CLEVER: A Curated Benchmark for Formally Verified Code Generation

Anonymous Authors[1]

Abstract

We introduce CLEVER[2], a high-quality, curated benchmark of 161 problems for end-to-end verified code generation in Lean. Each problem consists of (1) the task of generating a specification that matches a held-out ground-truth specification, and (2) the task of generating a Lean implementation that provably satisfies this specification. Unlike prior benchmarks, CLEVER avoids test-case supervision, LLM-generated annotations, and specifications that leak implementation logic or allow vacuous solutions. All outputs are verified post-hoc using Lean's type checker to ensure machine-checkable correctness. We use CLEVER to evaluate several few-shot and agentic approaches based on state-of-the-art language models. These methods all struggle to achieve full verification, establishing it as a challenging frontier benchmark for program synthesis and formal reasoning. Our benchmark can be found on Anonymized Repository. All our evaluation code is also available online.

1. Introduction

Interactive theorem-provers (ITPs) (Huet et al., 1997; Paulson, 1994; de Moura et al., 2015) are an established technology for engineering high-assurance software, leading to success stories like the CompCert verified C compiler (Leroy, 2009b) and the seL4 (Klein et al., 2009b) verified microkernel. However, writing formal specifications and correctness proofs for software systems can take tremendous effort — for example, the development of seL4 was reported to take 20+ person-years. These costs are a key impediment to the broad deployment of ITP-based formal verification.

Recent progress in autoformalization and neural theorem-proving (Polu & Sutskever, 2020; Li et al., 2024) has raised hopes of scaling up formal verification (Yang et al., 2024). Most existing work in this area has focused on formalizing and proving statements in pure mathematics (Zheng et al., 2021; Tsoukalas et al., 2024). However, the software verification setting opens up the challenge of *generating code that is formally verified by construction*, a problem without a well-studied analog in the mathematics setting.

To date, there are a handful of benchmarks (Dougherty & Mehta, 2025; Loughridge et al., 2024; Lohn & Welleck, 2024) for formally verified code generation. However, the formal specifications in these benchmarks tend not to capture the full (natural-language) intent behind the target program and sometimes hint at ways to implement the program. This ambiguity allows a code generator to "cheat" by generating trivial programs or copying code from the specification (see Appendix A.1).

In this paper, we address this gap in the prior art with CLEVER, a high-quality benchmark for formally verified AI-based code generation. CLEVER includes hand-crafted Lean specifications of 161 programming tasks from the HUMANEVAL benchmark (Chen et al., 2021).

It evaluates models in two stages: (1) *Specification certification:* Given a natural language specification, the model is required to generate a Lean specification and prove that it is semantically equivalent to the ground-truth specification. (2) *Implementation certification:* Once the model has correctly generated the specification, it is required to generate a Lean implementation and prove that it satisfies the ground-truth specification. A synthesis attempt is deemed *successful* only when both the proofs generated in the two stages are fully verified by Lean's type checker. This rigorous pipeline avoids the pitfalls of both automatically generated specifications and test-based supervision.

We use CLEVER to evaluate several state-of-the-art LLMs prompted in a few-shot manner and show that they can only solve up to 1/161 end-to-end verified code generation problem, establishing CLEVER as a challenging frontier benchmark for program synthesis and formal reasoning. In summary, our contributions include:

1. We introduce CLEVER, the first curated benchmark for evaluating the generation of specifications and formally verified code in Lean. The benchmark comprises of 161 programming problems; it evaluates both *formal specification generation* and *implementation synthesis* from natural language, requiring formal correctness proofs

[2]CLEVER: Curated Lean Verified Code Generation Benchmark

for both. All specifications are manually written to be complete, implementation-agnostic, and free from exploitable artifacts, preventing models from shortcutting the intended semantics.

2. We present an empirical evaluation of several state-of-the-art LLMs and agentic approaches on CLEVER and show that they all struggle at meeting the benchmark's goals, establishing the challenging nature of the benchmark.

2. The CLEVER Benchmark

CLEVER builds on HUMANEVAL (Chen et al., 2021) by adapting 161[3] of its 164 programming problems for formal verification in Lean 4. Each problem includes a natural language description (ν), a human-authored formal specification (ψ^*), a Lean function signature (π_{sig}) for the implementation, and Lean theorems for both specification equivalence and implementation correctness. All formal specifications are written as *non-computable* logical propositions — i.e., they use quantifiers and logical connectives that cannot be directly evaluated — ensuring that models cannot copy implementation logic from specification syntax.

During evaluation, a model being evaluated on the benchmark starts with the natural-language description ν. Given this text, the model must generate:

(1) a formal Lean specification ψ, expressed as a predicate (a function that returns a Lean 4 proposition i.e. `Prop`),

(2) a proof that ψ is semantically equivalent to a hidden ground-truth Lean specification ψ^*,

(3) a Lean implementation[4] π that matches the function signature (π_{sig}) and is designed to satisfy ψ^* (and hence ψ), and

(4) a formal proof establishing that π satisfies ψ^*.

These steps (Figure 1) form two certification goals: (1) *Specification certification:* Steps 1–2 verify that the model correctly inferred the intended behavior. (2) *Implementation certification:* Steps 3–4 verify that the generated implementation satisfies the formal intent.

Our staged reasoning setup allows fine-grained diagnosis: models may fail at generating specifications, proving equivalence between the generated and ground-truth specifications, synthesizing implementations, or proving implementation correctness. For example, note that we require the generated implementation π to satisfy the ground-truth specification ψ^* instead of the model-generated specification ψ. This is because we want the evaluation of π to be independent of the ability of the model to generate the correct specification. More generally, failures at the various stages of our pipeline are independently diagnosed using Lean's type checker.

Challenges Encountered during Formalization. A key design decision in our benchmark is the use of *non-computable* specifications, which are predicates or functions in Lean that return propositions (`Prop` in Lean) that cannot be evaluated or simplified (decided by Lean) through computation alone. These contrast with *computable* specifications, written as executable functions or decidable predicates that Lean can reduce directly. While easier to verify, computable specs often *leak* the desired logic: models can copy them into implementations and produce trivial proofs via rewriting. Figure 2 shows the difference between a computable and a non-computable specification.

Figure 3 demonstrates the importance of this contrast. The left side (a–c) shows a computable spec whose logic is mirrored exactly in the GPT-4o-generated implementation, enabling a trivial proof. On the right (d–f), the spec is non-computable and requires symbolic reasoning to prove correctness. Notably, the GPT-4o-generated implementation in (e) does not mirror the spec, and the proof fails without further reasoning. This design ensures that models must engage in deeper logical inference, not just syntactic pattern matching. By using non-computable specs across our benchmark, we eliminate leakage and enforce truly verified reasoning from models.

Creating this benchmark involved substantial manual effort. On average, writing a formal specification took annotators *25 minutes per problem* on average, with an additional 15 minutes spent reviewing each other's specifications. Some problems involving complex non-computable specs required over an hour. To better understand problem difficulty and verify feasibility, we manually authored correctness proofs for a small random sample of benchmark problems. These ranged from *10 lines (e.g., problem_17) to 225 lines (e.g., problem_0)*, reflecting a wide span of proof complexity.

In addition to the main benchmark, we *release a small hand-curated few-shot prompt dataset* comprising of 5 problems distinct from HUMANEVAL. All of these problems include hand-written implementations, and some of them additionally include manually written equivalence and isomorphism proofs. For example, one correctness proof spans 309 lines, while corresponding isomorphism proofs range from 29 to 82 lines. This auxiliary dataset is intended to support prompt tuning and evaluation in few-shot or in-context learning setups.

[3]Not all problems could be formalized due to limitations in Lean 4 and its supported libraries.

[4]Here, we use the fact that Lean is not just a language for mathematical specifications and proofs but a full-fledged functional programming language.

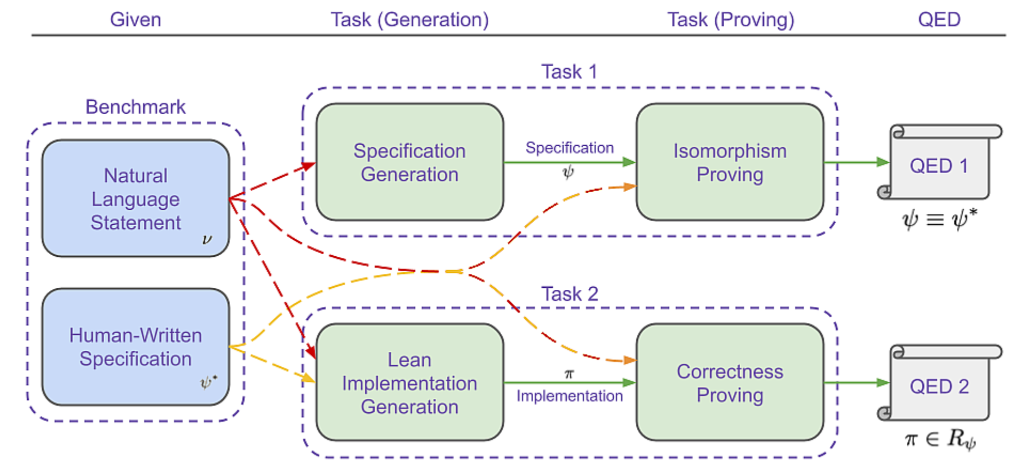

Figure 1: The two tasks of the CLEVER benchmark pipeline. Task 1 requires first generating a specification ψ from the natural language statement ν, then proving an isomorphism between the generated specification and a human-written specification ψ^*. Task 2 requires first generating a Lean implementation π, then proving its correctness according to the human-written specification. Both of these tasks must be completed correctly (reaching both QED 1 and QED 2) in order for a success to be counted.

(a)

```
-- computable spec
def problem_spec
-- function signature
(implementation: Nat → Nat)
-- inputs
(n: Nat) : Prop :=
-- spec
let spec (result: Nat) :=
match n with
| 0 => result = 0
| 1 => result = 1
| n' + 2 => result =
implementation n' +
implementation (n' + 1)
-- return value satisfies spec
∃ result, implementation n = result ∧ spec result
```

(b)

```
-- non-computable spec
inductive fibonacci_non_computable : ℕ → ℕ → Prop
| base0 : fibonacci_non_computable 0 0
| base1 : fibonacci_non_computable 1 1
| step  : ∀ n f₁ f₂,
fibonacci_non_computable n f₁ →
fibonacci_non_computable (n + 1) f₂ →
fibonacci_non_computable (n + 2) (f₁ + f₂)

def problem_spec
-- function signature
(implementation: Nat → Nat)
-- inputs
(n: Nat) :=
-- spec
let spec (result: Nat) :=
  fibonacci_non_computable n result
-- program termination
∃ result,
  implementation xs = result ∧ spec result
```

Figure 2: Two different specs for finding the n^{th} Fibonacci number. (a) shows a computable specification that *leaks* the implementation; (b) shows a non-computable specification leading to no-leakage of the implementation and enforcing the model to learn the deeper logical inference.

(a)

```
def problem_spec
(implementation: List Int → Int → Bool)
(q: List Int) (w: Int) :=
let spec (result : Bool) :=
  result ↔ (List.Palindrome q) ∧ (List.sum q ≤ w)
∃ result, implementation q w = result ∧ spec result
```

(b)

```
def implementation (q: List Int) (w: Int) : Bool :=
-- implementation generated by GPT-4o
List.Palindrome q ∧ List.sum q ≤ w
```

(c)

```
theorem correctness (q: List Int) (w: Int)
: problem_spec implementation q w := by
-- proof generated by GPT-4o
unfold problem_spec
let result := implementation q w
use result
simp [result]
simp [implementation]
```

(d)

```
def problem_spec
(implementation: List Int → Int → Bool)
(q: List Int) (w: Int) :=
let spec (result : Bool) :=
  (result → (List.Palindrome q)) ∧
  (result → (List.sum q ≤ w)) ∧
  (¬(List.Palindrome q) → ¬ result) ∧
  (¬(List.sum q ≤ w) → ¬ result)
∃ result, implementation q w = result ∧ spec result
```

(e)

```
def implementation (q: List Int) (w: Int) : Bool :=
-- implementation generated by GPT-4o
let is_palindrome := q = q.reverse
let sum_le_w := q.sum ≤ w
is_palindrome && sum_le_w
```

(f)

```
theorem correctness
(q: List Int) (w: Int)
: problem_spec implementation q w
:= by
-- proof generated by GPT-4o
unfold problem_spec
let result := implementation q w
use result
simp [result]
simp [implementation]
intro h -- <- The compilation fails here
simp [h]
exact List.eq_reverse_of_palindrome h.left
-- more proof trimmed
```

Figure 3: Illustration of specification leakage (left) and its mitigation (right) via non-computable specifications, using HUMANEVAL problem 72. The task is to return `true` iff a list `q` is a palindrome and its sum is at most `w`. In (a–c), the spec is *computable*: it encodes the desired logic in a Boolean expression, allowing the model to copy it directly in (b) and produce a trivial proof (c) via just unfolding and simplifying basic definitions used in the theorem statement. In contrast, (d–f) use a *non-computable* spec expressed in `Prop` with logical implications. The corresponding implementation (e), generated by GPT-4o using few-shot prompting, reflects the semantic intent without mirroring the spec. The proof (f) fails without additional reasoning, highlighting the challenge of proving correctness when logic cannot be mechanically unfolded. Non-computable specs thus act as guardrails, requiring models to reason rather than copy.

Curating the benchmark also revealed deeper challenges inherent to formal verification. For instance, in the HUMANEVAL problem involving root-finding for polynomials (see Figure 4), proving termination is difficult due to reliance on unbounded numerical search. Similarly, generating verified code for "finding all prime Fibonacci numbers" encounters foundational roadblocks, as there is no known proof that infinitely many such numbers exist—highlighting how natural language tasks can conceal deep mathematical issues when formalized. One potential way to deal with these types of formulations is by adding the concept of computational fuel and approximate answers (see Figure 4, and Figure 8 in Appendix A.2). Writing *non-computable* specifications is particularly challenging for problems that rely on language-level features like Python's `eval`, as seen in Problem 160. Since Lean lacks direct string-based evaluation, we had to reconstruct the behavior using inductive definitions over token lists and arithmetic expressions. This required converting a naturally computable task into a semantically equivalent, non-computable formulation without leaking implementation details. As shown in Figure 10 (in Appendix A.3), achieving this often involves layered recursive structures and careful abstraction to ensure both correctness and opacity.

Another instructive case is the problem of computing the MD5 checksum (problem 162). Here, the formal specification must, by necessity, describe the exact computation, making it closely related to the implementation itself. Since we could not find any popular hashing libraries in Lean, we chose not to formalize this specific problem. However, we prescribe the recipe for creating non-computable definitions in Appendix A.3, given that we know the computable definition.

While adapting HUMANEVAL to Lean, we encountered several language-level limitations. Some problems relying on dynamic typing or polymorphic return types—like Python's `Any`—could not be faithfully represented in a statically typed setting (e.g., problems 22 and 137). As a result, we were able to formalize 161 out of the original 164 problems. In problem 103, where the output is either a binary string or `None` based on input validity, we use `Option String` as the return type. In problem 129, where the function may return either a list of words or a number, we encode this using disjoint union type in Lean: `(List String)` $\oplus$ `Nat`, allowing only one of the two values to be populated at a time.

Prior work, such as FVAPPS (Dougherty & Mehta, 2025), relies on automatically generated specifications that can be incomplete or leaky, allowing trivial implementations (e.g., always returning zero) to pass (see Figure 7 in Appendix A.1). Our human-curated specifications ensure completeness and robustness, closing such loopholes and surfacing the real verification complexity hidden in everyday programming problems.

3. Evaluation

We evaluated several state-of-the-art LLMs and agentic approaches on CLEVER. Now we elaborate on the results.

Evaluation Metric. To fairly compare approaches that differ in model size, latency, and API usage, we adopt the metric *pass@k-seconds*—the fraction of benchmark problems solved within a fixed time budget k. A task is marked as solved only if both the formal specification and the implementation are generated and verified via Lean's type checker. As described in Figure 5, each step in the CLEVER pipeline (spec generation, equivalence proof, implementation, and correctness proof) is retried until a valid Lean-compilable output is found or the time runs out.

EVALUATE($approach$, timeout)

▷ *Assume* RETRY *retries the given function*
▷ *until it generates compilable Lean 4 code or timeouts.*
▷ RETRY *returns the Lean 4 code and remaining time.*
$t_{\text{rem}} \leftarrow$ timeout
$\psi,\ t_{\text{rem}} \leftarrow$ RETRY(`GenerateSpec`, ν, t_{rem})
$P_{\text{eq}},\ t_{\text{rem}} \leftarrow$ RETRY(`ProveEquivalence`, $(\psi,\ \psi^*)$, t_{rem})
if $t_{\text{rem}} \leq 0$ **return Fail**
$\pi,\ t_{\text{rem}} \leftarrow$ RETRY(`GenerateImpl`, $(\nu,\ \psi)$, t_{rem})
$P_{\chi},\ t_{\text{rem}} \leftarrow$ RETRY(`ProveCorrectness`, $(\pi,\ \psi^*)$, t_{rem})
if $t_{\text{rem}} \leq 0$ **return Fail**
return Success (all Lean 4 checks passed)

Figure 5: Evaluation strategy: retry each generation step until Lean compilation succeeds or a timeout is reached.

Evaluated Baselines. We evaluate three families of approaches for end-to-end verified code generation. The **Few-Shot Baseline** uses large language models (`GPT-4o`, `Claude-3.7`, `o4-mini`, and `DeepSeek-R1`) to generate all components—specifications, implementations, and proofs—via few-shot prompting with 1–2 exemplars. This baseline assesses the raw capability of LLMs to reason formally without task-specific training or tooling. The **COPRA Baseline** replaces the proof generation steps (Stages 2 and 4) with COPRA (Thakur et al., 2024), a neuro-symbolic proof search agent designed to produce Lean-compatible proofs when provided with an off-the-shelf foundational model and a Lean theorem statement to prove. This setup isolates proof search difficulty from the upstream generation task.

Results. Our primary evaluation metric focuses strictly on semantic correctness: a task is considered successful only if both the specification and the implementation are formally certified via Lean proofs. This strict definition ensures that reported scores reflect genuine end-to-end verification. However, to better diagnose failure modes, we also report auxiliary statistics: the fraction of tasks where generated specifications and implementations *compile* successfully. These serve as proxies for the model's fluency in Lean and its ability to produce well-typed artifacts.

In particular, implementation compilation includes not only type-checking against the declared function signature, but also validation against a suite of example-based test cases adapted from the original HUMANEVAL prompts. While passing these tests provides some evidence of functional correctness(Liu et al., 2023), we deliberately exclude them from our core success metric—since test cases offer only partial coverage and cannot guarantee semantic soundness (see Section 2 for discussion).

As shown in Table 1, compilation rates are broadly similar across few-shot models for both specification and implementation generation. A notable exception is the higher implementation compilation rate achieved by `o4-mini`, which contrasts with its lower success in proving correctness. More generally, even when an approach successfully certifies multiple specifications or verifies correctness for multiple implementations, the overall end-to-end success rate remains low. This is largely due to *mismatch*: tasks for which specification certification is tractable are often those where implementation correctness proofs are especially difficult, and vice versa. As a result, the joint success condition is rarely satisfied.

Another interesting observation is that `Claude-3.7`, when used along with COPRA, can certify more implementations (14) than all other models; however, its performance on specification certification is only comparable to other models. We believe that this might have to do with the length of proofs needed for specification certification, and hence, in the limited timeout it is hard to find the full proof for specification.

Proof Difficulty and Structure. As shown in Table 2, proofs for **specification certification** are consistently longer and harder to generate than those for implementation cor-

Approach Components					Pass@k-sec				
					Spec Cert.		Impl Cert.		End-to-End
Model	Spec Gen	Equiv Proof	Impl Gen	Corr Proof	*Compiled*	*Proved*	*Compiled*	*Proved*	
		Few-Shot Baseline							
GPT-4o	FS	FS	FS	FS	84.472%	0.621%	68.323%	0.621%	0%
o4-mini	FS	FS	FS	FS	82.609%	1.242%	83.230%	1.863%	0.621%
Claude-3.7	FS	FS	FS	FS	86.957%	0.621%	65.217%	1.863%	0.621%
DeepSeek-R1	FS	FS	FS	FS	71.42%	0.621%	60.870%	5.559%	0.621%
		COPRA Baseline							
GPT-4o	FS	COPRA	FS	COPRA	76.398%	1.863%	68.323%	3.727%	0.621%
Claude-3.7	FS	COPRA	FS	COPRA	81.366%	1.242%	65.217%	8.696%	0.621%

Table 1: Evaluation of different strategies for **end-to-end verified code generation**. Each approach consists of five components: **Model** (LLM used), **Spec Gen** (formal specification generation), **Equiv Proof** (proof of equivalence to ground-truth spec), **Impl Gen** (program synthesis), and **Corr Proof** (proof of implementation correctness). **FS** indicates few-shot prompting with 1–2 examples. Evaluation follows the pipeline in Figure 5. **Pass@k-seconds** with $k = 600$ reports the fraction of tasks where Lean successfully compiles the outputs and accepts the associated proofs within a 600-second time budget. The **Compiled** columns indicate whether the generated Lean code is syntactically valid and type-checks. The **Proved** columns reflect whether the corresponding proofs were accepted by Lean's kernel, thereby certifying semantic correctness. The **End-to-End** column reports full pipeline success—i.e., both the specification and implementation must compile and their respective proofs must be accepted. Despite strong models like GPT-4o achieving high compilation rates, formal correctness remains challenging: no approach has yet succeeded across all stages on more than one problem (specifically problem 53).

rectness. This is expected: proving that a generated spec is semantically equivalent to a non-computable reference specification requires models (or agents) to reason abstractly about intent, without access to implementation-level cues. In contrast, correctness proofs often benefit from direct pattern matching or automation through tactics like `simp`.

This distinction is especially evident in the only problem for which an end-to-end verified code generation succeeds across multiple models: **problem 53**, which asks for the sum of two integers. Despite the simplicity of the implementation, the ground-truth specification is expressed in a way that deliberately obfuscates the target behavior. This design makes the equivalence proof non-trivial and requires models (or COPRA) to recover the algebraic structure underlying addition. Even here, success is only possible because the proofs admit aggressive automation via `simp` and `ring`. The full problem is shown in Figure 6, which illustrates the separation between syntactic and semantic difficulty across spec, implementation, and proofs.

Notably, `Claude-3.7` in combination with COPRA successfully solves every implementation certification task that any other approach is able to solve. Figure 21 in Appendix A.5 illustrates one such case, showcasing a 35-line proof for the Brazilian factorial task that requires symbolic reasoning over factorial identities and recursive structure.

Unlike math-focused benchmarks such as MiniF2F (Zheng et al., 2021), where many proofs are short, goal-directed, and amenable to automation via tactics like `linarith`, `ring`, or `simp`, the proofs in our benchmark often mirror the *control flow* and *branching structure* of programs. As a result, standard automation is rarely sufficient. Correctness proofs frequently require reasoning case-by-case over pattern-matched inputs, recursive call structure, or multiple conditional branches. Even when the final goal involves simple arithmetic, the surrounding structure demands explicit handling of recursive unrolling, constructor cases, or fuel-based invariants. For example, proving correctness for recursive implementations like factorial products or root-finding procedures involves handling termination branches, intermediate values, and variable dependencies that make tactics like `linarith` or `ring` ineffective without significant manual decomposition. This structurally rich proof landscape contrasts with the often-flat logical forms seen in MiniF2F and underscores the need for symbolic agents like COPRA that can perform guided proof search beyond tactic chaining.

4. Related Work

Formal Verification. Formal verification encompasses a range of techniques aimed at mathematically proving the correctness of software or hardware systems with respect to a formal specification, thereby providing strong guarantees beyond traditional testing. Dafny and Verus (Leino, 2010; Lattuada et al., 2023) utilize SMT solvers to perform verification given proper verification conditions. Interactive theorem provers like Lean, Isabelle, and Coq (de Moura et al., 2015; Paulson, 1994; Huet et al., 1997) offer highly expressive logics where users construct proofs interactively with tactic-based automation. Notably, interactive theorem

Model	Approach	Certification	# Qed	Avg. # Lines	# Line (Min-Max)	Avg. Time (s)
GPT-4o	FS	Spec	1	16.0	16–16	124.3
GPT-4o	FS	Impl	1	6.0	6–6	291.6
o4-mini	FS	Spec	2	29.5	26–33	87.0
o4-mini	FS	Impl	3	14.0	10–21	204.0
Claude-3.7	FS	Spec	1	38.0	38–38	195.7
Claude-3.7	FS	Impl	3	12.7	6–21	414.4
DeepSeek-R1	FS	Spec	1	26.0	26–26	170.8
DeepSeek-R1	FS	Impl	9	14.1	3–27	137.73
GPT-4o	COPRA	Spec	3	26.3	16–44	97.9
GPT-4o	COPRA	Impl	6	10.8	6-19	199.6
Claude-3.7	COPRA	Spec	2	30.5	16-45	308.7
Claude-3.7	COPRA	Impl	14	14.3	4–35	165.8

Table 2: Analysis of successfully generated proofs across different models and certification types. We report: (1) the number of problems for which the correctness (isomorphism resp.) proofs are found by the approach in the column "# Qed" (see Figure 1), (2) the average number of lines in the proof, (3) the range of proof lengths (min–max), and (4) the average time it took for the approach to find a proof (given a proof was found). This analysis highlights variation in proof complexity and model behavior across settings. Few-shot prompting typically yields shorter, more brittle proofs, while COPRA-augmented configurations show higher robustness, with more consistent success and a broader range of proof strategies. Proof line counts serve as a coarse indicator of reasoning complexity.

provers have been involved in the verification of C compilers, microkernels, and distributed systems protocols (Leroy, 2009a; Klein et al., 2009a; Wilcox et al., 2015).

Benchmarks. Recent efforts have developed benchmarks for formal verification with the onset of powerful neural models. FVAPPS (Dougherty & Mehta, 2025) uses an LLM on scraped competition problems to automatically create formal specifications for 4715 problems, 1083 of which are guarded with test cases. However, the formal specifications themselves are often easily hackable (see Appendix A.1), with verification correctness guarded by a layer of test cases. Here, we aim to provide complete formal specifications, which cannot be done accurately with automatic annotation. miniCodeProps (Lohn & Welleck, 2024) contains 201 verification problems regarding data structures and induction problems; however, they do not include specification synthesis or equivalence tests. DafnyBench (Loughridge et al., 2024) is a benchmark of 782 stand-alone Dafny programs collected from prior benchmarks and Dafny repositories, where the synthesis task is to generate the verification conditions that allow Dafny to prove correctness. At the time, the best model was `Claude 3 Opus` which solved ≈ 68 % of the problems. Software engineering benchmarks have become extremely popular in recent literature, including benchmarking performance fixing real-world issues (Jimenez et al., 2024) and contamination-free code generation (Jain et al., 2024). In our work, we employ HUMANEVAL (Chen et al., 2021) to create CLEVER, our formal verification and synthesis benchmark. Formal verification is also applied in mathematical domains. Mathlib (mathlib Community, 2020) and the Archive of Formal Proofs (AFP) constitute formal mathematical repositories in Lean and Isabelle respectively, from which benchmarks have been derived (Hu et al., 2025; Jiang et al., 2021). ProofNet (Azerbayev et al., 2023) serves as a benchmark for producing proper specifications of mathematical problems. PutnamBench (Tsoukalas et al., 2024) is a formal benchmark of undergraduate-level competition problems in Lean, Isabelle, and Coq.

Proving Methods. Recent advances in neural models and LLMs have led to increased attention on formal verification and theorem-proving. AlphaVerus (Aggarwal et al., 2024) introduces a tree search and refinement algorithm to self-improve at producing formally verified Verus code. Similarly, SAFE (Chen et al., 2024) performs expert iteration in producing high-quality specification and proofs for generating verified Verus code. FVEL (Lin et al., 2024) uses symbolic methods to convert C programs into Isabelle, and then uses an LLM to generate correctness specifications which it then tries to prove. However, the automatic nature of the specification generation means correctness is not guaranteed. For mathematical theorem-proving, approaches involve tree search (Polu & Sutskever, 2020; Yang et al., 2023), reinforcement learning (Lin et al., 2025; Lample et al., 2022), LLMs (Thakur et al., 2024; Xin et al., 2024), and data augmentation and scale (Dong & Ma, 2025; DeepMind, 2024).

5. Conclusion

We introduced a new benchmark for end-to-end verified code generation that shifts the focus from surface-level correctness to formal semantic guarantees. Unlike prior benchmarks that rely on test cases or computable specifications, our tasks are grounded in *non-computable*, logic-based specifications that are explicitly designed to prevent implementation leakage. By enforcing a separation between specification intent and implementation behavior, the benchmark demands genuine reasoning rather than pattern matching or memorization.

Our evaluation protocol is deliberately staged, decomposing the pipeline into independently checkable phases: specification generation, isomorphism proof, implementation synthesis, and correctness proof. This staged design enables fine-grained diagnosis of where models succeed and fail—whether in interpreting informal intent, aligning it with formal meaning, or synthesizing verifiably correct programs. In particular, verifying the generated specification via *isomorphism proofs* ensures semantic fidelity and introduces a novel opportunity: verified *mining* of natural language and formal specification pairs from model generations, which could be reused for bootstrapping new training data.

Our benchmark introduces challenges beyond those in mathematical theorem-proving settings like miniF2F, where proofs are often short and tactic-friendly. In contrast, our tasks reflect the branching structure of real-world programs, requiring symbolic reasoning over control flow, recursion, and invariants—scenarios where automation alone breaks down. By combining structural complexity with formal soundness, non-leakage by design, and staged verification, the benchmark offers a rigorous, semantics-grounded testbed for verified code generation. It sets a new standard for advancing neural-symbolic reasoning toward scalable, trustworthy software verification.

References

AFP. Archive of Formal Proofs — isa-afp.org. `https://www.isa-afp.org/`, 2004. [Accessed 25-05-2024].

Aggarwal, P., Parno, B., and Welleck, S. Alphaverus: Bootstrapping formally verified code generation through self-improving translation and treefinement, 2024. URL `https://arxiv.org/abs/2412.06176`.

Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., and Avigad, J. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics, 2023. URL `https://arxiv.org/abs/2302.12433`.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chen, T., Lu, S., Lu, S., Gong, Y., Yang, C., Li, X., Misu, M. R. H., Yu, H., Duan, N., Cheng, P., Yang, F., Lahiri, S. K., Xie, T., and Zhou, L. Automated proof generation for rust code via self-evolution, 2024. URL `https://arxiv.org/abs/2410.15756`.

de Moura, L., Kong, S., Avigad, J., Van Doorn, F., and von Raumer, J. 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*, pp. 378–388. Springer, 2015.

DeepMind. AI achieves silver-medal standard solving International Mathematical Olympiad problems. Google DeepMind Blog, July 2024. URL `https://deepmind.google/discover/blog/ai-achieves-silver-medal-standard-solving-international-mat` Accessed on 2025-04-22. Blog post announcing AlphaProof and AlphaGeometry 2 results at IMO 2024. Technical details on AlphaProof were stated to be forthcoming.

Dong, K. and Ma, T. Stp: Self-play llm theorem provers with iterative conjecturing and proving, 2025. URL `https://arxiv.org/abs/2502.00212`.

Dougherty, Q. and Mehta, R. Proving the coding interview: A benchmark for formally verified code generation, 2025. URL `https://arxiv.org/abs/2502.05714`.

Hu, J., Zhu, T., and Welleck, S. minictx: Neural theorem proving with (long-)contexts, 2025. URL `https://arxiv.org/abs/2408.03350`.

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

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Jiang, A. Q., Li, W., Han, J. M., and Wu, Y. Lisa: Language models of isabelle proofs, 2021.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pp. 207–220, New York, NY, USA, 2009a. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 207–220, 2009b.

Lample, G., Lacroix, T., Lachaux, M.-A., Rodriguez, A., Hayat, A., Lavril, T., Ebner, G., and Martinet, X. Hypertree proof search for neural theorem proving. *Advances in Neural Information Processing Systems*, 35:26337–26349, 2022.

Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., and Hawblitzel, C. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023. doi: 10.1145/3586037. URL https://doi.org/10.1145/3586037.

Leino, K. R. M. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pp. 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3642175104.

Leroy, X. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, jul 2009a. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

Leroy, X. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009b.

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, 2024. URL https://arxiv.org/abs/2404.09939.

Lin, X., Cao, Q., Huang, Y., Wang, H., Lu, J., Liu, Z., Song, L., and Liang, X. Fvel: Interactive formal verification environment with large language models via theorem proving, 2024. URL https://arxiv.org/abs/2406.14408.

Lin, Y., Tang, S., Lyu, B., Wu, J., Lin, H., Yang, K., Li, J., Xia, M., Chen, D., Arora, S., and Jin, C. Goedel-prover: A frontier model for open-source automated theorem proving, 2025. URL https://arxiv.org/abs/2502.07640.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Lohn, E. and Welleck, S. minicodeprops: a minimal benchmark for proving code properties, 2024. URL https://arxiv.org/abs/2406.11915.

Loughridge, C., Sun, Q., Ahrenbach, S., Cassano, F., Sun, C., Sheng, Y., Mudide, A., Misu, M. R. H., Amin, N., and Tegmark, M. Dafnybench: A benchmark for formal software verification, 2024. URL https://arxiv.org/abs/2406.08467.

mathlib Community, T. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, POPL '20. ACM, January 2020. doi: 10.1145/3372885.3373824. URL http://dx.doi.org/10.1145/3372885.3373824.

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

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

Thakur, A., Tsoukalas, G., Wen, Y., Xin, J., and Chaudhuri, S. An in-context learning agent for formal theorem-proving. In *First Conference on Language Modeling*, 2024.

Tsoukalas, G., Lee, J., Jennings, J., Xin, J., Ding, M., Jennings, M., Thakur, A., and Chaudhuri, S. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition, 2024. URL https://arxiv.org/abs/2407.11214.

Wilcox, J. R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M. D., and Anderson, T. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pp. 357–368, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737958. URL https://doi.org/10.1145/2737924.2737958.

Xin, H., Guo, D., Shao, Z., Ren, Z., Zhu, Q., Liu, B., Ruan, C., Li, W., and Liang, X. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data, 2024.

Yang, K., Swope, A. M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. Leandojo: Theorem proving with retrieval-augmented language models. *arXiv preprint arXiv:2306.15626*, 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.

Zheng, K., Han, J. M., and Polu, S. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

A. Appendix

A.1. FVAPPS Benchmark

The FVAPPS benchmark (Dougherty & Mehta, 2025) is another code generation benchmark in Lean. However, unlike CLEVER, which requires a comprehensive proof of full program behavior, FVAPPS only requires the proof of a limited selection of properties of the program. The limitations of this are illustrated by the FVAPPS example in Figure 7. Here, a problem with a relatively complex natural language description only requires verifying lower-bound and upper-bound properties of the program implementation, as well as a few simple base cases. As can be seen, these properties are provably satisfied by a trivial program that always outputs 0 regardless of the input. Thus, it is clear that only requiring the proof of a small handful of properties does not capture the full intent of the natural language problem. This highlights the necessity of a verified code generation benchmark to require proofs of full program behavior, not just program properties.

A.2. Hard to write Specifications

Figure 8 shows some problems for which the formal specification or the implementation is hard to write.

A.3. Writing non-computable specifications

Figure 9 shows a computable vs non-computable version of the specification for finding the n^{th} Fibonacci number. It can be observed that the computable version of the specification *leaks* the implementation in contrast to the non-computable version. The non-computable specification uses an **inductive** definition of a recursive function.

Writing *non-computable* specifications is a non-trivial task that requires a deep understanding of the problem. Figure 10 (problem 160) presents another complex example illustrating the difficulty of formulating such specifications. Figure 10 shows two versions of a specification for evaluating an expression given as a list of strings (`["2","+","3","*","4","-","5"]`). Figure 10(a) evaluates the expression and later checks if the output matches the result (not specified in the figure), which is computable. Figure 10(b) shows a non-computable version of the specification that checks if the result matches the output of evaluating the expression without leaking the implementation. One can notice that we need multiple inductive recursive definitions to ensure that the specification is clean and non-computable.

A.4. Baseline Prompts

Snippets of the few-shot specification generator's system and example prompts are shown in **??** and **??**. Snippets of the few-shot isomorphism prover's system and example prompts are shown in ?? and ??. COPRA's system prompt, used for both isomorphism and correctness, is nearly identical to the original one in the COPRA paper (Thakur et al., 2024). Snippets of COPRA's example prompt for isomorphism are shown in ??.

Snippets of the few-shot implementation generator's system and example prompts are shown in ?? and ??. Snippets of the few-shot correctness prover's system and example prompts are shown in ?? and ??. Snippets of COPRA's example prompt for correctness are shown in ??.

A.5. Some Proof Found

Figure 21 shows an example of a proof found for implementation certification by `Claude-3.7` using COPRA.

(a)

```
def problem_spec
-- function signature
(implementation: List Rat →
      Rat)
-- inputs
(xs: List Rat) :=
-- spec
let spec (result: Rat) :=
  let eps := (1: Rat) / 1000000;

  xs.length ≥ 1 → xs.length %
       2 = 0 →
  ∀ poly : Polynomial Rat,
    poly.degree = some (xs.
      length - 1) →
    (∀ i, i ≤ xs.length - 1 →
       poly.coeff i = xs.get! i)
       →
    |poly.eval result| ≤ eps;
-- program termination
∃ result,
  implementation xs = result ∧
  spec result
```

(b)

```
-- possible implementation
      using Newton's method
def implementation (xs: List
      Rat) : Rat :=
let rec poly (xs: List Rat) (x:
       Rat) := xs.reverse.foldl
      (λ acc a => acc * x + a)
      0;
let rec poly' (xs: List Rat) (x:
       Rat) := (xs.drop 1).
      reverse.foldl (λ acc a =>
       acc * x + a) 0;
let rec eps := (1: Rat) /
      1000000;
let rec find_zero (xs: List Rat
      ) (guess: Rat) (fuel: Nat)
       :=
let eval := poly xs guess;
let eval' := poly' xs guess;
if eval ≤ eps ∨ fuel = 0 then
      (guess, fuel)
else
let guess' := (eval' * guess -
      eval) / eval';
find_zero xs guess' (fuel - 1);
(find_zero xs 1.0 1000000).1
```

Figure 4: **Polynomial Root-Finding.** Problem 32 asks for an approximate real root of a degree-n polynomial. The spec enforces small residual error ($< 10^{-6}$). The implementation uses Newton's method with bounded recursion; proving termination is non-trivial due to lack of guaranteed derivative behavior.

(a)

```
def problem_spec (impl : Int
      → Int → Int) (x y :
      Int) :=
  let spec (res : Int) := res -
      x - y = 0
  ∃ result, impl x y = result
      ∧ spec result
```

(b)

```
def generated_spec (impl : Int
      → Int → Int) (x y :
      Int) : Prop :=
  impl x y = x + y
```

(c)

```
def implementation (x y : Int)
      : Int := x + y
```

(d)

```
theorem correctness (x y : Int)
       : problem_spec
      implementation x y :=
by
  unfold problem_spec
  let result := implementation
      x y
  use result
  simp [result]
  simp [implementation]
```

(e)

```
theorem spec_isomorphism :
  ∀ impl, (∀ x y, problem_spec
      impl x y) ↔
          (∀ x y,
      generated_spec impl x y)
      :=
by
  intro impl
  apply Iff.intro
  -- → direction
  intro h_prob_spec
  intro x y
  have h := h_prob_spec x y
  simp [generated_spec,
      problem_spec] at h
  rw [generated_spec]
  linarith
  -- ← direction
  intro h_gen_spec
  intro x y
  unfold problem_spec
  simp
  have h := h_gen_spec x y
  simp [generated_spec] at h
  rw [h]
  ring
```

Figure 6: **End-to-end verified example: Problem 53 (Add Two Numbers).** This task requires adding two integers `x` and `y`. Shown are all components of the certification pipeline: (a) a non-computable ground truth spec using subtraction to hide the implementation, (b) the model-generated spec, (c) the implementation `x + y`, (d) a short correctness proof, and (e) an isomorphism proof relating the two specs. While the implementation is simple, the spec equivalence proof requires symbolic reasoning. This is the only HumanEval-derived task with full verification across multiple approaches.

```
/--
solve_elections:
There are n voters, and two ways to convince
    each of them to vote for you. The first way
     to convince the i-th voter is to pay him
    p_i coins. The second way is to make m_i
    other voters vote for you, and the i-th
    voter will vote for free. Moreover, the
    process of such voting takes place in
    several steps. For example, if there are
    five voters with m_1 = 1, m_2 = 2, m_3 = 2,
    m_4 = 4, m_5 = 5, then you can buy the vote
    of the fifth voter, and eventually everyone
     will vote for you. Set of people voting
    for you will change as follows: 5 → 1,5 →
     1,2,3,5 → 1,2,3,4,5. Calculate the
    minimum number of coins you have to spend
    so that everyone votes for you.
-/

def solve_elections (n : Nat) (voters : List (
    Nat × Nat)) : Nat := 0

theorem solve_elections_nonnegative (n : Nat) (
    voters : List (Nat × Nat)) :
    solve_elections n voters >= 0 :=
by rfl

theorem solve_elections_upper_bound (n : Nat) (
    voters : List (Nat × Nat)) :
    solve_elections n voters <= List.foldl (λ
    acc (pair : Nat × Nat) => acc + pair.2) 0
    voters :=
Nat.zero_le _

theorem solve_elections_zero_votes (n : Nat) (
    voters : List (Nat × Nat)) : (List.all
    voters (fun pair => pair.1 = 0)) ->
    solve_elections n voters = 0 :=
fun _ => rfl

theorem solve_elections_single_zero_vote :
    solve_elections 1 [(0, 5)] = 0 :=
by rfl
```

Figure 7: FVAPPS sample 23 and a trivial program that solves it, illustrating the limitations of not verifying full program behavior.

(a)

```
def problem_spec
-- function signature
(implementation: List Rat → Rat)
-- inputs
(xs: List Rat) :=
-- spec
let spec (result: Rat) :=
  let eps := (1: Rat) / 1000000;
  xs.length ≥ 1 → xs.length % 2 = 0 →
  ∀ poly : Polynomial Rat,
    poly.degree = some (xs.length - 1) →
    (∀ i, i ≤ xs.length - 1 → poly.coeff i = xs.
     get! i) →
    |poly.eval result| ≤ eps;
-- program termination
∃ result,
  implementation xs = result ∧
  spec result

-- possible implementation using Newton's method
def implementation (xs: List Rat) : Rat :=
let rec poly (xs: List Rat) (x: Rat) := xs.reverse.
     foldl (λ acc a => acc * x + a) 0;
let rec poly' (xs: List Rat) (x: Rat) := (xs.drop 1)
     .reverse.foldl (λ acc a => acc * x + a) 0;
let rec eps := (1: Rat) / 1000000;
let rec find_zero (xs: List Rat) (guess: Rat) (fuel:
      Nat) :=
let eval := poly xs guess;
let eval' := poly' xs guess;
if eval ≤ eps ∨ fuel = 0 then (guess, fuel)
else
let guess' := (eval' * guess - eval) / eval';
find_zero xs guess' (fuel - 1);
(find_zero xs 1.0 1000000).1
```

(b)

```
def problem_spec
-- function signature
(implementation: Nat → Nat)
-- inputs
(n: Nat) :=
-- spec
let spec (result: Nat) :=
  n > 0 →
    (∃ i, Nat.fib i = result ∧ Nat.Prime result ∧
      (∃! S : Finset Nat, S.card = n - 1 ∧
      (∀ y ∈ S, (∃ k, y = Nat.fib k) ∧ y < result
     ∧ Nat.Prime y))
    )

-- implementation without proof of
-- termination
def implementation (n: Nat) : Nat :=
let rec fib_prime (n: Nat) (i: Nat) : Nat :=
  if Nat.Prime (Nat.fib i) then
    if n = 1 ∨ n = 0
    then Nat.fib i
    else fib_prime (n - 1) (i + 1)
  else fib_prime n (i + 1)
termination_by n
decreasing_by
  -- Proof of termination is open problem
  sorry
  sorry
fib_prime n 0
```

Figure 8: Examples of benchmark challenges. (a) Polynomial root-finding: difficulties in proving termination of numerical search; (b) Prime Fibonacci finder: problem complexity rooted in the lack of a known proof of infinitude.

(a)

```
-- computable spec
def problem_spec
-- function signature
(implementation: List Nat → Nat)
-- inputs
(n: Nat) :=
-- spec
let spec (result: Nat) :=
  (n = 0 → result = 0) ∨
  (n = 1 → result = 1) ∨
  (2 ≤ n → ∃ fib_array : List Nat,
   fib_array.length = n + 1 ∧
   fib_array[0]! = 0 ∧
   fib_array[1]! = 1 ∧
   (∀ i, 1 < i → i < n + 1 →
   fib_array[i]! = fib_array[i - 1]! +
   fib_array[i - 2]!) ∧
   result = fib_array[n]!)
-- program termination
∃ result,
  implementation xs = result ∧
  spec result
```

(b)

```
-- non-computable spec
inductive fibonacci_non_computable : ℕ → ℕ →
    Prop
| base0 : fibonacci_non_computable 0 0
| base1 : fibonacci_non_computable 1 1
| step  : ∀ n f₁ f₂,
fibonacci_non_computable n f₁ →
fibonacci_non_computable (n + 1) f₂ →
fibonacci_non_computable (n + 2) (f₁ + f₂)

def problem_spec
-- function signature
(implementation: Nat → Nat)
-- inputs
(n: Nat) :=
-- spec
let spec (result: Nat) :=
  fibonacci_non_computable n result
-- program termination
∃ result,
  implementation xs = result ∧
  spec result
```

Figure 9: Two different specs for finding the n^{th} Fibonacci number. (a) shows a computable specification that *leaks* the implementation; (b) shows a non-computable specification leading to no-leakage of the implementation and enforcing the model to learn the deeper logical inference.

(a)

```
inductive Op where
  | add | sub | mul | floordiv
deriving Repr, DecidableEq

def parseOp : String → Option Op
  | "+" => some .add | "-" => some .sub
  | "*" => some .mul | "//" => some .floordiv
  | _ => none

def precedence : Op → Nat
  | .mul | .floordiv => 2
  | .add | .sub      => 1

def apply : Op → Int → Int → Int
  | .add, a, b => a + b
  | .sub, a, b => a - b
  | .mul, a, b => a * b
  | .floordiv, a, b => a / b

inductive Tok where
  | num : Int → Tok
  | op  : Op → Tok
deriving Repr

def tokenize : List String → Option (List Tok)
  | [] => some []
  | s :: t =>
    match parseOp s with
    | some o => (tokenize t).map (Tok.op o :: ·)
    | none   => s.toInt?.bind (fun n => (tokenize t)
     .map (Tok.num n :: ·))

partial def evalPass (xs : List Tok) (ops : List Op)
     : List Tok :=
  match xs with
  | Tok.num a :: Tok.op o :: Tok.num b :: rest =>
    if o ∈ ops then evalPass (Tok.num (apply o a b)
     :: rest) ops
    else Tok.num a :: Tok.op o :: evalPass (Tok.num
     b :: rest) ops
  | x :: xs => x :: evalPass xs ops
  | [] => []

def evalTokens (tokens : List Tok) : Option Int :=
  let result := [[.mul, .floordiv], [.add, .sub]].
     foldl evalPass tokens
  match result with | [Tok.num n] => some n | _ =>
     none

def do_algebra (input : List String) : Option Int
     :=
  tokenize input >>= evalTokens
```

(b)

```
def applyOp (x y : Int) : String → Option Int
  | "+"  => some (x + y)
  | "-"  => some (x - y)
  | "*"  => some (x * y)
  | "//" => if y == 0 then none else some (x / y)
  | _      => none

inductive evalArith_pass : List String → Int →
    Prop
| num {s : String} {n : Nat} (h : s.toNat! = n) :
    evalArith_pass [s] (Int.ofNat n)
| binOp {ts1 ts2 : List String} {op : String} {r1
    r2 r : Int}
    (h1 : evalArith_pass ts1 r1)
    (h2 : evalArith_pass ts2 r2)
    (hop : applyOp r1 r2 op = some r) :
    evalArith_pass (ts1 ++ op :: ts2) r

inductive evalArith_mul : List String → Int →
    Prop
| of_pass {ts r} (h : evalArith_pass ts r) :
    evalArith_mul ts r
| step {ts1 ts2 r1 r2 r} (h1 : evalArith_mul ts1 r1)
     (h2 : evalArith_mul ts2 r2)
    (hop : applyOp r1 r2 "*" = some r ∨ applyOp r1
    r2 "//" = some r) :
    evalArith_mul (ts1 ++ "*" :: ts2) r

inductive evalArith_add : List String → Int →
    Prop
| of_mul {ts r} (h : evalArith_mul ts r) :
    evalArith_add ts r
| step {ts1 ts2 r1 r2 r} (h1 : evalArith_add ts1 r1)
     (h2 : evalArith_add ts2 r2)
    (hop : applyOp r1 r2 "+" = some r ∨ applyOp r1
    r2 "-" = some r) :
    evalArith_add (ts1 ++ "+" :: ts2) r

-- Noncomputable spec to evaluate an expression
def do_algebra (input : List String) (result : Int)
     : Prop :=
  evalArith_add input result
```

Figure 10: Two different specs for evaluating an expression (as a list of strings): `["2","+","3","*","4","-","5"]`. (a) shows a computable specification that evaluates using *do_algebra*, and later checked with the result (b) shows a non-computable specification using an inductive definition where *do_algebra* checks if the result matches the value of the expression without leaks.

(a)

```
def problem_spec (impl : Nat → Nat) (n : Nat) :=
  let spec (result : Nat) :=
    let factorial := Nat.factorial n;
    (0 < n → result / factorial = impl (n - 1)) ∧
    (n = 0 → result = 1);
  ∃ result, impl n = result ∧ spec result
```

(b)

```
def implementation (n : Nat) : Nat :=
  match n with
  | 0 => 1
  | n+1 => Nat.factorial (n+1) * implementation n
```

(c)

```
#test implementation 4 = 288
```

(d)

```
theorem correctness (n : Nat) : problem_spec implementation n
      := by
  unfold problem_spec
  let result := implementation n
  use result
  simp [result]
  apply And.intro
  unfold implementation
  simp [Nat.factorial_succ]
  intro h_n_pos
  cases n
  exfalso
  exact h_n_pos.false
  rename_i n
  simp [Nat.succ_eq_add_one, Nat.add_sub_cancel]
  rw [Nat.factorial_succ]
  have h_fac_pos : 0 < (n + 1) * n.factorial := by
    exact Nat.mul_pos (Nat.succ_pos n) (Nat.factorial_pos n)
  have h_cancel : (n + 1) * n.factorial * implementation n
    / ((n + 1) * n.factorial) = implementation n := by
    rw [Nat.mul_div_cancel_left (implementation n) h_fac_pos]
  simp [h_cancel]
  unfold implementation
  cases n
  simp [Nat.factorial_zero]
  rename_i n
  simp [Nat.add_zero]
  simp [Nat.factorial_succ]
  left
  rw [implementation.eq_def]
  simp [Nat.mul_assoc]
  cases n
  simp [Nat.zero_eq]
  rename_i n
  simp [Nat.factorial_succ]
  rw [Nat.mul_assoc]
  intro h_n_eq_0
  rw [h_n_eq_0, implementation]
```

Figure 21: **Problem 139 (Brazilian Factorial)**: Given an integer `n`, compute the product of all factorials from `n!` down to `1!`. Part (a) defines the **ground truth specification**, which expresses recursive structure without leaking the implementation. Part (b) shows the **implementation** using a recursive product of factorials. Part (c) lists a **test case** used for validation. Part (d) presents the full **correctness proof**, showing that the implementation satisfies the spec. This proof, generated by COPRA using Claude-3.7, spans 35 lines and involves reasoning over factorial identities, case analysis, and symbolic manipulation.