miniCodeProps: a Minimal Benchmark for Proving Code Properties

Evan Lohn
Carnegie Mellon University
`evanlohn@cmu.edu`

Sean Welleck
Carnegie Mellon University
`wellecks@cmu.edu`

Abstract

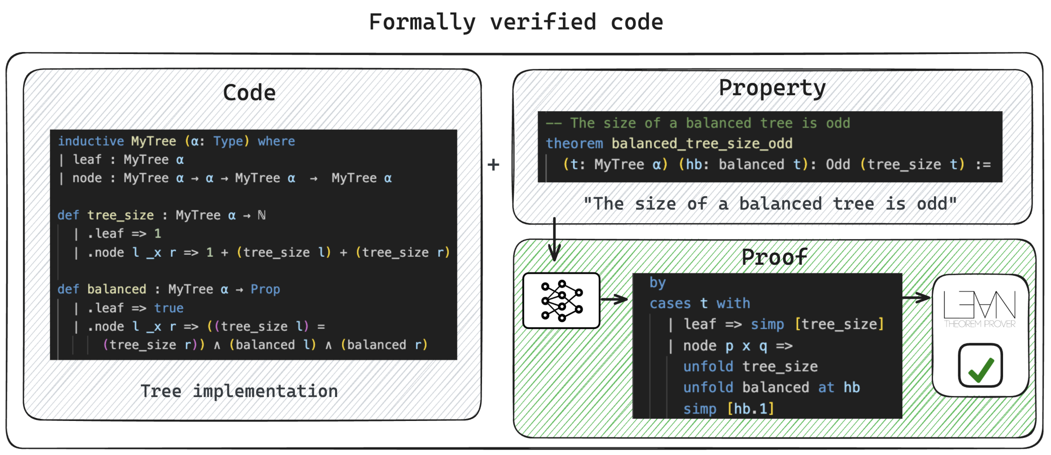
AI agents have shown initial promise in automating mathematical theorem proving in proof assistants such as Lean. The same proof assistants can be used to verify the correctness of code by pairing code with specifications and proofs that the specifications hold. Automating the writing of code, specifications, and proofs could lower the cost of verification, or, ambitiously, enable an AI agent to output safe, provably correct code. However, it remains unclear whether current neural theorem provers can automatically verify even relatively simple programs. We present `miniCodeProps`, a benchmark of 201 program specifications in the Lean proof assistant, aimed at the subproblem of automatically generating a proof for a provided program and specification. `miniCodeProps` contains specifications about simple, self-contained programs (e.g., lists, natural numbers, binary trees) with varied proof difficulty. Despite its simplicity, `miniCodeProps` is sufficient to break current LLM-based provers, with state-of-the-art methods showing promise on the easy properties in `miniCodeProps`, yet failing to prove nearly all of the medium and hard properties. We publicly release `miniCodeProps` as a benchmark for furthering automated theorem proving in the context of formally verified code.[1]

1 Introduction

Writing code that meets a specification is a desirable, yet difficult task. In safety-critical contexts, users require stronger safeguards than code reviews and test cases provide. To address this need, the formal methods community has produced a variety of tools for specifying and proving properties of code [1–5]. One approach uses interactive theorem provers (ITPs) such as Isabelle [6], Coq [7], and Lean [8]. In an ITP, a user writes code, a specification, and a proof that the specification holds (Figure 1). The ITP checks each step of the proof as it is written, and a complete proof means that the specification is guaranteed to hold. A complete proof thus gives guarantees on the underlying code, which is valuable in practice. For instance, the Lean proof assistant was recently used to verify properties of Amazon's Cedar Policy language [9]. Ambitiously, future code generation models could also output safe, provably correct code if they are able to write code, specifications, and proofs [10].

Despite its promise, interactive theorem proving remains a difficult task, and better automation–be it writing specifications, proofs, or combinations thereof–could help make it easier to verify code. Recently, machine learning techniques based on large language models (LLMs) have shown promise in automatically generating code [11], proving mathematical theorems in ITPs [12], and generating proofs of properties from verification projects [13]. However, evaluating the capabilities of LLMs for program verification remains a challenge. For example, benchmarks often evaluate across projects with complex dependencies such as the CompCert compiler or Archive of Formal Proofs [14, 13], making it difficult to isolate and reason about a model's core capabilities and weaknesses. Rather than

[1]Our benchmark and evaluation code can be found here: https://github.com/cmu-l3/minicodeprops-eval



```
inductive MyTree (α: Type) where
| leaf : MyTree α
| node : MyTree α → α → MyTree α  →  MyTree α

def tree_size : MyTree α → ℕ
  | .leaf => 1
  | .node l _x r => 1 + (tree_size l) + (tree_size r)

def balanced : MyTree α → Prop
  | .leaf => true
  | .node l _x r => ((tree_size l) =
      (tree_size r)) ∧ (balanced l) ∧ (balanced r)
```

```
-- The size of a balanced tree is odd
theorem balanced_tree_size_odd
  (t: MyTree α) (hb: balanced t): Odd (tree_size t) :=
```

```
by
cases t with
  | leaf => simp [tree_size]
  | node p x q =>
    unfold tree_size
    unfold balanced at hb
    simp [hb.1]
```

Figure 1: Formal verification of code in an interactive proof assistant (here, Lean) consists of writing (i) code (e.g., an implementation of a Tree), (ii) a property that you want to verify (e.g. that the size of a balanced tree is odd), and (iii) a proof that the code satisfies the property. The underlying language verifies that the proof is correct, providing certainty that the property holds. `miniCodeProps` focuses on the important subproblem of proof generation: given a property and associated code, a model must generate a proof (e.g., shown here in green) that the property holds. `miniCodeProps` contains 201 properties about lightweight, self-contained code blocks, and measures a range of proving abilities.

evaluating on complex projects, our goal is to find a minimal set of meaningful program properties that breaks current LLM-based provers, and thus requires new methods or models to solve.

To this end, we introduce `miniCodeProps`, a benchmark for evaluating the ability to prove properties of relatively simple, self-contained programs in the Lean proof assistant. `miniCodeProps` contains 201 specifications about simple, self-contained programs (involving e.g., lists, natural numbers, binary trees) with varied proof difficulty, sourced by translating Haskell programs from Tons of Inductive Programs [15] into Lean 4. Despite its simplicity, `miniCodeProps` is challenging for current LLM-based provers. For example, our best baseline approach based on GPT-4 proved very few specifications requiring proofs longer than a few lines. We publicly release `miniCodeProps` as a benchmark for furthering automated theorem proving in the context of formally verified code.

2 Related Work

Automating mathematical theorem proving. Recently there has been wide interest in automating mathematical theorem proving in interactive proof assistants; see [16, 12] for surveys. A typical approach [17] is to train on a large corpus of mathematical proofs such as Lean's Mathlib [18–22]. A model learns to call automation that is designed for mathematics, such as nonlinear arithmetic tactics, and to use definitions and theorems from within the corpus. It is unclear whether such methods transfer to the distribution of proofs encountered in program verification, which may rely on calling different automation, using program-specific definitions or lemmas, or different proof strategies. By focusing on Lean, `miniCodeProps` allows for testing this transferability. A second class of methods design prompting strategies with mathematical problems in mind, such as conditioning generation on an informal proof [23–25] or examples from Mathlib [26]. We hope that `miniCodeProps` motivates development of similar methods for program verification.

Automating formal code verification. There is a rich history of developing automation for formal verification of code; see [4] for a survey. For machine learning in interactive theorem proving, many methods that preceded large language models focused on Coq, such as ProverBot [27, 28], ASTactic [14], TacTok [29], Diva [30], and Passport [31]. Recent work with LLMs includes exploring prompting strategies in Coq [32], and Baldur [13] which explores proof generation and repair in Isabelle's Archive of Formal Proofs. We are not aware of machine learning-based proof automation targeting program verification in Lean, despite the activity in automated mathematical proving in

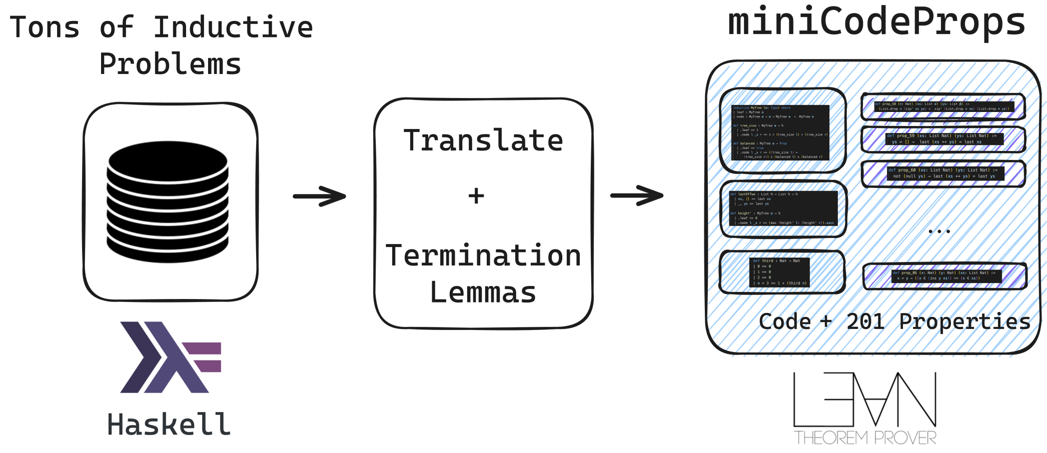


Figure 2: `miniCodeProps` is sourced from Tons of Inductive Problems [15], a collection of programs and specifications written in Haskell. We translate the programs into Lean, and write and prove termination lemmas that are needed to state and prove properties of recursive functions.

Lean. One of our motivations is bridging this gap. Finally, some methods such as COPRA [25] aim to be language-agnostic; `miniCodeProps` could participate in evaluating and developing such systems.

A second paradigm of verification is based on automated reasoning (SAT/SMT-based) languages such as Dafny [1] and Verus [5] (Rust). Recent work explores LLM-based automation in these languages, such as Clover [10] for Dafny, and [33] for Verus. These are complementary to studying automated ITP verification. Finally, Tons of Inductive Problems [15]–from which we derive our data–targets automated reasoning-based verification, while `miniCodeProps` targets interactive theorem proving.

Interactive theorem proving benchmarks. The CoqGym [14] benchmark tests on over 100 repositories in Coq, while some papers use code properties from a single large, complex repository such as the CompCert compiler verification project in Coq [34, 25], which arguably tests different aspects of automated code verification than those tested by `miniCodeProps`. In Isabelle, the Archive of Formal Proofs contains some verification-related sections that are used for evaluation, e.g., in Baldur [13]. Benchmarks for automated theorem proving in Lean are focused on theorems from mathematical domains. For example, miniF2F [35] contains 488 self-contained, easy-to-state theorems from math competitions, while ProofNet [36] contains self-contained textbook problems. miniF2F's simplicity and impact as a benchmark motivated the creation (and naming) of `miniCodeProps`.

3 Benchmark Contents

We create `miniCodeProps` by translating programs from Tons of Inductive Programs (TIP) [15] (https://github.com/tip-org/benchmarks) from Haskell into Lean 4. We manually translated code from three files in TIP that contain a mix of function definitions, propositions describing the results of calling those functions, and termination lemmas used in the function definitions. In total, this yields 201 Lean 4 theorem statements. We specifically make `miniCodeProps` fairly small, since evaluating theorem proving models requires many calls to the language model (e.g., in a tree search or through repeated sampling), and is hence time-consuming and prohibitively expensive for large datasets.

3.1 Collection Process

The properties in TIP describe induction in both programming and purely mathematical contexts. Since our focus is programming, we filtered out files that contained primarily mathematical induction, then translated the remaining Haskell code into Lean 4. We associated each property in the translated files with dependencies and metadata useful for different evaluation modes of the benchmark.

	Numbers	**Lists**	**Trees**	**Heaps**	**Difficulty**
Medley Functions	0	26	10	0	–
Medley Properties	21	84	5	0	Easy
Termination+Sorting Functions	3	49	2	5	–
Termination Properties	3	22	0	3	Medium
Sorting Properties	0	63	0	0	Hard

Table 1: **miniCodeProps Composition** We classified each function and property in the Lean source files by their primary subject matter. We also separate these numbers by the splits of `miniCodeProps`: Medley, Termination lemmas, and Sorting algorithms. Termination and Sorting algorithm functions are grouped because the properties in both splits describe the same set of functions. The properties are classified into easy (Medley), medium (Termination), and hard (Sorting) difficulty levels.

3.2 Source Code Composition

`miniCodeProps` contains properties that we view as a "minimum level of competency" for automated neural theorem provers. We draw an analogy to an expert software engineer that spends time writing code in a complex repository, yet is expected to be capable of solving self-contained interview problems. `miniCodeProps` embodies the latter, which is absent from existing theorem proving benchmarks. Namely, rather than pulling code and properties from complex verification projects, `miniCodeProps` tests key capabilities with lightweight, self-contained code blocks and properties.

These fall into three categories: (1) intuitive properties of lists, trees, and natural numbers (*medley*), (2) lemmas that support termination proofs, which are an essential (and tedious) part of verification in an interactive theorem prover (*termination*); (3) properties of nonstandard sorting algorithms that require a deep understanding of the corresponding code (*sorting*). We provide more detail below.

Functions. Most of the translated functions operate on linked lists, while the rest involve natural numbers and binary trees. With a few notable exceptions, the functions perform conceptually simple operations such as filtering, returning the last element, and counting the elements of a list. The more complicated functions are increasingly esoteric sorting functions. In total, `miniCodeProps` is derived from 95 TIP functions. A breakdown is provided in Table 1.

Properties. The properties to be proven in `miniCodeProps` express intuitive properties of the function(s) being described. Properties in the *medley* classification typically require a standard induction argument along with the application of 1-2 library theorems. Properties of the nonstandard sorting algorithms are easy to state, yet are likely much more difficult prove due to the complexity of the sorting functions. See Section 3.3 for examples, and Table 1 for a breakdown by subject type.

Termination Lemmas. In Lean 4, recursive functions must be paired with a proof of termination. While Lean automatically infers the proof in simple cases, more complicated recursive calls require the user to explicitly prove termination. As the vast majority of our function definitions are recursive, our benchmark includes 28 lemmas that support 4 termination proofs of nonstandard sorting algorithms from TIP. These lemmas represent highly nontrivial properties of code that, roughly speaking, may take hours of human effort to prove. Automating these proofs would be highly desirable in practice.

Dataset format. The benchmark is formatted as a jsonlines file with the information in Table 2 per line. The "full_name" property is used to determine the split the property belongs to: *medley* properties are named "prop_n" for $n \in [1, 86]$, *sorting* algorithm properties are those remaining theorems named beginning with "prop", and the remainder are *termination* lemma properties.

3.3 Qualitative Examples

We study several benchmark examples and describe what makes them interesting and difficult.

Zip over concatenation. We begin with a reimplementation of the standard 'zip' function that combines two lists into a single list of paired elements. 'prop_84' states that zipping a list xs with the concatenation of two other lists ys, zs is equivalent to separately zipping appropriate sections of

Table 2: Example code property and associated data

Key	Description	Examples
full_name	Unique name of the property in the benchmark	'prop_60'

prop_defn	The text of the code property to be proven	`lemma prop_60 (xs: List Nat)`
		`(ys: List Nat) : not (null ys)`
		`→ last (xs ++ ys) = last ys:=`
		`by sorry`

prop_loc	The file name and line number where the property definition begins	LeanSrc/LeanSrc/Properties.lean:257
score	An integer in the range [1-5] used as a difficulty heuristic	5

deps	Text containing all the dependencies (functions and lemmas) that the property requires to be fully defined	`import Mathlib`
		`def last: List Nat → Nat`
		`  | [] => 0`
		`  | [x] => x`
		`  | _x::xs => (last xs)`
		`def null : List α → Bool`
		`  | [] => True`
		`  | _ => False`

proof_state	The initial proof state of the property	`xs ys : List ℕ`
		`⊢ (!null ys) = true →`
		`last (xs ++ ys) = last ys`

file_locs	A list of the last line in each file needed to define the property. Appending each file up to this line before the property fully defines the property.	[["LeanSrc/LeanSrc/Definitions.lean", 249], ["LeanSrc/LeanSrc/Properties.lean", 258]]

xs with each of ys and zs, then concatenating the results. To the best of our knowledge, Mathlib does not contain an equivalent lemma for the standard definition of the zip function.

```
def zip' : List α → List β → List (α × β)
  | [], _ => []
  | _, [] => []
  | x::xs, y::ys => ⟨x, y⟩ :: zip' xs ys

lemma prop_84 (xs: List α) (ys: List β) (zs: List β) :
  (zip' xs (ys ++ zs) =
  zip' (List.take (List.length ys) xs) ys ++
  zip' (List.drop (List.length ys) xs) zs):= by
```

This property is intuitively correct, but requires some intuition about proofs of recursive structures to correctly prove in Lean. The following is a correct proof of prop_84 generated by GPT4. There are several unnecessary definitions passed to the calls to the simplifier (the 'simp' tactic). However, GPT4 demonstrates the main ideas of the proof: induct on ys, pair off the first elements of xs and ys, then apply the inductive hypothesis to what remains.

```
induction ys generalizing xs with
  | nil => simp [zip', List.take, List.drop]
  | cons y ys ih =>
```

```
    cases xs with
    | nil => simp [zip', List.take, List.drop]
    | cons x xs =>
      simp [List.cons_append, zip', List.take,
       List.drop, List.length]
      rw [ih]
```

Many of the *medley* properties test similar abilities, which we view as bare minimum capabilities for a system aimed at automating theorem proving in the context of formally verified code.

Non-standard insertion increments length. Although the zip concatenation property is not part of Mathlib, one can argue that because the property involves a utility function defined in Mathlib with its own associated properties and proofs, it is not altogether surprising that GPT4 was able to infer a proof of a novel property of a re-definition of the function. We therefore present a property of a function 'ins' not included in Mathlib that inserts an element into a list before the first element it is less than. Proving this property requires the ability to reason about previously unseen code.

```
def ins: Nat → List Nat → List Nat
  | n, []  =>  [n]
  | n, x::xs => if (n < x) then n :: x :: xs else x :: (ins n xs)
```

The following is a correct proof generated by GPT-4. This proof is remarkable for its simplicity. Unlike the previous correct proof there is little wasted space, a feature often sought after by human proof experts. GPT4 demonstrates understanding of how to work with the 'ins' function via the 'split_ifs' tactic, and the usage of the inductive hypothesis only in the second branch where it is necessary.

```
induction xs with
| nil => simp [ins]
| cons y ys ih =>
  simp [ins]
  split_ifs with h
  { simp }
  { simp [ih] }
```

Example sorting properties. The Lean source code of `miniCodeProps` contains implementations of 9 sorting algorithms, with associated properties to prove for each algorithm. We display several of the functions and lemmas associated with one of our two heapsort implementations (each uses a different strategy for merging heaps). The 'toList' function contains an example of a termination lemma: the line beginning with 'have _h' references a lemma that is used by Lean to prove that the recursion in 'toList' terminates, a property required by default for recursive functions in Lean. Several of the functions and lemmas referenced are omitted from the code for brevity.

```
def toHeap : List Nat → MyHeap
| xs => hmerging (xs.map (fun x => MyHeap.node MyHeap.nil x MyHeap.nil))

lemma numElem_merge_branches_lt (p q: MyHeap) (x: Nat): numElem (hmerge p q) <
    numElem (MyHeap.node p x q) := by
  rw [←merge_elems _ _];
  have h': numElem (MyHeap.node p x q) = 1 + numElem p + numElem q; rfl
  rw [h']
  linarith;

def toList : MyHeap → List Nat
| MyHeap.nil => []
| MyHeap.node p x q =>
    have _h := numElem_merge_branches_lt p q x
    x :: toList (hmerge p q)
termination_by hp => numElem hp

def hsort : List Nat → List Nat
 | xs => toList (toHeap xs)
```

```
theorem prop_HSortSorts (xs: List Nat) : ordered (hsort xs) == True := by sorry
theorem prop_HSortCount (x: Nat) (xs: List Nat) : count x (hsort xs) == count x xs
    := by sorry
theorem prop_HSortPermutes (xs: List Nat) : isPermutation (hsort xs) xs == True :=
    by sorry
theorem prop_HSortIsSort (xs: List Nat) : hsort xs == isort xs := by sorry
```

The properties above express the following intuitive properties: heapsort orders list elements, preserves the list size, returns a permutation of the original list, and returns the same result as insertion sort. These properties are repeated for each sorting algorithm. However, given the complexity of the sorting functions involved, we expect that proving such properties will be very difficult. An example of an incorrect proof of one of the sorting properties produced by GPT4 can be found in Appendix A.

Finally, `miniCodeProps` includes termination lemmas such as ‘numElem_merge_branches_lt’ with the human-produced proof removed. Some of these lemmas took hours of human effort to prove. Since proving termination is an essential part of human interactive theorem proving, it is highly likely that improved performance on these lemmas could translate into useful automation for practitioners.

Availability and usage. We have published our benchmark as a public dataset on Huggingface, and released a repository for evaluating language models on `miniCodeProps`.

4 Baselines

We test two standard proving modes: full-proof generation and tactic-by-tactic generation. Since the underlying code block for a property is typically necessary to complete the proofs, we always provide the models with the code dependencies (stored as “deps” in our dataset, see Table 2).

For full-proof generation, the model generates one or more potential proofs that are checked by the proof checker (in our case, the Lean 4 kernel). For tactic-by-tactic generation, we follow the common practice of taking the proof state as input and returning suggestions for the next tactic to use in the proof. Each suggestion is then given as input to the Lean 4 kernel, and the resulting proof state is used to prompt the model for the next tactic. We test the *de-facto* standard algorithm for tactic generation, best-first search [17, 19, 22, 26]. Our publicly available evaluation code supports both modes.

Tactic-by-tactic generation. We use `ntp-ctx-1.3B` [37] since at the time of experimentation it is the only tactic generation model specifically trained to accept both the current proof state and an additional input context (in our case, the necessary underlying code for the property being proven). The `ntp-ctx-1.3B` model is a DeepSeek-Coder 1.3B language model fine-tuned on (context, proof state, next tactic) examples derived from Mathlib [37]. For the context, following a similar setup to [37] we provide a context containing the dependent code (“deps” in Table 2), the property being proven, and the history of generated tactics. We use the same input format and search settings as [37].

Full proof generation. We use GPT-4o, since at the time of experimentation it is a highly-performant general-purpose language model on mathematical and code tasks, and its model variants (e.g., previous models of GPT-4) are common baselines in neural theorem proving (e.g., [25, 37]). We provide a context containing the dependent code (“deps” in Table 2) and the property being proven. We measure pass@32, meaning that we generate 32 proof candidates (with default generation parameters), and consider a property to be successful when at least one of them passes the verifier.

Full proof generation with refinement. As a strictly stronger full proof generation baseline, we perform 2 rounds of an experimental “many-to-one” refinement technique using GPT-4o. Specifically, suppose that none of the 32 candidate proofs $y^{(1)}, \ldots, y^{(32)}$ passed the verifier. We provide all 32 candidate proofs in a prompt, and ask the model to learn from the failed attempts and generate a new proof. We sample k such generations, which constitutes one round of refinement. We do this for T rounds, feeding in only the previous round’s generations. We use $k = 16$ and $T = 2$. In a small scale experiment we confirmed that this outperformed sampling 64 candidates without refinement.

5 Results and Discussion

Results. Our baseline results are displayed in Table 3. All three baselines show nontrivial performance on the easy (*medley*) properties. This shows that current neural theorem provers have some

Model	**Medley** (Easy)	**Termination + Sorting** (Medium + Hard)	**Overall** (All)
GPT-4o (pass@32)	75.6% (65/86)	4.34% (5/115)	34.8% (70/201)
+ refinement	77.9% (67/86)	6.96% (8/115)	37.3% (75/201)
`ntp-ctx-1.3B` [37]	72.1% (62/86)	8.69% (10/115)	35.8% (72/201)

Table 3: **miniCodeProps results.** Number of specifications proven using full proof generation with GPT4 and best-first tactic generation with `ntp-ctx-1.3B` to the problem of verifying program specifications. The Medley section contains mostly of specifications that can be proven in several lines. Proofs of the sorting algorithm properties and termination lemmas are expected to require at least tens of lines and hours of programmer effort.

ability to perform basic inductive proofs in a program verification setting. However, the three methods perform very poorly on the medium and hard properties. The results show that `miniCodeProps` is sufficient to break current neural theorem proving methods, hence motivating the need for further research. Moreover, it is unlikely that these high-level conclusions would change given more properties drawn from the same distribution, thereby justifying `miniCodeProps`' size of 201 properties.

However, it is promising that changing the generation algorithm (namely, to refinement) leads to some additional successes on the medium and hard split. This suggests that further algorithmic development could lead to additional gains. Furthermore, the small `ntp-ctx-1.3B` model trained on Mathlib shows comparable, or better, performance than GPT-4o. As a result, we view training specialized models for verification-style proofs as a promising direction as well (though requiring nontrivial research due to the scarcity of training data). Based on these observations, we speculate that neural theorem provers have the potential to improve on `miniCodeProps` in the near future.

Scope. We chose to source `miniCodeProps` from the TIP dataset to provide some assurance that the code and properties involved were of interest to the broader code verification community. Clearly, performing well on this subset is only a proxy for the ability to perform well on all realistic code properties. However, we designed `miniCodeProps` to target a minimal set of code properties that we would expect a reasonable automated system to be able to prove. This is similar to an expert software engineer that spends time working in large, complex codebases, yet is expected to be capable of solving self-contained exercises that target core programming skills.

Societal context. The broader goal of our work is to facilitate the development of automated (likely, neural) theorem proving agents that can prove properties of code in ITPs. Verification engineers already draw from a wide variety of automated tools in their work. An automated version of ITP still requires a human in the loop to generate (or validate) the statement of the property to be proven, similar to the type of work these engineers perform while using SMT-based verification. Arguably, the main benefit to society of the ITP approach is that it induces our automated tools to use different reasoning patterns than the ones commonly used in Automated Reasoning (SAT/SMT) approaches, which may in turn lead to different subsets of formal guarantees that can be obtained in practice.

6 Conclusion

`miniCodeProps` is intended as a meaningful benchmark for evaluating techniques that automate ITP-based code verification, including for safe generation of code by AI agents. It contains a diverse array of functions and properties from Tons of Inductive Problems, allowing the possibility of incremental progress in verifying the types of programs contained in TIP. We show that simple baseline approaches from mathematical neural theorem proving currently fall short on much of our benchmark, and hope that `miniCodeProps` spurs development of future ITP code verification agents.

Acknowledgements

Sean Welleck thanks Convergent Research, the Lean FRO, and the OpenAI Researcher Access Program for their support.

References

[1] Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference, LPAR-16, Dakar, Senegal*, pages 348–370. Springer Berlin Heidelberg, April 2010.

[2] Rustan Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *Conference: Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010.*, March 2010.

[3] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in f*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 256–270, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2–3):102–281, 2019.

[5] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types (extended version), 2023.

[6] Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow. The isabelle framework. In *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings 21*, pages 33–38. Springer, 2008.

[7] The Coq Development Team. The coq proof assistant, oct 2019.

[8] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 378–388. Springer, 2015.

[9] Lean into Verified Software Development. `https://aws.amazon.com/blogs/opensource/lean-into-verified-software-development/`, 2024.

[10] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation, 2024.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[12] Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. A survey on deep learning for theorem proving, 2024.

[13] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models, 2023.

[14] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. *ArXiv*, abs/1905.09381, 2019.

[15] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Tip: Tons of inductive problems. In *TIP: Tons of Inductive Problems*, volume 9150, 07 2015.

[16] Pan Lu, Liang Qiu, Wenhao Yu, Sean Welleck, and Kai-Wei Chang. A survey of deep learning for mathematical reasoning. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14605–14631, Toronto, Canada, July 2023. Association for Computational Linguistics.

[17] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

[18] mathlib. The lean mathematical library. In *CPP 2020 - Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, co-located with POPL 2020*, 2020.

[19] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. In *International Conference on Learning Representations*, 2022.

[20] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning, 2022.

[21] Guillaume Lample, Timothee Lacroix, Marie anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[22] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS)*, 2023.

[23] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023.

[24] Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. LEGO-prover: Neural theorem proving with growing libraries. In *The Twelfth International Conference on Learning Representations*, 2024.

[25] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving, 2024.

[26] Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen Marcus McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics. In *The Twelfth International Conference on Learning Representations*, 2024.

[27] Joseph Redmon and Alex Sanchez-Stern. Proverbot 9000 : Neural networks for proof assistance, 2016.

[28] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating correctness proofs with neural networks, 2020.

[29] Emily First, Yuriy Brun, and Arjun Guha. Tactok: semantics-aware proof synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[30] Emily First and Yuriy Brun. Diversity-driven automated formal verification. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 749–761, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. Passport: Improving automated formal verification using identifiers. *ACM Trans. Program. Lang. Syst.*, 45(2), jun 2023.

[32] Shizhuo Dylan Zhang, Talia Ringer, and Emily First. Getting more out of large language models for proofs, 2023.

[33] Jianan Yao, Ziqiao Zhou, Weiteng Chen, and Weidong Cui. Leveraging large language models for automated proof synthesis in rust, 2023.

[34] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[35] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2022.

[36] Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics, 2023.

[37] Jiewen Hu, Thomas Zhu, and Sean Welleck. minictx: Neural theorem proving with (long-)contexts, 2024.

A Appendix

This is an incorrect proof of the 'heap sort orders list elements' property. GPT4 leaves in "placeholders" as arguments to lemmas from the environment that it does not use in the proof. It also attempts the invalid operation of induction generalizing the inductive type 'MyHeap'. Transforming this proof into a correct one is non-trivial.

```
unfold hsort
unfold ordered
induction xs generalizing MyHeap with
| nil => unfold toHeap; unfold toList; simp
| cons x xs' ih =>
  unfold toHeap
  unfold toList
  unfold hmerge
  have h := numElem_merge_branches_lt ... -- Placeholder for specific heaps
  have m := merge_elems ... -- Placeholder for specific heaps
  specialize ih (xs'.map (fun x => MyHeap.node MyHeap.nil x MyHeap.nil))
  induction (toList (toHeap xs')) generalizing (toList (toHeap (x :: xs')))
  | nil =>
    apply ih
  | cons y ys' ysih =>
    unfold toList at ih
    unfold toHeap at ih
    unfold ordered at ih ⊢
    unfold hsort at ih
    have h := numElem_merge_branches_lt ... -- Placeholder for specific values
    have m := merge_elems ... -- Placeholder for specific values
    case h_1 =>
      unfold ordered
      simp [List.map]
      simp [toList]
    case h_2 =>
      unfold toHeap
      unfold ordered
      simp [toList]
      rfl
```