CLEVER: A Curated Benchmark for Formally Verified Code Generation

Amitayush Thakur[1], Jasper Lee[1], George Tsoukalas[1], Meghana Sistla[1], Matthew Zhao[1], Stefan Zetzsche[2], Greg Durrett[1], Yisong Yue[3], Swarat Chaudhuri[1]
[1]The University of Texas at Austin [2]Amazon [3]Caltech

Leaderboard

Supported by NSF Awards CCF-2212559, CCF-2403211, NSF IFML, and Renaissance Philanthropy.

Key Idea

End-to-End verified code (provably correct code) generation from Natural Language (NL) description requires generation of formal specification from the NL description and proof of correctness of the generated code in an Interactive Theorem Proving (ITP) language like Lean 4. The challenge is not just writing the proof of correctness in a formal language but coming up with the semantically correct formal specification (aligned with the NL description). This can be addressed using multi-phased correctness certification where not just the correctness of implementation matters but also the correctness of the formalization itself.

Intro: Formal Verification

- Formal verification involves writing formal specification in languages like Lean 4. See the example below:

```
-- Formal spec for checking correctness of an
implementation for sorting a list
theorem spec_list_is_sorted
-- list of natural numbers & implementation to sort it
(lst : List Nat) (sort_func : List Nat → List Nat)
:let sorted_lst := sort_func lst; -- call the sort
function
-- First order condition for checking if the list is
sorted
∀ i j, i < j ∧ j < lst.length → lst[i]! ≤ lst[j]! :=
```

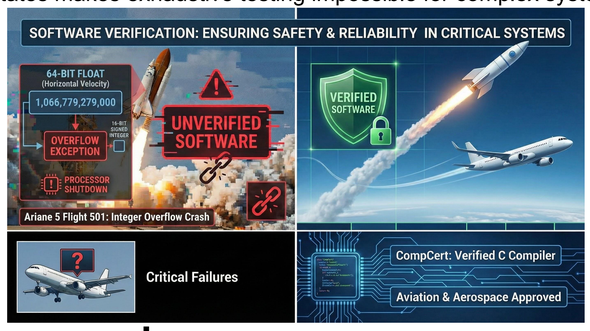
- Once we describe the formal meaning of correctness for the code being generated. We can then write proofs about correctness of the implemented function. In fact, we can write proof about any formal statement (including the ones about correctness of programs). Example of simple proof for mathematical fact:

```
-- If a number is even then so is its square.
theorem mod_arith_2 (x : Nat): x % 2 = 0 → (x * x)
% 2 = 0:=
 intro h
 rw [Nat.mul_mod]
 rw [h] -- Proof checked mechanically via Lean
compiler
```

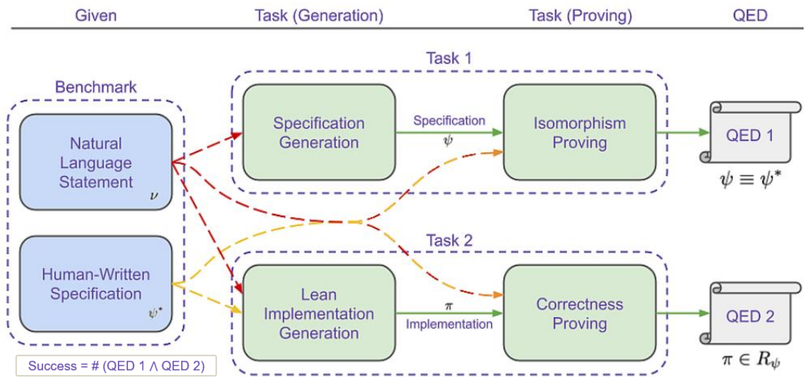
- In terms of software verification, we can think of specification as a combination pre and post condition for an implementation of a function.

Why proof of correctness matters?

- **The Compiler Trust Gap:** Formally verified compilers like CompCert [1] eliminate the risk of compiler-introduced bugs by proving mathematically that the generated assembly code works exactly as it is expected to.
- **The Limits of Testing:** Because the astronomical number of possible input states makes exhaustive testing impossible for complex systems.



Approach



- **Dataset Origin:** The benchmark adapts 161 problems from HumanEval[2] (originally Python coding tasks) into Lean 4, transforming them from simple coding tests into rigorous formal verification challenges.
- **Task 1 (Specification):** The model must translate the Natural Language statement (v) into a formal specification (ψ) and prove it is mathematically equivalent (isomorphic) to the hidden human-written ground truth (ψ*).
- **Task 2 (Implementation):** The model must generate the actual functional code (π) and prove that it satisfies the specification, guaranteeing correctness without relying on test cases.
- **Strict "QED" Success:** Unlike standard benchmarks that check if code "runs," CLEVER considers a solution correct only if the model produces valid formal proofs for both the specification alignment and the implementation correctness.

Leaderboard

Model	Approach	End-to-End Code Generation	Note	Generated
Claude-3.7	COPRA-enhanced	1/161	Problem 53	2/161 (spec) 14/161 (impl = 16/282
DeepSeek-R1	Few-Shot	1/161	Problem 53	1/161 (spec) 9/161 (impl) 10/282
GPT OSS 20b	COPRA-enhanced	1/161	Problem 53	2/161 (spec) 8/161 (impl) 10/282
GPT-4o	COPRA-enhanced	1/161	Problem 53	3/161 (spec) 6/161 (impl) 9/282
GPT-4o mini	Few-Shot	1/161	Problem 53	2/161 (spec) 3 / 161 (imp = 5/282
Claude-3.7	Few-Shot	1/161	Problem 53	1/161 (spec) 3/161 (impl) 4/282
GPT-4o	Few-Shot	0/161	-	1/161 (spec) 1/161 (impl) 2/282
GPT-5 mini (For Code Generation) + Kimina Prover (For proofs)	Few-Shot	0/161	-	0/161 (spec) 1/161 (impl) 1/282

Model	Approach	Spec Certification	Impl Certification
		Compiled	Compiled
Few-Shot Baseline			
GPT-4o mini	Few-Shot	82.609%	83.230%
Claude-3.7	Few-Shot	86.957%	65.217%
GPT-4o	Few-Shot	84.472%	68.323%
DeepSeek-R1	Few-Shot	71.42%	60.870%
COPRA Baseline			
Claude-3.7	COPRA-enhanced	81.366%	65.217%
GPT OSS 20b	COPRA-enhanced	78.261%	65.839%
GPT-4o	COPRA-enhanced	76.398%	68.323%

- We evaluated both few shot invocation of models and using an theorem proving agent like COPRA [3]

Future directions

- **(Natural Language ↔ Formal Specification ↔ Proof) Mining:** "Informal-to-Formal" Data: To overcome data scarcity, future work must mine and align massive datasets of natural language descriptions (like docstrings) paired with formal specifications, training models to better "translate" human intent into logic.
- **Neuro-Symbolic "AlphaProof" Systems:** Moving beyond simple text generation to hybrid systems where the LLM acts as an "intuitive" guide for a rigorous "logical" theorem prover (similar to DeepMind's AlphaProof), solving the problem of hallucinated proofs.

References

[1] Thakur, Amitayush, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. "An In-Context Learning Agent for Formal Theorem-Proving." In First Conference on Language Modeling.
[2] Chen, Mark, et al. "Evaluating Large Language Models Trained on Code." arXiv preprint arXiv:2107.03374 (2021).
[3] Leroy, Xavier. "Formal Verification of a Realistic Compiler." Communications of the ACM 52, no. 7 (2009): 107–115.