`miniCodeProps`: a Minimal Benchmark for Proving Properties of Code

Evan Lohn[1] and Sean Welleck[1]

Carnegie Mellon University, Pittsburgh, Pennsylvania, U.S.A
(elohn,wellecks)@cmu.edu

1 Introduction

Large language models (LLMs) have led to rapid progress in automated code generation. However, LLM-generated code lacks correctness guarantees, limiting its safety and reliability. Interactive theorem provers (ITPs) offer guarantees on the correctness of code (either generated or human-written) by pairing code with formal specifications of desired behavior, along with proofs that the code meets the specifications. This makes the intersection of LLMs and ITPs potentially fruitful for dramatically improving the safety and reliability of LLM-generated code. However, writing verified code often requires a deep understanding of a program's semantics and extreme effort, even for human experts. As a result, it is unclear to what extent LLMs are capable of automatically generating verified code using an ITP, even in fairly simple cases.

Towards this end, we develop `miniCodeProps`, a benchmark of 177 program specifications in the Lean proof assistant, aimed at the subproblem of automatically generating a proof for a provided program and specification. Despite its simplicity, `miniCodeProps` is challenging for current LLM-based provers. For example, our best baseline approach 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.[1]

2 Benchmark Contents

We create `miniCodeProps` by translating programs from Tons of Inductive Programs (TIP) [1] (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 177 Lean 4 theorem statements.

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 76 TIP functions.

Propositions The propositions express intuitively correct properties of the function(s) being described. One example property is the formalized version of the following: "if a list l is non-empty, the concatenation of all but the last element with the last element yields the original list." The formalized version is as follows:

```
def prop_48 (xs: List Nat) := not (null xs) -> butlast xs ++ [last xs] = xs
```

[1]Our benchmark and associated baseline code can be found here

	Medley (Easy)	Termination (Med.)	Sorting (Hard)
LLMStep + Pythia2.8B	44/86	1/28	0/63
LLMStep + Llemma7B	46/86	2/28	0/63
LLMStep + ntp-mathlib-context-deepseek-coder1.3B	38/86	0/28	0/63
GPT-4-turbo	44/86	1/28	9/63

Table 1: Number of specifications proven when applying next-step tactic generation with LLMStep and full proof generation with GPT4 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.

Termination Lemmas In Lean 4, recursive functions must be paired with a proof of termination. While Lean infers the proof without user input 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 themselves represent practically useful and highly nontrivial properties of code.

Availability and Usage We have published our benchmark as a public dataset on Huggingface, including a link to our example benchmarking code. The Huggingface dataset is in the jsonlines format. Each entry contains the relevant programs, the specification, the initial proof state of the specification, a subjective difficulty score, and the location of the specification in the source file. The source files are also included. This format is intended to make it easy to experiment with various input strategies, such as providing a model with a proof state, a full file, or a filtered set of code blocks as context.

Related work. Many benchmarks for automated theorem proving in Lean are focused on theorems from mathematical domains. For example, miniF2F [4] contains 488 self-contained, easy-to-state theorems from math competitions. Its simplicity and impact as a benchmark motivated the creation of `miniCodeProps`. Other benchmarks provide code properties from large, complex repositories such as the CompCert compiler verification project in Coq [2, 3], which arguably tests different aspects of automated code verification than those tested by `miniCodeProps`. Finally, Tons of Inductive Problems [1] targets SMT-based verification, while `miniCodeProps` targets interactive theorem proving.

3 Baselines

Recent work on neural theorem proving uses two main approaches: Full proof generation via few-shot prompting a language model and next-step tactic generation. Few-shot prompts typically contain initial proof state and file context paired with complete proofs. After the model generates one or more potential proofs, they are verified by the proof checker (in our case, the Lean 4 kernel). Next-step tactic prediction models commonly take only the proof state as input and return 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 use the common "best-first" heuristic to choose proof states to prompt the language model. Our baseline results are displayed in Table 1. In our talk, we will discuss our results qualitatively and quantitatively. Our Huggingface repo contains full details of our experiment setup.

References

[1] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Tip: Tons of inductive problems. volume 9150, 07 2015.

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

[3] A. Thakur, G. Tsoukalas, Y. Wen, J. Xin, and S. Chaudhuri. An in-context learning agent for formal theorem-proving, 2024.

[4] K. Zheng, J. M. Han, and S. Polu. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2022.