs with incorrect autoformalization. Relatedly, we assume a student proof is incorrect if the Lean compiler errors. However, errors may also result from incorrect autoformalization, which could lead to false positives (though spot checking revealed this was not a big issue).

7. Conclusions and Future Work

Our hope is that LeanTutor's approach of combining state-of-the-art LLMs with the Lean theorem prover supports students' self-learning of math proofs. Our aim is to eventually deploy LeanTutor in large undergraduate mathematics classes such as discrete math and linear algebra. However, all LeanTutor modules require much improvement before we can realize this goal. In particular, there is significant room to more effectively use both the informal and formal versions of the *staff-solution* proofs in the Autoformalizer and NSG. For a large classroom deployment, another future direction entails exploring small models that can run on-device, similar to the work of Koutcheme et al. (2025) on programming feedback.

The challenges to be overcome to develop AI-math-tutors are very similar to the challenges in developing general AI-mathematics-assistants. Riehl (2025) provides a list of teaching tasks that a machine that can truly understand mathematics should be able to perform, such as generating appropriate examples, grading complex proofs and identifying main ideas in a proof. Achieving these teaching tasks is a stepping stone to building more general mathematics machines that are understandable (clearly expressed via known algorithms), verifiable (via software or proof assistant), and well-sourced (with references to human-generated content) (Riehl, 2025). We hope that future work on systems such as LeanTutor will take steps in these directions.

Acknowledgments

The authors would like to acknowledge support from: NSF CAREER grant ECCS-2240031, the 2023 CITRIS Institute Seed Grant, and the UC Berkeley Instructional Technology and Innovation MicroGrant Program.

Impact Statement

This paper presents work whose goal is to advance the fields of AI for Math and AI for Education.

References

Alur, R., D'Antoni, L., Gulwani, S., and Kini, D. Automated grading of DFA constructions. In *IJCAI'13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pp. 1976–1982, 2013.

Anthropic. Anthropic education report: How university students use claude, April 2025. URL https://www.anthropic.com/news/anthropic-education-report-how-university-students-use-claude.

Askell, A., Bai, Y., Chen, A., Drain, D., Ganguli, D., Henighan, T., Jones, A., Joseph, N., Mann, B., DasSarma, N., et al. A general language assistant as a laboratory for alignment. *arXiv preprint arXiv:2112.00861*, 2021.

Autexier, S., Dietrich, D., and Schiller, M. Towards an intelligent tutor for mathematical proofs. *arXiv preprint arXiv:1202.4828*, 2012.

Avigad, J. Learning logic and proof with an interactive theorem prover. *Proof technology in mathematics research and teaching*, pp. 277–290, 2019.

Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., and Avigad, J. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023a.

Azerbayev, Z., Schoelkopf, H., Paster, K., Santos, M. D., McAleer, S., Jiang, A. Q., Deng, J., Biderman, S., and Welleck, S. Llemma: An open language model for mathematics. *arXiv preprint arXiv:2310.10631*, 2023b.

Aziz, M., Chi, H., Tibrewal, A., Grossman, M., and Sarkar, V. Auto-grading for parallel programs. In *Proceedings of the Workshop on Education for High-Performance Computing*, pp. 1–8, 2015.

Baker, J. D. Students' difficulties with proof by mathematical induction. In *Annual Meeting of the American Educational Research Association*, April 1996.

Balunović, M., Dekoninck, J., Jovanović, N., Petrov, I., and Vechev, M. Mathconstruct: Challenging llm reasoning with constructive proofs. *arXiv preprint arXiv:2502.10197*, 2025.

Barnes, T. and Stamper, J. Toward automatic hint generation for logic proof tutoring using historical student data. In *International conference on intelligent tutoring systems*, pp. 373–382. Springer, 2008.

Billingsley, W. and Robinson, P. Student proof exercises using mathstiles and isabelle/hol in an intelligent book. *Journal of Automated Reasoning*, 39:181–218, 2007.

Boldo, S., Clément, F., Hamelin, D., Mayero, M., and Rousselin, P. Teaching divisibility and binomials with coq. *arXiv preprint arXiv:2404.12676*, 2024.

Briggle, A. et al. Towards an intelligent tutoring system for propositional proof construction. *Current Issues in Computing and Philosophy*, 175:145, 2008.

Bundy, A., Moore, J., and Zinn, C. An intelligent tutoring system for induction proofs. In *CADE-17 Workshop on Automated Deduction in Education*, pp. 4–13, 2000.

Buzzard, K. Teaching formalisation to mathematics undergraduates, July 2022. URL https://xenaproject.wordpress.com/2022/07/29/teaching-formalisation-to-mathematics-undergraduates/.

Buzzard, K., Eugster, J., Pedramfar, M., Bentkamp, A., Massot, P., Carey, S., Farabella, I., and Browne, A. NNG4: Natural number game in lean 4, 2023. URL https://github.com/leanprover-community/NNG4.

Cao, J., Lu, Y., Li, M., Ma, H., Li, H., He, M., Wen, C., Sun, L., Zhang, H., Qin, S., et al. From informal to formal–incorporating and evaluating LLMs on natural language requirements to verifiable formal proofs. *arXiv preprint arXiv:2501.16207*, 2025.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Petroski Such, F., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Cooper, A. *The inmates are running the asylum*. Springer, 1999.

Corbett, A. T., Koedinger, K. R., and Anderson, J. R. Intelligent tutoring systems. In *Handbook of human-computer interaction*, pp. 849–874. Elsevier, 1997.

Cunningham, G., Bunescu, R. C., and Juedes, D. Towards autoformalization of mathematics and code correctness: Experiments with elementary proofs. *arXiv preprint arXiv:2301.02195*, 2023.

D'antoni, L., Kini, D., Alur, R., Gulwani, S., Viswanathan, M., and Hartmann, B. How can automatic feedback help students construct automata? *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):1–24, 2015.

DeepMind, G. AI achieves silver-medal standard solving International Mathematical Olympiad problems, July 2024. URL https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/.

DeNero, J. and Martinis, S. Teaching composition quality at scale: human judgment in the age of autograders. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pp. 421–426, 2014.

Dermeval, D., Paiva, R., Bittencourt, I. I., Vassileva, J., and Borges, D. Authoring tools for designing intelligent tutoring systems: a systematic review of the literature. *International Journal of Artificial Intelligence in Education*, 28:336–384, 2018.

Gadgil, S., Tadipatri, A. R., Agrawal, A., Narayanan, A., and Goyal, N. Towards automating formalisation of theorem statements using large language models. In *36th Conference on Neural Information Processing Systems (NeurIPS 2022) Workshop on MATH-AI*, 2022.

Gao, G., Wang, Y., Jiang, J., Gao, Q., Qin, Z., Xu, T., and Dong, B. Herald: A natural language annotated lean 4 dataset. *arXiv preprint arXiv:2410.10878*, 2024.

Gloeckle, F., Roziere, B., Hayat, A., and Synnaeve, G. Temperature-scaled large language models for lean proofstep prediction. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS*, volume 23, pp. 33, 2023.

Goetze, C. The real reason why students are using ai to avoid learning. *Time*, April 2025. URL https://time.com/7276807/why-students-using-ai-avoid-learning/.

Gupta, A., Reddig, J., Calo, T., Weitekamp, D., and MacLellan, C. J. Beyond final answers: Evaluating large language models for math tutoring. *arXiv preprint arXiv:2503.16460*, 2025.

Head, A., Glassman, E., Soares, G., Suzuki, R., Figueredo, L., D'Antoni, L., and Hartmann, B. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*, pp. 89–98, 2017.

Hecht, R., Liu, R., Zenke, C., and Malan, D. J. Distributing, collecting, and autograding assignments with github classroom. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2*, pp. 1179–1179, 2023.

Hendriks, M., Kaliszyk, C., Raamsdonk, F. v., and Wiedijk, F. Teaching logic using a state-of-the-art proof assistant. 2010.

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. *arXiv preprint arXiv:2402.08957*, 2024.

Huet, G., Kahn, G., and Paulin-Mohring, C. The coq proof assistant a tutorial. *Rapport Technique*, 178:113, 1997.

Iannone, P. and Thoma, A. Interactive theorem provers for university mathematics: an exploratory study of students' perceptions. *International Journal of Mathematical Education in Science and Technology*, 55(10):2622–2644, 2024.

Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S., and Sharma, R. Jigsaw: Large language models meet program synthesis.

In *Proceedings of the 44th International Conference on Software Engineering*, pp. 1219–1231, 2022.

Jiang, A. Q., Li, W., Han, J. M., and Wu, Y. LISA: Language models of ISAbelle proofs. In *6th Conference on Artificial Intelligence and Theorem Proving*, pp. 378–392, 2021.

Jiang, A. Q., Li, W., Tworkowski, S., Czechowski, K., Odrzygóźdź, T., Miłoś, P., Wu, Y., and Jamnik, M. Thor: Wielding hammers to integrate language models and automated theorem provers. *Advances in Neural Information Processing Systems*, 35:8360–8373, 2022a.

Jiang, A. Q., Welleck, S., Zhou, J. P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., and Lample, G. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022b.

Jiang, A. Q., Li, W., and Jamnik, M. Multilingual mathematical autoformalization. *arXiv preprint arXiv:2311.03755*, 2023.

Karsten, N., Jacobsen, F. K., Eiken, K. J., Nestmann, U., and Villadsen, J. Proofbuddy: A proof assistant for learning and monitoring. *arXiv preprint arXiv:2308.06970*, 2023.

Keenan, M. and Omar, C. Learner-centered design criteria for classroom proof assistants. In *Proceedings of 5th Workshop on Human Aspects of Types and Reasoning Assistants (HATRA). Available at https://api. semanticscholar. org/CorpusID*, volume 273399313, 2024.

Kerjean, M., Mayero, M., and Rousselin, P. Maths with coq in l1, a pedagogical experiment. In *13th International Workshop on Theorem proving components for Educational software-ThEdu 2024*, 2024.

Koutcheme, C., Dainese, N., Sarsa, S., Hellas, A., Leinonen, J., Ashraf, S., and Denny, P. Evaluating language models for generating and judging programming feedback. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, pp. 624–630, 2025.

Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., and Liang, P. S. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32, 2019.

Lean Community, T. Defining new syntax — identifiers. https://lean-lang.org/doc/reference/latest/Notations-and-Macros/Defining-New-Syntax/, 2024. https://lean-lang.org/doc/reference/latest/Notations-and-Macros/Defining-New-Syntax/#--tech-term-Identifiers.

Levonian, Z., Henkel, O., Li, C., Postle, M.-E., et al. Designing safe and relevant generative chats for math learning in intelligent tutoring systems. *Journal of Educational Data Mining*, 17(1), 2025.

Li, Z., Sun, J., Murphy, L., Su, Q., Li, Z., Zhang, X., Yang, K., and Si, X. A survey on deep learning for theorem proving. In *First Conference on Language Modeling*, 2024a.

Li, Z., Wu, Y., Li, Z., Wei, X., Zhang, X., Yang, F., and Ma, X. Autoformalize mathematical statements by symbolic equivalence and semantic consistency. *arXiv preprint arXiv:2410.20936*, 2024b.

Lin, Y., Tang, S., Lyu, B., Wu, J., Lin, H., Yang, K., Li, J., Xia, M., Chen, D., Arora, S., et al. Goedel-Prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*, 2025.

Liu, Q., Zheng, X., Lu, X., Cao, Q., and Yan, J. Rethinking and improving autoformalization: Towards a faithful metric and a dependency retrieval-based approach. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=hUb2At2DsQ.

Lodder, J., Heeren, B., Jeuring, J., and Neijenhuis, W. Generation and use of hints and feedback in a hilbert-style axiomatic proof tutor. *International Journal of Artificial Intelligence in Education*, 31:99–133, 2021.

Lu, J., Wan, Y., Huang, Y., Xiong, J., Liu, Z., and Guo, Z. Formalalign: Automated alignment evaluation for autoformalization. *arXiv preprint arXiv:2410.10135*, 2024a.

Lu, J., Wan, Y., Liu, Z., Huang, Y., Xiong, J., Liu, C., Shen, J., Jin, H., Zhang, J., Wang, H., et al. Process-driven autoformalization in lean 4. *arXiv preprint arXiv:2406.01940*, 2024b.

Massot, P. Teaching mathematics using lean and controlled natural language. In *15th International Conference on Interactive Theorem Proving (ITP 2024)*, pp. 27–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

mathlib Community, T. The lean mathematical library. In *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. ACM, 2020.

Maurya, K. K., Srivatsa, K., Petukhova, K., and Kochmar, E. Unifying ai tutor evaluation: An evaluation taxonomy for pedagogical ability assessment of llm-powered ai tutors. *arXiv preprint arXiv:2412.09416*, 2024.

Messer, M., Brown, N. C., Kölling, M., and Shi, M. Automated grading and feedback tools for programming education: A systematic review. *ACM Transactions on Computing Education*, 24(1):1–43, 2024.

Miller, P. and DiCerbo, K. LLM based math tutoring: Challenges and dataset, 2024.

Mitra, C., Miroyan, M., Jain, R., Kumud, V., Ranade, G., and Norouzi, N. RetLLM-E: retrieval-prompt strategy for question-answering on student discussion forums. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 23215–23223, 2024.

Mitra, J. Studying the impact of auto-graders giving immediate feedback in programming assignments. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pp. 388–394, 2023.

Moura, L. d. and Ullrich, S. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, pp. 625–635. Springer, 2021.

Mozafari, J., Piryani, B., Abdallah, A., and Jatowt, A. HintEval: A comprehensive framework for hint generation and evaluation for questions. *arXiv preprint arXiv:2502.00857*, 2025.

Murphy, L., Yang, K., Sun, J., Li, Z., Anandkumar, A., and Si, X. Autoformalizing Euclidean geometry. *arXiv preprint arXiv:2405.17216*, 2024.

Omar, C., Voysey, I., Chugh, R., and Hammer, M. A. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–32, 2019.

OpenAI. GPT-4 technical report, 2023. URL https://arxiv.org/abs/2303.08774.

OpenAI. College students and chatgpt adoption in the us, February 2025. URL https://openai.com/global-affairs/college-students-and-chatgpt/.

Pardos, Z. A. and Bhandari, S. Learning gain differences between ChatGPT and human tutor generated algebra hints. *arXiv preprint arXiv:2302.06871*, 2023.

Park, H. and Manley, E. D. Using ChatGPT as a proof assistant in a mathematics pathways course. *The Mathematical Education*, 63(2):139–163, 2024.

Patel, N., Saha, R., and Flanigan, J. A new approach towards autoformalization. *arXiv preprint arXiv:2310.07957*, 2023.

Paulson, L. C. *Isabelle: A generic theorem prover*. Springer, 1994.

Phung, T., Pădurean, V.-A., Singh, A., Brooks, C., Cambronero, J., Gulwani, S., Singla, A., and Soares, G. Automating human tutor-style programming feedback: Leveraging gpt-4 tutor model for hint generation and gpt-3.5 student model for hint validation. In *Proceedings of the 14th learning analytics and knowledge conference*, pp. 12–23, 2024.

Poesia, G., Broman, D., Haber, N., and Goodman, N. Learning formal mathematics from intrinsic motivation. *Advances in Neural Information Processing Systems*, 37: 43032–43057, 2024.

Poiroux, A., Kuncak, V., and Bosselut, A. Leaninteract: A python interface for lean 4, 2025. URL https://github.com/augustepoiroux/LeanInteract.

Polu, S. and Sutskever, I. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Polu, S., Han, J. M., Zheng, K., Baksys, M., Babuschkin, I., and Sutskever, I. Formal mathematics statement curriculum learning. *arXiv preprint arXiv:2202.01344*, 2022.

Ren, Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., et al. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

Riehl, E. Testing artificial mathematical intelligence. https://emilyriehl.github.io/files/testing.pdf, 2025.

Sanchez-Stern, A., Alhessi, Y., Saul, L., and Lerner, S. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 1–10, 2020.

Scarfe, P., Watcham, K., Clarke, A., and Roesch, E. A real-world test of artificial intelligence infiltration of a university examinations system: A "Turing Test" case study. *PloS one*, 19(6):e0305354, 2024.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y., Wu, Y., et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Sieg, W. The AProS project: Strategic thinking & computational logic. *Logic Journal of the IGPL*, 15(4):359–368, 2007.

Singh, R., Gulwani, S., and Solar-Lezama, A. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN* *conference on Programming language design and implementation*, pp. 15–26, 2013.

Sonkar, S., Ni, K., Chaudhary, S., and Baraniuk, R. G. Pedagogical alignment of large language models. *arXiv preprint arXiv:2402.05000*, 2024.

Sufrin, B. and Bornat, R. Jnj in Jape. 1997.

Suraweera, P. and Mitrovic, A. KERMIT: A constraint-based tutor for database modeling. In *Intelligent Tutoring Systems: 6th International Conference, ITS 2002 Biarritz, France and San Sebastian, Spain, June 2–7, 2002 Proceedings 6*, pp. 377–387. Springer, 2002.

Suzuki, R., Soares, G., Glassman, E., Head, A., D'Antoni, L., and Hartmann, B. Exploring the design space of automatically synthesized hints for introductory programming assignments. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pp. 2951–2958, 2017.

Szegedy, C., Rabe, M., and Michalewski, H. Retrieval-augmented proof step synthesis. In *Conference on Artificial Intelligence and Theorem Proving (AITP)*, volume 4, 2021.

Tarrach, G., Jiang, A. Q., Raggi, D., Li, W., and Jamnik, M. More details, please: Improving autoformalization with more detailed proofs. In *AI for Math Workshop@ ICML 2024*, 2024.

Thakur, A., Tsoukalas, G., Wen, Y., Xin, J., and Chaudhuri, S. An in-context learning agent for formal theorem-proving. *arXiv preprint arXiv:2310.04353*, 2023.

Thakur, A., Tsoukalas, G., Durrett, G., and Chaudhuri, S. ProofWala: Multilingual proof data synthesis and theorem-proving. *arXiv e-prints*, pp. arXiv–2502, 2025.

Thoma, A. and Iannone, P. Learning about proof with the theorem prover lean: the abundant numbers task. *International Journal of Research in Undergraduate Mathematics Education*, pp. 1–30, 2022.

Tonga, J. C., Clement, B., and Oudeyer, P.-Y. Automatic generation of question hints for mathematics problems using large language models in educational technology. *arXiv preprint arXiv:2411.03495*, 2024.

Trinh, T. H., Wu, Y., Le, Q. V., He, H., and Luong, T. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

Tyen, G., Mansoor, H., Cărbune, V., Chen, Y. P., and Mak, T. LLMs cannot find reasoning errors, but can correct them given the error location. In *Findings of the Association for Computational Linguistics ACL 2024*, pp. 13894–13908, 2024.

VanLehn, K. The behavior of tutoring systems. *International journal of artificial intelligence in education*, 16 (3):227–265, 2006.

Villadsen, J. and Jacobsen, F. K. Using isabelle in two courses on logic and automated reasoning. In *Formal Methods Teaching Workshop*, pp. 117–132. Springer, 2021.

Wang, H., Xin, H., Zheng, C., Li, L., Liu, Z., Cao, Q., Huang, Y., Xiong, J., Shi, H., Xie, E., et al. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656*, 2023a.

Wang, H., Yuan, Y., Liu, Z., Shen, J., Yin, Y., Xiong, J., Xie, E., Shi, H., Li, Y., Li, L., et al. Dt-solver: Automated theorem proving with dynamic-tree sampling guided by proof-level value function. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12632–12646, 2023b.

Wang, H., Xin, H., Liu, Z., Li, W., Huang, Y., Lu, J., Yang, Z., Tang, J., Yin, J., Li, Z., et al. Proving theorems recursively. *arXiv preprint arXiv:2405.14414*, 2024a.

Wang, R., Zhang, J., Jia, Y., Pan, R., Diao, S., Pi, R., and Zhang, T. Theoremllama: Transforming general-purpose llms into lean4 experts. *arXiv preprint arXiv:2407.03203*, 2024b.

Wang, R. E., Ribeiro, A. T., Robinson, C. D., Loeb, S., and Demszky, D. Tutor CoPilot: A human-AI approach for scaling real-time expertise. *arXiv preprint arXiv:2410.03017*, 2024c.

Welleck, S. and Saha, R. Llmstep: Llm proofstep suggestions in lean. *arXiv preprint arXiv:2310.18457*, 2023.

Welleck, S., Liu, J., Lu, X., Hajishirzi, H., and Choi, Y. Naturalprover: Grounded mathematical proof generation with language models. *Advances in Neural Information Processing Systems*, 35:4913–4927, 2022.

Wemmenhove, J., Arends, D., Beurskens, T., Bhaid, M., McCarren, S., Moraal, J., Garrido, D. R., Tuin, D., Vassallo, M., Wils, P., et al. Waterproof: educational software for learning how to write mathematical proofs. *arXiv preprint arXiv:2211.13513*, 2022.

Wu, Y., Jiang, A. Q., Li, W., Rabe, M., Staats, C., Jamnik, M., and Szegedy, C. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35:32353–32368, 2022.

Wu, Z., Huang, S., Zhou, Z., Ying, H., Wang, J., Lin, D., and Chen, K. Internlm2. 5-stepprover: Advancing automated theorem proving via expert iteration on large-scale lean problems. *arXiv preprint arXiv:2410.15700*, 2024.

Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R. J., and Anandkumar, A. LeanDojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

Yang, K., Poesia, G., He, J., Li, W., Lauter, K., Chaudhuri, S., and Song, D. Formal mathematical reasoning: A new frontier in ai. *arXiv preprint arXiv:2412.16075*, 2024.

Yeh, E., Hitaj, B., Owre, S., Quemener, M., and Shankar, N. CoProver: a recommender system for proof construction. In *International Conference on Intelligent Computer Mathematics*, pp. 237–251. Springer, 2023.

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. *arXiv preprint arXiv:2406.03847*, 2024a.

Ying, H., Zhang, S., Li, L., Zhou, Z., Shao, Y., Fei, Z., Ma, Y., Hong, J., Liu, K., Wang, Z., et al. Internlm-math: Open math large language models toward verifiable reasoning. *arXiv preprint arXiv:2402.06332*, 2024b.

Zhao, C., Silva, M., and Poulsen, S. Autograding mathematical induction proofs with natural language processing. *arXiv preprint arXiv:2406.10268*, 2024a.

Zhao, X., Li, W., and Kong, L. Subgoal-based demonstration learning for formal theorem proving. In *Forty-first International Conference on Machine Learning*, 2024b.

Zheng, C., Wang, H., Xie, E., Liu, Z., Sun, J., Xin, H., Shen, J., Li, Z., and Li, Y. Lyra: Orchestrating dual correction in automated theorem proving. *arXiv preprint arXiv:2309.15806*, 2023.

Zhou, J. P., Staats, C., Li, W., Szegedy, C., Weinberger, K. Q., and Wu, Y. Don't trust: Verify–grounding llm quantitative reasoning with autoformalization. *arXiv preprint arXiv:2403.18120*, 2024.

A. Appendix

A.1. Extended Review of Math Proof Tutors

We identify three main categories of autonomous proof tutoring systems: (1) intelligent tutoring systems, (2) LLM-based tutoring systems, and (3) theorem prover based systems. Each of these systems has unique advantages, which LeanTutor attempts to build upon.

A.1.1. Intelligent Tutoring Systems.

Corbett et al. (1997) characterize a system as an *intelligent tutoring system* (ITS) if it fulfills eight design principles, which include: scaffolding student learning, modeling students' learning trajectories over time, and providing immediate feedback. Researchers have made attempts to develop (Autexier et al., 2012; Briggle et al., 2008) or developed ITS for math proofs (Barnes & Stamper, 2008; Lodder et al., 2021; Bundy et al., 2000). ITS maintain a high quality of education through expert authoring of solutions or feedback, but this also makes them difficult to develop and scale (Dermeval et al., 2018). To reduce this burden, LeanTutor dynamically generates proof trees based on student solutions, similar to Lodder et al. (2021) approach, but in contrast, also generates feedback on-demand via a generative language model.

A.1.2. LLM-Based Tutors.

Given the extremely recent advance of high performance LLMs, there are not yet many LLM-based math tutors for proofs specifically. Zhao et al. (2024a) propose an LLM-based autograder for inductive proofs, which provides students with real-time feedback on the correctness of their proofs. Park & Manley (2024) evaluated ChatGPT's abilities to aid students in refining and improving their proofs. Broadly speaking, many LLM-based math tutors have been developed and studied (Tonga et al., 2024; Miller & DiCerbo, 2024; Autexier et al., 2012; Wang et al., 2024c; Park & Manley, 2024). These math tutors have shown to maintain conversations without inappropriate content (Levonian et al., 2025) and even lead to learning gains for students studying algebra (Pardos & Bhandari, 2023). However, LLMs still cannot suffice as effective tutors due to (1) hallucinations,(Maurya et al., 2024; Balunović et al., 2025) (2) models revealing the whole answer (Sonkar et al., 2024), (3) models do not necessarily provide the correct reasoning behind an answer (Gupta et al., 2025), and (4) models struggle to identify mistakes (Tyen et al., 2024; Miller & DiCerbo, 2024). LeanTutor capitalizes on the conversational ability of LLMs, but "outsources" reasoning tasks to theorem provers.

A.1.3. Proof Assistant-based Tutors.

Theorem provers, such as Lean (Moura & Ullrich, 2021), Coq (Huet et al., 1997), and Isabelle (Paulson, 1994), have all been used by some math educators as tools to teach students proofs (Avigad, 2019; Villadsen & Jacobsen, 2021; Boldo et al., 2024; Kerjean et al., 2024). Additionally, proof tutors or educationally-geared tools have been developed on top of theorem provers: ProofTutor using APRoS (Sieg, 2007), ProofWeb (Hendriks et al., 2010) based on Coq, JAPE (Sufrin & Bornat, 1997), Waterproof (Wemmenhove et al., 2022) built on Coq, HazelProver built on Agda (Omar et al., 2019; Keenan & Omar, 2024), Verbose Lean based on Lean (Massot, 2024), and MathsTiles build on Isabelle/HOL (Billingsley & Robinson, 2007). These tools have led to unique benefits in students' learning of proofs (Thoma & Iannone, 2022), but students struggle to learn the complex syntax required to interact with most (Avigad, 2019; Buzzard, 2022; Villadsen & Jacobsen, 2021; Karsten et al., 2023). LeanTutor combats this issue by allowing the student to interface only in natural language and hiding the Lean formalizations of student proofs altogether.

A.2. Proofs from PeanoBench

The PeanoBench dataset contains three main subsets of proofs: *staff solution* proofs, *correct* proofs, and *incorrect* proofs. *Correct* proofs are derived from the staff solution proofs, with two main differences: (1) Lean syntax in the proof is changed when possible and (2) the NL in-line comments are in differing "personas" (the equation-based and justification-based personas). Figure 3 demonstrates the *staff solution* proof of the theorem `add_comm` (proving the commutativity of addition) as well as the equation-based and justification-based commented versions of the original proof (with small changes in Lean code). Figure 4 is an example of an incorrect proof of `add_comm`, created by skipping a step in the justification-based persona proof.

```
theorem add_comm_staff_solution (a b : ℕ) : a + b = b + a := by
  -- Induct on b, with d = 0 as the base case and the inductive hypothesis a + d = d + a. There are now two proof goals, prove base case: a + 0 = 0 + a and the inductive step: a + succ d = succ d + a
  induction b with d hd
  -- First prove base case. Simplify LHS a + 0 to a.
  rw [add_zero]
  -- Simplify RHS 0 + a to a
  rw [zero_add]
  -- Prove LHS and RHS are equal, a = a, completing the base case.
  rfl
  -- Now prove the inductive step. Rewrite LHS a + succ (d) to succ (a + d)
  rw [add_succ]
  -- Rewrite RHS succ (d) + a to succ (d + a)
  rw [succ_add]
  -- Rewrite LHS succ (a + d) to succ (d + a) using the inductive hypothesis
  rw [hd]
  -- Prove succ LHS and RHS are equal, (d + a) = succ (d + a), completing the proof
  rfl
```

```
theorem add_comm_equation_based (a b : ℕ) : a + b = b + a := by
  -- Start by inducting on b
  induction b with d hd
  -- 0 + a -> a on RHS giving us a + 0 = a
  rw [zero_add]
  -- a + 0 -> a into the LHS to get a = a
  rw [add_zero]
  -- a=a, we are done with the base case
  rfl
  -- a + succ d -> succ (a + d) on LHS giving us succ (a + d) = succ d + a
  rw [add_succ]
  -- succ d + a -> succ (d + a) on RHS giving us succ (a + d) = succ (d + a)
  rw [succ_add]
  -- using the induction hypothesis, succ (a + d) -> succ (d + a) on the LHS giving us succ (d + a) = succ (d + a)
  rw [hd]
  -- succ (d + a) = succ (d + a), we are done.
  rfl
```

```
theorem add_comm_justification_based (a b : ℕ) : a + b = b + a := by
  -- Start by inducting on b
  induction b with d hd
  -- We start with the base case. using properties of addition by 0 we can rewrite a + 0 to a on the LHS
  rw [add_zero]
  -- using properties of addition by 0 we can rewrite 0 + a to a on the RHS
  rw [zero_add]
  -- since both sides are equal, we are done with the base case
  rfl
  -- Now to the (n+1) step. using properties of successors, succ (n) + a -> succ (n + a) and substitute this into the RHS
  rw [succ_add]
  -- using properties of succession, we substitute a + succ(n) -> succ(a+n) on the RHS
  rw [add_succ]
  -- Use the induction hypothesis on the LHS to substitute succ (a + n) -> succ (n + a)
  rw [hd]
  -- since both sides are equal, we are done with the proof
  rfl
```

Figure 3. Examples of annotated Peano Arithmetic proofs from PeanoBench for the theorem proving commutativity of addition, that is, for all $a, b \in \mathbb{N}$, $a + b = b + a$. The first proof,`add_comm_staff_solution` follows the exact Lean code from NNG4. The second and third proofs, `add_comm_equation_based` and `add_comm_justification_based`, are written in two different personas.

```
theorem add_comm_incorrect (a b : ℕ) : a + b = b + a := by
  -- Start by inducting on b
  induction b with d hd
  -- We start with the base case using properties of addition by 0 we can rewrite a + 0 to
 a on the LHS
  rw [add_zero]
  -- using properties of addition by 0 we can rewrite 0 + a to a on the RHS
  rw [zero_add]
  -- since both sides are equal, we are done with the base case
  rfl
  -- Now to the (n+1) step. using properties of successors, succ (n) + a -> succ (n + a)
and substitute this into the RHS
  rw [succ_add]
  -- using properties of succession, we substitute a + succ(n) -> succ(a+n) on the RHS
  rw [add_succ]
  -- since both sides are equal, we are done with the proof
  rfl
```

Figure 4. Example of an incorrect proof for the theorem proving commutativity of addition, that is, for all $a, b \in \mathbb{N}$, $a + b = b + a$. This proof, originally the justification-based persona, has the `rw [hd]` step, which applies the inductive hypothesis, skipped.

A.3. Proof Breakdown by Worlds

NNG4 categorizes proofs based on distinct worlds. The table below presents the distribution of proofs across these worlds, illustrating the relative frequency of each category.

World	# Tactics	# Proofs
Implication	38	13
Multiplication	57	9
Advanced Multiplication	66	10
Algorithm	20	5
Less or Equal	86	11
Power	70	9
Tutorial	24	7
Advanced Addition	32	6
Addition	41	5
Total	**434**	**75**

Table 3. Distribution of selected proofs from NNG4 by world.

A.4. PeanoBench: Incorrect Proof Generation Algorithm

Algorithm 1 StepSkipping

for $P \in$ CorrectDeviatingProofs **do**
  $n \leftarrow$ length(P)
  if $n = 2$ **or** $n = 3$ **then**
    delete step 2
  else if $n = 4$ **then**
    randomly delete step $n - 1$ or $n - 2$
  else if $n > 4$ **then**
    randomly delete one of step $n - 1$, $n - 2$, or $n - 3$
  end if
end for

Algorithm 1: Step-skipping algorithm for generating incorrect proofs.

A.5. Next Step Generator

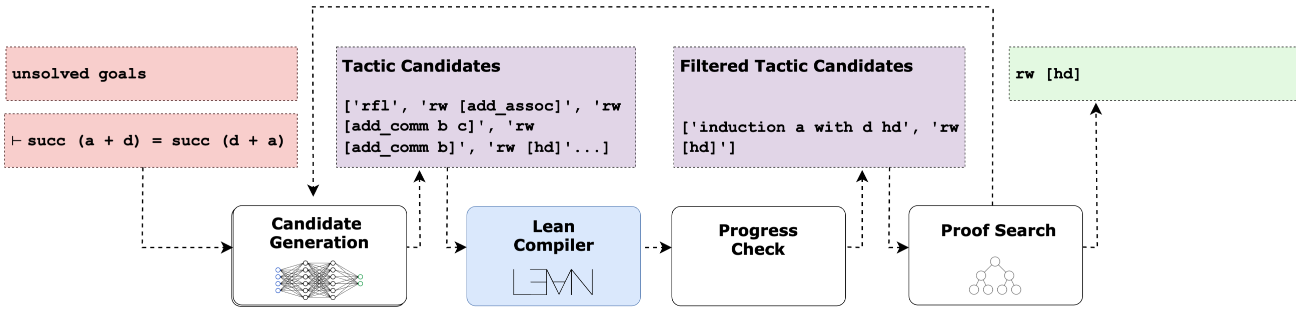

Figure 5. Architecture of the Next Step Generation module. An LLM generates tactic candidates which are appended to the pre-existing proof. Tactics that compile correctly are then passed through a "progress check" filter which ensures goal states are not being re-visited. This process of generating and checking tactics is repeated until the proof is completed.

A.6. Autoformalizer Extended Results

We additionally experiment with adding the following information into the autoformalizer prompt. All formalizations were generated step-by-step (Section 4.1).

Experiments include:

- *Staff Solution*: The staff solution proof, a complete and correct proof for the theorem in both NL and FL. The autoformalizer accuracy with the staff solution is also presented in the main paper.
- *Previous NL*: The student's previous proof steps (in natural language) up until that point,
- *Previous Goal State*: The Lean goal state of the proof formed by appending autoformalizations of the student's NL proof to the Lean theorem statement. (Note that this goal state may become "corrupted" if any previous formalizations were incorrect. If a goal state displays an error message, we did not include the goal state in the prompt and the prompt was then identical to the baseline.)

The results of these experiments (in addition to experiments discussed in the main paper) are summarized in Table 4.

Condition	Correct Tactics	Correct Proofs	Incorrect Proofs
Baseline	296 / 900 = 32.89%	10 / 150 = 6.67%	21 / 146 = 14.38%
+ Staff Solution	511 / 900 = 56.78%	27 / 150 = 18.00%	44 / 146 = 30.14%
+ Previous Goal State	312 / 900 = 34.67%	15 / 150 = 10.00%	29 / 146 = 19.86%
+ Previous NL	331 / 900 = 36.78%	10 / 150 = 6.67%	20 / 146 = 13.70%
+ Previous NL + Staff Solution	522 / 900 = 58.00%	28 / 150 = 18.67%	42 / 146 = 28.77%
Whole Proof (Baseline)	254 / 900 = 28.22%	16 / 150 = 10.67%	19 / 146 = 13.01%
+ Whole Proof (Staff Solution)	466 / 900 = 51.78%	40 / 150 = 26.67%	32 / 146 = 21.92%

Table 4. Extended autoformalizer experiment results.

A.7. Metric

Since we are interested in faithful autoformalization, we measure the accuracy of our autoformalizer on a tactic-by-tactic basis. For this, we check that check either the tactic itself or the proof state after every tactic matches the corresponding ground truth tactic/proof state. First, the tactics themselves are compared using exact string matching, with the minor exception that `rw [...` and `rw[...` (the only difference between the strings is the space before the brackets) are considered equivalent. This covers a lot of cases, but sometimes two tactics behave identically, but are not literally the same string (for example, `rw [add_comm]` and `rw [add_comm a b]` might do the same thing in a proof, but string matching would fail). Additionally, two tactics might use different variable names (for example, `induction n with d hd` and `induction n with k hk` are equally valid). So, we cannot just use exact string matching.

If string matching does not identify the tactics as identical, then the tactics are verified in Lean (appended to any previous tactics for the predicted and ground truth proofs respectively) and we check if the resulting proof states are syntactically identical up to variable naming. If either the string matching or proof state matching check succeeds, the generated tactic is considered correct. By "up to variable naming", we mean that two goals are considered equivalent if they are structurally the same, but may use different variable names. For example, the following proof states are identical up to variable naming, but neither of them are exact string matches.

```
n : ℕ
h : 1 ≤ n
⊢ n + 0 = n
```

```
m : ℕ
hm : 1 ≤ m
⊢ m + 0 = m
```

A.7.1. Proof State Comparison

The algorithm to compare proof states up to variable renaming works as follows. First, the proof states are split into cases and each case is compared individually. All cases must be equivalent for the proof states to be considered equivalent. Then, within each case, free variables (which are not bound by a binder and can be seen for the first time above the $\vdash$) [4] are identified by checking what appears before the first colon on each line. In the proof states below, `n` and `hn` in the first proof and `m` and `hm` in the second proof are all free variables. After identifying free variables, the proof states are normalized by renaming each appearance of a variable according to its position in the variable list (see Algorithm 2). [5]

The proof state normalization algorithm is written in Algorithm 2. To normalize a proof case (one case in a proof state), we make a list of all variables (including proofs) in the local context, which includes everything listed before a colon in a line above the $\vdash$. Next, we locate all identifiers in the goal states we are comparing via a Python implementation of Lean identifiers (Lean Community, 2024). An identifier in Lean is a string that acts as a variable name or refers to a constant such as a theorem or a type. For example, `x` and `MyNat.add_comm` are both identifiers. Identifiers that match a variable name are replaced with `var`*i*, where *i* is the index of the variable in the variable list created earlier. To locate identifiers, we use a greedy algorithm which loops through all characters in the proof state.

So, for example, the following proof states,

```
n : ℕ
h : 1 ≤ n
⊢ n + 0 = n
```

```
m : ℕ
hm : 1 ≤ m
⊢ m + 0 = m
```

would both be converted to

```
var0 : ℕ
var1 : 1 ≤ n
⊢ var0 + 0 = var0
```

[4] Lean supports three types of variables: bound variables, which first appear under a *binder* such as $\forall$ or `fun`; free variables, which are not bound by a binder and can be seen for the first time above the $\vdash$; and meta-variables, which represent holes in an expression that must be filled in before the proof is complete. Only free variables are supported for variable renaming; bound variables and meta-variables are not handled because they rarely ever appear within proof states in NNG4 and handling them would amount to a drastic increase in complexity.

[5] Our code to determine what constitutes a valid Lean identifier does not handle double guillemets (and) because they are not used in NNG4.

The algorithm *Normalize* is below. Note that *GetVariables* is a function that collects all the variables from proof state as described earlier.

Algorithm 2 Normalize Proof State

```
function Normalize(proof_state)
variable_list ← GetVariables(proof_state)
result ← ""
i ← 0
while i < len(proof_state) do
    ident ← LongestIdentifierStartingAt(proof_state, i)
    if ident ≠ Null then
        if ident ∈ variable_list then
            result ← result + "var" + IndexOf(ident, variable_list)
        else
            result ← result + ident
        end if
        i ← i+ len(ident)
    else
        result ← result + GetChar(proof_state, i)
        i ← i + 1
    end if
end while
return result
```

The *Normalize* algorithm relies on the *LongestIdentifierStartingAt* algorithm as described below.

Algorithm 3 Longest Identifier

```
function LongestIdentifierStartingAt(str, i)
len ← 0
if IsValidLeanIdentifier(Substring(str, i, i + 2)) then
    len ← 2
end if
while IsValidLeanIdentifier(Substring(str, i, i + len + 1)) and i + len < Len(str) do
    len ← len +1
end while
if len > 0 then
    return Substring(str, i, i + len)
else
    return Null
end if
```

A.8. Examples of Generated Hints

We demonstrate an example of the three generated hint types (error identification, question, bottom-out hint) for the following theorem. The incorrect proof skips the `tauto` tactic and results in the base case never being proven. Unfortunately, LeanTutor's feedback fails to get at the heart of the student's issue and seems more inclined towards the inductive case.

```
-- Theorem Statement: Prove that if a is not equal to 0, then a is the successor of some natural number.
theorem eq_succ_of_ne_zero_correct (a : ) (ha : a  0) :  n, a = succ n := by
  -- Induct on a
  induction a with d _
  -- For the base case, a = 0, we have a contradiction in hypotheses because we know a  0.
  tauto
  -- For the inductive step, we set n to be d.
  use d
  -- succ d = succ d
  rfl
```

```
-- Theorem Statement: Prove that if a is not equal to 0, then a is the successor of some natural number.
  theorem eq_succ_of_ne_zero_incorrect (a : ) (ha : a  0) :  n, a = succ n := by
  -- We use induction on a.
  induction a with d _
  -- For the inductive step, we set n to be d which gives us the goal succ d = succ d
  use d -- incorrect
  -- Since the LHS and RHS are equal, we can use reflexivity to prove the goal.
  rfl -- incorrect
```

1. Error Identification: You are incorrectly inducting on the variable 'a' instead of its successor form, which leads to confusion in your proof.

[Accuracy: 1, Relevance: 1, Readability: 4, Answer Leakage: 5]

2. Question: Do you think you should consider the structure of 'a' in terms of its base case and successor?

[Accuracy: 3, Relevance: 2, Readability: 3, Answer Leakage: 4]

3. Bottom-out Hint: The next step is to use the fact that if 'a' is a successor, you should analyze 'a' as 'succ d' for some natural number 'd'.

[Accuracy: 4, Relevance: 3, Readability: 5, Answer Leakage: 1]

A.9. Cold-start Proof Results

A "cold-start" proof is a proof in which the student does not know how to start the proof. For this experiment, we use `gpt-4` (OpenAI, 2023) for both the baseline and LeanTutor. LeanTutor is given no student input and the system is asked to generate feeback types 2 (hint/question) and 3 (next step). We develop a simple baseline, providing the model with the erroneous student proof and prompting the model to generate the two feedback types. For the LeanTutor model, since there is no NL from the student in this case, we do not run the Autoformalizer or Next Step Generator. Instead, the system directly extracts the first step in the proof (in Lean) from the available *staff-solution*, and this is passed to the Feedback Generation module. We evaluate 18 cold start proofs, two from each world. The results (Table 5) indicate that LeanTutor outperforms the baseline on Accuracy and Relevance axes.

Feedback Type	Accuracy	Relevance	Readability	Answer Leakage
Baseline Hint/Question	3.6	3.2	**4.9**	**4.5**
LeanTutor Hint/Question	**4.3**	**4.4**	4.4	4.1
Baseline Next Step	3.6	3.2	4.9	N/A
LeanTutor Next Step	**3.9**	**4.8**	4.9	N/A

Table 5. Average (across all proofs) qualitative scores of generated feedback from baseline and LeanTutor experiments on 18 cold-start proofs. A score closer to 5 indicates desired performance.

A.10. Model Prompts

A.10.1. Autoformalizer Prompts

Autoformalizer prompt for step-by-step formalization contains the system and user prompts used for autoformalization. The following were given as input to the system prompt: the theorem statement of the proof (in NL and FL), the theorem and tactic dictionaries, five hard-coded examples of the autoformalization task, and the staff solution. The prompt for full proof generation, in *Autoformalizer prompt for whole proof formalization*, is the same, except the five hard-coded examples were adjusted to whole proof translations, to match the whole proof autoformalization task.

Autoformalizer prompt for step-by-step formalization

```
### System:
An undergraduate student is proving the following Peano Arithmetic theorem:
Theorem statement in natural language:  {theorem_statement_NL}
Theorem statement in formal language:  {theorem_statement_FL}


Convert the student's natural language mathematical proof step to Lean4
syntax.


[If staff_solution is provided]
This is one example of the completed proof in Lean4, with in-line comments
of the natural language proof corresponding to the Lean4 syntax:
whole_theorems[theorem_name]


These are the formal theorems you have access to:
{theorem_dict}


These are the Lean tactics you have access to:
{tactic_dict}


Your response must be written as a single line of Lean tactic code, as used
in the body of a by block of a Lean theorem.It should match the structure of
Lean DSL tactic proofs, such as:
intro h
rw [← is_zero_succ a]
apply succ_inj at h
exact h
contrapose!  h


Note:  Only 1 lean tactic, do not write multiple lean tactics that are comma
seperated.

DO *NOT* wrap your answer in markdown syntax, e.g.  '''lean '''.  It must
be simply a Lean tactic script that can be inserted into a proof.


Here are some examples.  NOTE: These are just examples.  The correct Lean4
code may not necessarily use the propositions shown in these proofs.
```

All strings in typewriter font are runtime placeholders. {`theorem_statement_NL`} – theorem in natural language; {`theorem_statement_FL`} – the same theorem in Lean's formal syntax; {`whole_theorems[theorem_name]`} – The staff solution; {`theorem_dict`} – dictionary of Peano-arithmetic facts available to the model; {`tactic_dict`} – dictionary of Lean tactics the model may use; {`prev_goal`} – current Lean proof state ; {`prev_nl`} – previous student proof lines ; {`nl_statement`} – the natural-language step to be converted. The optional block, corresponding to the staff solution, renders optionally.

```
Example 1:
Input:  Rewrite the LHS pred (succ a) with the given statement that succ a =
succ b, LHS is now pred (succ b)
Output:  rw [h]


Example 2:
Input:  Rewrite LHS using the commutative property of addition, changing a +
(b + c) to a + b + c
Output:  rw [← add_assoc]


Example 3:
Input:  Assume that the hypothesis 'h' is true, that is, a + succ d = 0.
The goal now is to prove that a = 0.
Output:  rw [add_zero] at h

Example 4:
Input:  Split the natural number 'b' into two cases:  'b' is zero, and 'b'
is the successor of another natural number 'd'.
Output:  cases b with d


Example 5:
Input:  Use the case of a + b to simplify the goal to equal z = x + (a + b).
Output:  use a + b


### User:  The natural-language statement to formalize is:
{nl_statement}
```

Autoformalizer prompt for whole proof formalization

```
### System:
An undergraduate student is proving the following Peano Arithmetic theorem:
Theorem statement in natural language:  {theorem_statement_NL}
Theorem statement in formal language:  {theorem_statement_FL}


Convert the student's natural language mathematical proof to Lean4 syntax.


[If staff_solution is provided]
This is one example of the completed proof in Lean4, with in-line comments
```

```
of the natural language proof corresponding to the Lean4 syntax:
whole_theorems[theorem_name]


These are the formal theorems you have access to:
{theorem_dict}


These are the Lean tactics you have access to:
{tactic_dict}


Your response must be written as a proof in Lean, in a list of tactics on
each new line.  SUch as:
intro h
rw [← is_zero_succ a]
apply succ_inj at h
exact h
contrapose!  h


Each tactic must be formatted consistently with Lean4's syntax and DO NOT
include any comments in the list.
DO *NOT* wrap your answer in markdown syntax, e.g.  '''lean '''.  It must be
simply a list of Lean tactics separated by \n.

Here are some examples.  NOTE: These are just examples.  The correct Lean4
code may not necessarily use the propositions shown in these proofs.


Example 1:
Input:  Induct on b, with d = 0 as the base case and the inductive
hypothesis a * d = d * a.  There are now two proof goals, prove base case:
a * 0 = 0 * a, and inductive step:  a * succ d = succ d * a.
First we prove base case.
Simplify RHS 0 * a to 0.
Simplify LHS a * 0 to 0.
Prove LHS and RHS are equal, 0 = 0, completing base case.
Next prove inductive step.  Rewrite RHS succ d * a to d * a + a.
Rewrite the RHS from d * a + a to a * d + a using the inductive hypothesis.
Rewrite the LHS, changing a * succ d to a * d + a.
Prove LHS and RHS are equal, a * d + a = a * d + a, completing the proof.
Output:  induction b with d hd
rw [zero_mul]
rw [mul_zero]
rfl
rw [succ_mul]
rw [← hd]
rw [mul_succ]
rfl


Example 2:
Input:  We must assume succ (succ 0) + succ (succ 0) = succ (succ (succ
```

```
(succ (succ 0)))) and derive a contradiction or falsehood.
Using our previous theorems, we can change succ (succ 0) + succ (succ 0)
into succ (succ (succ (succ 0))).
By the injectivity of succ, we know that 0 = succ 0.  0 is not equal to the
successor of any natural number, so we have a contradiction.
Thus, we have a falsehood/contradiction, which is what we wanted to show.
Output:  intro h
rw [add_succ, add_succ, add_zero] at h
repeat apply succ_inj at h
apply zero_ne_succ at h
exact h


Example 3:
Input:  We consider the case where the successor of x is less than or equal
to the successor of y.  This implies that the successor of y is equal to the
successor of x plus some natural number d.
We assume d as the difference such that when added to x results in y.  The
goal now is to prove that y is equal to x plus d.
We rewrite the right-hand side of succ y = succ x + d using the theorem that
states the the successor of a sum of two natural numbers is the same as the
successor of the first number added to the second number.
We apply the property that if two natural numbers with successors are equal,
then the original numbers are also equal.
We have shown that x = y + d, so we can use this to prove the goal.
Output:  cases hx with d hd
use d
rw [succ_add] at hd
apply succ_inj at hd
exact hd


Example 4:  Input:  We use proof by contraposition.  So, we assume succ m =
succ n and show m = n.
By the injectivity of succ, we have m = n.
So, m = n, which is exactly what we wanted to show.
Output:  contrapose!  h
apply succ_inj at h
exact h


Example 5:
Input:  Rewrite the expression for the square of (a + b), a2̂, and b2̂ to be
(a + b) * (a + b), a * a, and b * b respectively.
Rearrange the terms on the right hand side of the equation, swapping the
order of b * b and 2 * a * b.  This is based on the commutative property
of addition, which states that the order of the terms does not change the
result of the addition.
Rewrite the left-hand side of the equation using the distributive property
of multiplication over addition.  This expands (a + b) * (a + b) to a * a +
b * a + a * b + b * b.
Rewrite the term 2 * a * b in the goal as (a * b + a * b) using the theorem
that 2 times a number is the same as the number added to itself.  Also,
```

All strings in typewriter font are runtime placeholders. {`theorem_statement_NL`} – theorem in natural language; {`theorem_statement_FL`} – the same theorem in Lean's formal syntax; {`whole_theorems[theorem_name]`} – The staff solution; {`theorem_dict`} – dictionary of Peano-arithmetic facts available to the model; {`tactic_dict`} – dictionary of Lean tactics the model may use; {`prev_goal`} – current Lean proof state ; {`prev_nl`} – previous student proof lines ; {`nl_statement`} – the natural-language proof to be converted. The optional block, corresponding to the staff solution, renders optionally.

```
rewrite the term a * b + b * b as (a * b + a * b) + b * b using the theorem
that the product of a sum is the sum of the products.
We rewrite the expression a * b as b * a in the goal.  This is based on the
commutative property of multiplication, which states that the order of the
factors does not change the product.  This results in the new goal:  a * a +
a * b + (a * b + b * b) = a * a + (a * b + a * b) + b * b.
We use the theorem that states the associativity of addition twice to
rearrange the left-hand side of the equation.  This changes the goal to
proving that a * a + a * b + a * b + b * b equals a * a + a * b + a * b +
b * b.
The goal is now to prove that a * a + a * b + a * b + b * b = a * a + a * b
+ a * b + b * b, which is true by reflexivity
Output:  rw [pow_two, pow_two, pow_two]
rw [add_right_comm]
rw [mul_add, add_mul, add_mul]
rw [two_mul, add_mul]
rw [mul_comm b a]
rw [← add_assoc, ← add_assoc]
rfl
### User:  The natural language proof that we want to formalize:
{nl_statement}
```

A.10.2. Natural Language Feedback Generation

Natural language feedback generation prompt is the prompt to generate student feedback for incorrect proof inputs. This prompt is used in our final end-to-end system evaluation.

Natural language feedback generation prompt

```
### System:      You are a math professor, identifying the error in student
proofs, with the help of the Lean4 verifier.

### User:  A first-year math student's incomplete Peano Arithmetic proof has
been formalized in Lean4, but it has an error.
This is the incorrect student proof in Lean4:

{lean_proof}

This is the current Lean4 state, throwing an error due to the last step
last_line:

{error}

The actual correct step in Lean4 is:

{next_step}

Error Categories include:
1.  Inducting on the incorrect variable
2.  Selecting the incorrect base case
3.  Not generalizing the inductive step to all cases
4.  Failing to apply the inductive hypothesis
5.  Incorrect/Incomplete simplification or expansion
6.  Incorrect calculation or careless mistake
7.  Other

Explain the student error, ask a guiding question to reach correct next
step, and give a hint that explicitly reveals the answer in 1-2 sentences.
Be specific and use equations from goal states.

DO NOT USE any "Lean" or any Lean tactics or syntax such as "tactic" or
"reflexivity" or theorems such as "add_comm".  You are speaking directly
to the student, use "You" language.

Example:

Type:  Incorrect simplification
Message:  The RHS of your equation, a + (b + succ d), cannot be simplified
with your applied strategy.    Question/Hint:  Do you know of a theorem that
can perform this simplification?  Informalization:  The next step is to
rewrite a + (b + succ d) as (a + b) + succ d.

IMPORTANT: Respond with ONLY a raw JSON object in the following format,
without any code block formatting or additional text:
{
"Type":  "Students' error type",
"Message":  "Brief description of error in this problem"
"Question":  "Do you....?"
"Informalization":  "The next step is to..."
}
```

{lean_proof} is a placeholder for the autoformalized proof until now. {error} is the Lean compiler error thrown by the formalized proof. {next_step} is a placeholder for the next tactic generated by the NSG module.

A.10.3. Baseline Prompt for Full System Evaluation

Natural language error + next-step prompt is the baseline prompt used in end-to-end system evaluation. This prompt does not receive any Lean inputs.

Natural language error + next-step prompt

```
### System:     You are a math professor helping a student debug their Peano
Arithmetic proof.


### User:       A first-year math student is working on the following Peano
Arithmetic theorem:
{theorem}


Below are the steps of the proof the student has completed thus far.  There
may be errors and/or the work may be incomplete:
{proof}


Identify and explain the student error, if it exists.  Then, identify the
correct next step.  Ask a guiding question or give a hint that can help the
student reach the correct next step in 1-2 sentences.  Be specific.


Speak directly to the student using "You" language.  Avoid using Lean
tactics or syntax like "apply", "intro", or "rw".


Example:
Error Message:  The RHS of your equation, a + (b + succ d), cannot be
simplified with your applied strategy.
Next Step:  The next step is to rewrite a + (b + succ d) as (a + b) + succ
d.
Question/Hint:  Do you know of a theorem that can perform this
simplification?


IMPORTANT: Respond with ONLY a raw JSON object in the following format,
without any code block formatting or additional text:
{
"Error_Message":  "Brief description of error in this problem",
"Next_Step":  "The next step is to...",
"Question":  "Do you....?"
}
```

{theorem} is a runtime placeholder for the theorem statement (in NL). {proof} is a placeholder for the student's current attempt.