BRIDGE: BUILDING REPRESENTATIONS IN DOMAIN GUIDED PROGRAM VERIFICATION

Robert Joseph George[1,2], Carson Eisenach[2], Udaya Ghai[2],
Dominique Perrault-Joncas, Anima Anandkumar[1], Dean Foster[2]

[1] California Institute of Technology [2] Amazon

Abstract: Large language models (LLMs) have achieved impressive results in code generation, yet struggle with program verification, especially in interactive proof frameworks such as LEAN4. A central challenge is scalability: verified synthesis requires not just code, but also precise specifications and correctness proofs, and existing approaches rarely span all three domains. We present **BRIDGE**, the first systematic study of structured prompting for scalable verified program generation. BRIDGE decomposes verification into three interconnected domains: *Code* (executable implementations), *Specifications* (formal intent statements), and *Proofs* (constructive correctness arguments). Our key idea is to elicit distinct reasoning behaviors—*functional*, *specification-driven*, and *proof-oriented*—as intermediate representations that preserve semantic structure and connect these domains. Through systematic ablations, we show that this approach substantially improves both accuracy and efficiency beyond standard error feedback methods. For example, functional reasoning improves correctness of code in formal languages (LEAN4) by nearly $1.5\times$ (pass@5) over direct baselines. In inference-time compute, functional reasoning is also $2\times$ *more efficient*, achieving higher pass rates with fewer generations and lower total sampling budgets. Similarly, we find that specification-driven prompting boosts Python coding pass rates by up to 17.5%. These findings suggest that structured domain alignment is a promising direction for advancing verified synthesis. BRIDGE establishes a foundation for training via expert iteration or RLVR, models that internalize these reasoning strategies across code, specifications, and proofs.

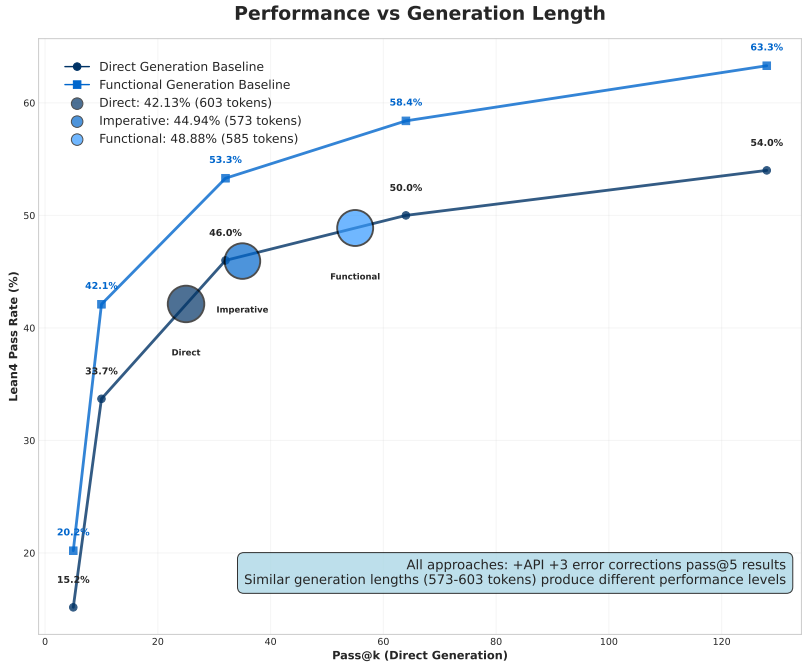


Figure 1: Functional reasoning improves verification accuracy while reducing inference-time compute. Lines show inference-time scaling via n-way parallel generation without iterative refinement. Circular markers (dots) report pass@5 with the Lean +API available in context (tool access) and with 3 rounds of error correction. The blue annotation applies only to the dot markers. Overall, functional reasoning reaches performance comparable to direct generation with roughly half the total sampling budget, improving both efficiency and accuracy.

1 Introduction

Program verification aims to establish not only that a program behaves correctly on known tests, but that it meets its specification *for all possible executions*. It traditionally requires producing three interconnected artifacts: (1) executable code that implements the desired behavior, (2) a formal specification that precisely defines the intended behavior, and (3) a theorem with a constructive proof showing that the code satisfies the specification. Classical verification frameworks such as DAFNY (19), BOOGIE (2), WHY3 (4), and large-scale efforts like SEL4 (17) and COMPCERT (20) have shown that such guarantees are possible and highly valuable. However, producing these artifacts has historically required expert human effort and months or years of engineering.

Large language models (LLMs) promise to automate this process. LLMs perform strongly on general programming tasks (6; 1), but current results show a performance cliff once formal guarantees are required. Benchmarks such as VERINA (37) show that GPT-4 achieves about 61.4% correct LEAN4 code and 51.0% sound specifications[1], but only 3.6% correct proofs at pass@5, rising to 22% only after 64 refinement attempts. CLEVER (28) reports that among 161 HumanEval-style tasks (6), only one could be fully verified end-to-end. VERIBENCH (25) similarly finds that even sophisticated models such as Claude 4 Sonnet compile fewer than 15% of Lean verification tasks. More recently, DAFNYCOMP (34) introduced a benchmark for compositional formal verification in Dafny, highlighting that local successes often fail to compose globally, while the VERICODING BENCHMARK (5) extends evaluation across Dafny, Verus, and Lean for verified program synthesis. On the other hand, for satisfiability modulo theories (SMT)-based deductive frameworks such as DAFNYBENCH (22), the best model+prompting scheme verified 68% of programs by generating hints that allow the SMT solver (via BOOGIE) to discharge verification conditions. Specification-synthesis systems like AUTOSPEC (33) also synthesize specifications using static analysis and verify 79% of C programs across four benchmarks. Despite these advances, none of these works provide a *scalable framework* for unifying all three domains—code, specifications, and proofs – instead treating them as loosely coupled translations. Our work explores *domain-specific reasoning strategies* tailored to each view (e.g., functional reasoning for code, property-based reasoning for specs, constructive reasoning for proofs) and how to connect them.

We propose **BRIDGE**, a prompting-based reasoning methodology that reframes automated formal verification as an inference-time process of maintaining *semantic consistency* across three interconnected domains: **Code** (executable implementations that Lean can type-check and reason about), **Specifications** (formal behavioral intent that constrains implementations), and **Proofs** (constructive correctness arguments built on top of code, specifications, and candidate theorems). Rather than requiring models to leap directly from natural language to full proofs, BRIDGE introduces structured prompting strategies that guide intermediate reasoning in each domain. The key idea is that these are interconnected views on the same mathematical object[2] and our approach allows models to move incrementally from problem statements to Lean code, from code to specifications, and from specifications to meaningful theorem statements and proofs—while leveraging insights across domains. To our knowledge, this is the *first work on verifiable program synthesis* to systematically apply chain-of-thought style prompting across all stages of the verification pipeline. While proofs remain the end goal, our results show that the critical bottleneck lies in reliably generating verifiable Lean code, meaningful specifications, and non-trivial theorem statements, a necessary foundation for any proof construction. We instantiate BRIDGE in LEAN4 (8; 24), but the methodology could be extended to other systems such as DAFNY and ROCQ.

Our central finding is that **reasoning strategy matters as much as language familiarity** for code synthesis in formal languages. Functional paradigms such as OCAML and HASKELL naturally support constructive reasoning and type-driven proof search, while imperative ones like PYTHON and C++ often conflict with the requirements of formal logic. Structured reasoning strategies yield consistent, multiplicative gains across code, specification, and proof generation. Moreover, functional reasoning is not only more accurate but also more *efficient*, achieving nearly $2\times$ the pass rates of direct baselines at comparable generation lengths. These inference-time results suggest that the main bottleneck is the *structured representation of reasoning*, rather than memorization of syntax or proof scripts. As a counterpoint, our small study of literate-programming–style generation (Appendix A) improved human readability but did not increase verification success, underscoring that concise, structured reasoning is more valuable than verbose narrative. Looking forward, reinforcement learning from verification rewards (RLVR) offers a path to internalize these strategies, moving beyond inference-time prompting toward learned reasoning that incorporates multiple views including code, specifications, and proofs.

Contributions. This paper makes the following contributions:

1. **BRIDGE Framework.** We introduce **BRIDGE**, the first prompting-based framework for verified code synthesis. It decomposes verification into three domains–*Code*, *Specifications*, and *Proofs* and provides a scalable way to generate and interconnect them. Unlike prior work, BRIDGE supports reasoning strategies tailored to each domain.
2. **Empirical Gains across Models and Domains.** Across 178 LeetCode algorithmic problems and five LLMs, BRIDGE yields substantial improvements. Functional prompting achieves up to a $1.5\times$ increase in success rates on verifiable code synthesis.

[1] By *sound specifications* we mean contracts that are not only syntactically valid but also semantically meaningful, i.e., they are neither vacuous nor trivially satisfiable.

[2] Similarly, one can think of a Python implementation as yet another representation

It also does so more *efficiently*, in some cases reaching double the pass rates of direct baselines with comparable generation lengths. Specification prompting enhances Python code pass rates by up to 17.5%, and proof prompting produces stronger, non-trivial theorems rather than vacuous statements.

3. **Reasoning Strategy and Language Familiarity Interaction.** We show that reasoning strategy and language familiarity interact multiplicatively: functional paradigms (OCAML, HASKELL) consistently outperform imperative ones (PYTHON, C++), even when less represented in pretraining. This highlights structured reasoning, rather than dataset frequency, as the central bottleneck.

Though instantiated in LEAN4, BRIDGE generalizes to both programming languages and verification systems (e.g., ROCQ, DAFNY, BOOGIE), enabling scalable, paradigm-aware code verification.

Table 1: Chain-of-thought (CoT) / reasoning prompting vs. benchmarks for software and verification. ● = supported/evaluated; ○ = not a focus. "Compose" indicates whether the method is designed to connect artifacts across Code/Specs/Proofs.

Work / Method	CoT Style	CodeGen	SpecGen	ProofGen	Compose
Prompting / Reasoning Methods					
COT PROMPTING (32)	Rationale CoT	●	○	○	○
SELF-CONSISTENCY (31)	Sample & vote	●	○	○	○
PAL (12)	Program-aided	●	○	○	○
REACT (36)	Reason+Act (tools)	●	○	○	○
TREE-OF-THOUGHT (35)	Search over CoT	●	○	○	○
REFLEXION (26)	Self-reflection	●	○	○	○
SELF-REFINE (23)	Iterative critique	●	○	○	○
CLOVER (27)	Closed-loop checks	●	●	○	○
AUTOSPEC (33)	Spec synthesis (SMT)	○	●	○	○
BRIDGE (ours)	Domain-specific CoT	●	●	●	●
Benchmarks					
DAFNYBENCH (22)	Solver-backed verify	○	●	○	○
MINICODEPROPS (21)	Properties in Lean	○	○	●	○
VERINA (37)	Lean eval (code+spec+proof)	●	●	●	○
CLEVER (28)	End-to-end Lean verify	●	●	●	○
DAFNYCOMP (34)	Compositional specs	○	●	○	○
VERICODING (5)	Cross-system vericoding	●	●	●	○

Paper Structure. Section 2 reviews related work in LLMs for program verification. Section 3 presents BRIDGE and its prompting strategies for Code, Specifications, and Proofs. Section 4 describes datasets, models, and evaluation setup. Section 5 reports empirical results and error analyses. Section 6 summarizes implications for verified AI and discusses future directions including reinforcement learning from verification rewards (RLVR) to internalize reasoning traces. Appendix A, B, and C present further analysis, including additional prompting strategies and detailed error breakdowns across paradigms.

2 Related Work

Formal verification has a rich history spanning both foundational theory and applied systems. We highlight the most relevant strands of prior work that inform our paradigm-aware approach to LLM-assisted verification.

Program Verification. Program verification aims to establish that a program satisfies its specification under all possible executions. This notion dates back to the seminal work of Hoare and Floyd, who introduced axiomatic semantics for reasoning about program correctness (14; 11). Dijkstra later formalized the notion of weakest preconditions in his book *A Discipline of Programming* (9). Subsequent decades saw the development of deductive verification frameworks and formal proof assistants (7). A key but often underemphasized factor is the influence of *programming paradigms* on verifiability: imperative languages such as C++ and PYTHON emphasize mutable state and loops, requiring complex reasoning about state transitions and loop invariants (15). By contrast, purely functional paradigms (e.g., OCAML, HASKELL) avoid side effects and encourage structural recursion, aligning naturally with logical reasoning (16). Logic- and constraint-based paradigms (e.g., PROLOG, MERCURY) further reduce the gap between specification and implementation, but lack widespread tooling. This paradigm gap helps explain

why systems like LEAN and ROCQ, which enforce functional and dependently typed programming styles (8; 3), are in principle more amenable to formal verification—though challenging for models trained primarily on imperative corpora.

Large-Scale Formal Verification. Projects such as the SEL4 microkernel and the COMPCERT verified compiler demonstrated that end-to-end proofs of functional correctness are possible for realistic software, but only at immense manual cost, often spanning a decade of expert engineering effort. Tools like DAFNY, BOOGIE, WHY3, and VERUS (18) aim to lower this barrier by translating annotated code into verification conditions discharged by SMT solvers, albeit at the expense of expressiveness and flexibility. These developments underscore the importance of bridging programming paradigms and automated reasoning systems for scalability.

Program Verification with SMT-Based Systems. Deductive verification frameworks such as DAFNY, BOOGIE, and VERUS verify programs by translating annotated code into logical verification conditions and discharging them with SMT solvers. Because proof search is delegated to automated backends, large language models (LLMs) can perform surprisingly well in these settings. For instance, DAFNYBENCH reports overall verification success rates around 60–70% for GPT-4-class models, and specification-synthesis methods like AUTOSPEC (33) achieve up to 79% success. These results suggest that when solvers can close the reasoning gap, the primary difficulty shifts from constructing proofs to generating correct and complete specifications.

Automated Formal Verification with LLMs. SMT-backed deductive verification systems provide one end of the spectrum. In this setting, CLOVER (27), evaluated on the CloverBench suite of Dafny programs, implements a closed-loop pipeline in which an LLM generates code, docstrings, and annotations and then uses consistency checks plus SMT-based verification to decide acceptance; on this benchmark it accepts the vast majority of correct solutions while rejecting all incorrect ones. VERIBENCH further systematizes these evaluations across multiple solver-based frameworks, illustrating that when powerful SMT backends are available, LLMs can achieve high end-to-end verification rates despite noisy intermediate outputs. Interactive theorem provers (ITPs) reveal sharper limits for LLM-based verification. Unlike solver-driven systems, ITPs require explicit proof construction, making them a stricter test of deep reasoning. Benchmarks such as VERINA show that even frontier models produce around 61% correct LEAN4 code and 51% sound specifications, yet only 3.6% correct proofs at pass@5, rising to 22% after 64 refinements. On CLEVER, which requires end-to-end verified Lean code, models succeed on just one of 161 tasks. The LEAN slice of VERIBENCH similarly reports that Claude 4 Sonnet compiles fewer than 15% of Lean verification tasks. Beyond Lean, DAFNYCOMP (34) and the VERICODING BENCHMARK (5) show that even when local verification succeeds, composing specifications and proofs across modules remains challenging, underscoring a broader brittleness. This pattern aligns with earlier formal methods literature (7) in highlighting the difficulty of scaling from local assertions to whole-program reasoning.

Interactive Theorem Proving and LEAN4. LEAN4 (8) unifies programming and proving through dependent type theory, making it unusually expressive but also demanding. Its MATHLIB4 library now includes over 200,000 theorems (24), yet LLMs struggle with Lean's strict termination discipline, rich type system, and tactic-based proof search. Work like THEOREMLLAMA (30) demonstrates that domain-specific fine-tuning can improve performance on isolated proofs but does not address full verification pipelines linking implementations, specifications, and theorems. Unlike SMT-based frameworks, Lean does not rely on external solvers to discharge generated conditions; every proof is checked internally by its kernel, yielding stronger guarantees but making the task substantially harder for models. This structural difference helps explain the stark performance gap between Dafny/Boogie-style tasks and Lean4 benchmarks.

Multi-View Representations of Programs. A complementary line of research explores representing programs from multiple semantic views. Models such as CODEBERT (10), GRAPHCODEBERT (13), and CLOVER show that jointly modeling code, documentation, and annotations can improve code understanding and generation. CLOVER in particular introduced closed-loop consistency checks, regenerating docstrings from code to detect semantic drift. Our work extends this multi-view philosophy to formal verification: BRIDGE treats *Code*, *Specifications*, and *Proofs* as three fundamental and interdependent views that must remain consistent for provable correctness.

3 Methodology

Existing approaches address code, specification, and proof generation separately, but lack a scalable and reasoning-driven framework that connects them. Most models reason locally within a single domain, rather than maintaining consistency across domains or leveraging structured reasoning about verification itself.

BRIDGE reframes verification as a structured inference-time reasoning process that operates across three interconnected domains *Code*, *Specifications*, and *Proofs*. Instead of relying on ground-truth annotations or monolithic fine-tuning, BRIDGE decomposes verification into modular reasoning stages, where each stage serves as a representational checkpoint. Code captures executable intent, specifications formalize behavioral constraints, and proofs establish correctness relationships between the

Minimum Key Pushes

You are given a string `word` consisting of distinct lowercase English letters. A traditional telephone keypad maps letters to the digit keys 2 through 9. Each key may contain multiple letters, and typing the k-th letter on a key requires exactly k pushes of that key.

You are allowed to *remap* the letters to keys 2 through 9 arbitrarily, subject to the following constraints:

- Each letter must be assigned to exactly one key.
- Keys may hold any number of letters.
- Keys 1, $*$, $\#$, and 0 do not map to any letters.

Your task is to determine the minimum possible number of key presses required to type the string `word` after optimally remapping the keys.

Figure 2: Minimum key pushes LeetCode problem.

two. This decomposition enables scalable verification by evaluating cross-domain consistency rather than isolated accuracy, allowing models to reason locally within each domain while maintaining global coherence across the pipeline. A summary of these domains and their challenges is shown in Table 2.

Table 2: Summary of the three domains in the BRIDGE framework

Domain	Core Role	Main Challenge
Code	Define *how* the program works	Structuring logic in functional form for provability
Specifications	Define *what* the program must do	Avoiding vacuity while matching intended behavior
Proofs	Show the program *meets* its specification	Constructing valid, non-trivial proofs in Lean4

In the remainder of this section we describe each of the domains and give examples of the reasoning content elicited via prompting using the LeetCode problem in Figure 2. See Appendix B for details on the exact prompts used.

3.1 The Lean Code Domain

Implementing correct programs in LEAN4 is uniquely challenging because it serves as both a programming language and a proof assistant grounded in dependent type theory (8). Every definition must be total, terminating, and type-correct, so Lean treats programs as mathematical objects with explicit proof obligations. Even simple recursion requires a structural or well-founded justification, and imperative control patterns often fail to type-check or terminate. In this setting, writing Lean code is itself a form of formal reasoning. Imperative languages such as PYTHON and C++ rely heavily on mutable state and explicit control flow, demanding auxiliary invariants and reasoning about state transitions (14; 11; 29). By contrast, LEAN4 enforces a purely functional core and effectful computations are supported through monads like `IO` and `State`. The Lean ecosystem, including MATHLIB4 (24) and STD, follows this discipline closely—most Lean programs, including large mathematical and algorithmic libraries, are expressed in a functional style.

BRIDGE provides a reasoning-driven approach for scalable Lean code generation. Instead of generating Lean directly, the model first constructs solutions in a functional style inspired by languages such as OCAML and HASKELL, then translates them into Lean syntax. Figure 3 shows an example of the reasoning content.

3.2 The Lean Specifications Domain

If code defines *how* a program operates, specifications define *what* it must guarantee. Formal specifications describe behavioral intent as logical contracts, forming the bridge between informal problem statements and provable implementations. Since the foundational work of Floyd and Hoare (14; 11), this distinction has remained central to verification: correctness means demonstrating that an implementation satisfies its specification for all admissible inputs. In LEAN4, specifications appear as dependent types, pre/postconditions, and invariants that constrain program behavior. While Lean provides automation tools such as `omega` and `lean-smt` for discharging simple goals, verifying realistic specifications still requires explicit reasoning. Poorly structured or vacuous specifications can yield meaningless "correct" proofs, whereas overly rigid contracts may reject valid implementations.

Example (Functional Reasoning): Minimum Key Pushes

We have 8 keys available. Each key position has a cost equal to its index:

$$\text{1st position} \to 1 \text{ push}, \qquad \text{2nd position} \to 2 \text{ pushes}, \qquad \text{etc.}$$

All letters in the word are distinct, so every letter has frequency 1. Thus, to minimize total pushes, we assign letters to the lowest-cost positions:

$$\begin{gathered}\text{First 8 letters} \to 1 \text{ push each},\\ \text{Next 8 letters} \to 2 \text{ pushes each},\\ \text{Next 8 letters} \to 3 \text{ pushes each},\\ \vdots\end{gathered}$$

Haskell-like pseudocode:

```
minimumPushes word =
  let n = length word
      groups = groupsOf8 [0..n-1]
      costs = concatMap (\(gnum, idxs) ->
                map (\_ -> gnum + 1) idxs)
              (zip [0..] groups)
  in sum costs
```

Here, `groupsOf8` divides the indices into blocks of size 8:

$$[0..7],\ [8..15],\ [16..23],\ \ldots$$

Figure 3: Functional reasoning content from Claude 4.

The **BRIDGE** framework allows for different paradigms – including DbC, Property-Based Testing, Test-Driven Development, and Defensive Programming – to leverage additional views on the same task. Figure 4 shows an example of specification reasoning, where the model is prompted to reason about the contracts the implementation must satisfy before solving in Lean4. Detailed comparisons across specification styles and examples of cross-language reasoning are included in Appendix C.

3.3 The Lean Proofs Domain

Finally, proofs provide the formal evidence that what the specifications state must in fact hold. While proof generation is the end goal, a necessary precursor is identifying *meaningful theorem goals*. In most prior work, these goals are the standard verification conditions (VCs) derived from the specification and implementation (e.g., the code satisfies its pre/postconditions and loop invariants). BRIDGE treats theorem discovery as an explicit reasoning step: the model proposes substantive properties such as bounds, invariants, monotonicity, or termination that clarify what must be proved and how it connects to the specification and code. In this work we focus primarily on evaluating proofs as a reasoning strategy for improving code synthesis (the reverse direction is left as future work).

Figure 5 shows an example of the reasoning content produced when the model is prompted to reason about the mathematical properties for the purpose of writing a theorem, before implementing the Lean 4 solution. We can see that it identifies the mathematical properties of the optimal solution. Additional detailed examples and intersection analyses are presented in Appendix D.

3.4 The Semantic Bridge Hypothesis

Large language models bring immense statistical knowledge of code and mathematics, yet verification success depends not only on what they know but on how they *structure* that knowledge during reasoning. Our hypothesis is that formal verification requires an intermediate stage of structured reasoning, a **semantic bridge** that connects the model's internal representations to the logical abstractions expected by proof assistants. Our central hypothesis is that verification requires more than direct translation: it requires **alignment between the model's reasoning paradigm and the mathematical structure of the target system**.

Example (Specification Reasoning): Minimum Key Pushes

Preconditions

1. The input `word` is a string of length $1 \leq |\texttt{word}| \leq 26$, consisting only of lowercase English letters, and all letters are distinct.
2. Exactly eight keys (digits 2–9) are available for remapping.

Postconditions

1. The function returns a positive integer equal to the minimum possible number of key presses needed to type the entire word.
2. No alternative assignment of letters to keys yields a smaller cost.

Invariants

1. Each letter must be mapped to exactly one key.
2. The cost of typing a letter equals its position on the key

Key Insights.
Since all letters occur exactly once, every character has frequency 1. Thus, minimizing cost reduces to filling the cheapest positions first. With eight keys available, the first 8 letters cost 1 push each, the next 8 letters cost 2 pushes each, the next 8 cost 3 pushes each, and so on. This greedy level-by-level placement is globally optimal.

Cost Characterization.
If letters are indexed $i = 0, 1, \ldots, n-1$, then the cost contributed by letter i is

$$\left\lfloor \frac{i}{8} \right\rfloor + 1,$$

so the total minimal cost is

$$\sum_{i=0}^{n-1} \left(\left\lfloor \frac{i}{8} \right\rfloor + 1 \right).$$

Figure 4: Specification reasoning content from Claude 4.

Functional representations, for instance, naturally capture recursion and compositionality; specification-oriented reasoning captures correctness constraints; and theorem-driven reasoning expresses abstract invariants. These are not different languages, but different *modes of representation* that help the model preserve semantic structure as it transitions from problem description to formal proof. BRIDGE leverages these reasoning modes to translate diffuse statistical knowledge into precise, verifiable Lean artifacts. A persistent challenge, however, is data scarcity. While MATHLIB4 offers rich mathematical content, there remains little data linking *code, specifications, and proofs* in a unified way. Meanwhile, LLMs are heavily trained on imperative languages like PYTHON and C++, providing surface familiarity but little exposure to the functional or logical constructs required by Lean. This asymmetry explains why models often fail at direct Lean generation yet perform better when reasoning first through structured, functionally grounded representations.

4 Experimental Setup and Methodology

4.1 Model and Dataset Settings

We evaluate BRIDGE on 178 algorithmic problems curated from LEETCODE, spanning arrays, strings, graphs, dynamic programming, numerical algorithms, and trees. Each problem includes natural language documentation and test suites, enabling meaningful evaluation across all three domains: *Code* (correct Lean4 implementations), *Specifications* (non-vacuous formal contracts), and *Proofs* (constructive proofs linking code to specs).

Five state-of-the-art LLMs are chosen to represent diverse architectures and training regimes: CLAUDE 4.0 SONNET, DEEPSEEK CODER V2,

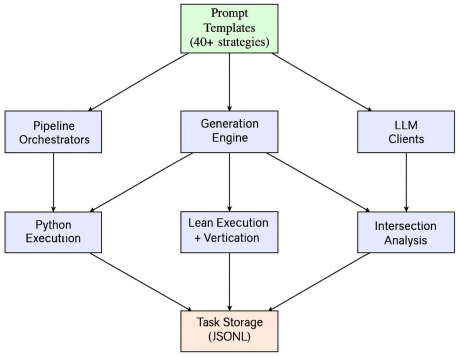


Figure 6: BRIDGE Experimental Architecture

Example (Theorem Reasoning): Minimum Key Pushes

Looking at this problem, we reason about the mathematical properties and theorems that should guide the solution.
Mathematical Properties

1. **Optimality Property**: The minimum pushes are achieved by assigning the most frequent characters to positions requiring fewer pushes.
2. **Greedy Assignment**: Since all characters are distinct (frequency $= 1$), we can assign them greedily to minimize total cost.
3. **Cost Structure**: With 8 keys (2–9), the cost for position i (by assignment order) is $\lceil i/8 \rceil$.
4. **Cost Structure**: Cost increases as we assign more characters (every 8th character increases the push count).

Correctness Conditions

- Each character is assigned to exactly one key position.
- The assignment minimizes total pushes.
- All distinct characters in the input are processed.

Input/Output Relationships

$$\text{For } n \text{ distinct characters:} \quad \texttt{total\_pushes} = \sum_{i=1}^{n} \left\lceil \frac{i}{8} \right\rceil .$$

Invariants

- Total characters processed equals input length.
- Each character contributes at least 1 push.
- Cost assignment follows the ceiling-division pattern.

Figure 5: Theorem reasoning content from Claude 4.

DEEPSEEK R1, LLAMA-3.1-70B, and QWEN2.5 CODER. All are run with consistent decoding parameters (temperature 0.7, max output tokens 4096).

4.2 Reasoning Strategies and Evaluation Design

We systematically vary reasoning strategies to test the semantic bridge hypothesis. For the *Code* domain, we compare

- direct natural language→Lean4 generation
- reasoning functionally (e.g., in HASKELL, OCAML)
- reasoning imperatively (PYTHON, C++, JAVA)

For *Specifications*, we explore multiple frameworks as reasoning strategies for producing Lean4 specifications, following and extending the methodology in VERINA:

- Design by Contract,
- Property-Based Testing,
- Test-Driven Development, and
- Defensive Programming before translating.

For *Proofs*, we test four pathways

- NL→Code+Proofs
- Unit Tests→Code+Proofs
- Termination Reasoning→Code+Proofs
- Documentation→Code+Proofs

and perform intersection analysis to identify robust correctness proofs. We evaluate code by test pass rates, specifications by vacuity detection and coverage metrics, and proofs by proof validity in LEAN4. To ensure robustness, we also apply this suite of reasoning strategies to the VERINA benchmark (37). In particular, we compare functional versus imperative intermediates for code generation, employ multiple specification frameworks for spec generation, and evaluate multiple proof generation reasoning strategies.

5 Experimental Results and Analysis

5.1 Thinking "Functionally" is a Win For All Models

We find strong evidence for our central hypothesis: functional reasoning paradigms consistently outperform imperative approaches across all tested models and enhancement strategies. As shown in Figure 7, they achieve higher success rates on Lean4 code synthesis across the board, underscoring their structural advantages for formal verification. For example, CLAUDE 4.0 SONNET reached 48.9% success with functional reasoning (vs. 44.9% imperative, 42.1% direct). This advantage holds across diverse models—DEEPSEEK-R1 (16.3% vs. 14.6%), DEEPSEEK-CODER-V2 (3.4% vs. 3.4%), LLAMA-3.1-70B (6.2% vs. 5.6%), and QWEN2.5-CODER-7B (1.1% vs. 0.6%), showing that functional paradigms align more naturally with the structural requirements of Lean-based verification.

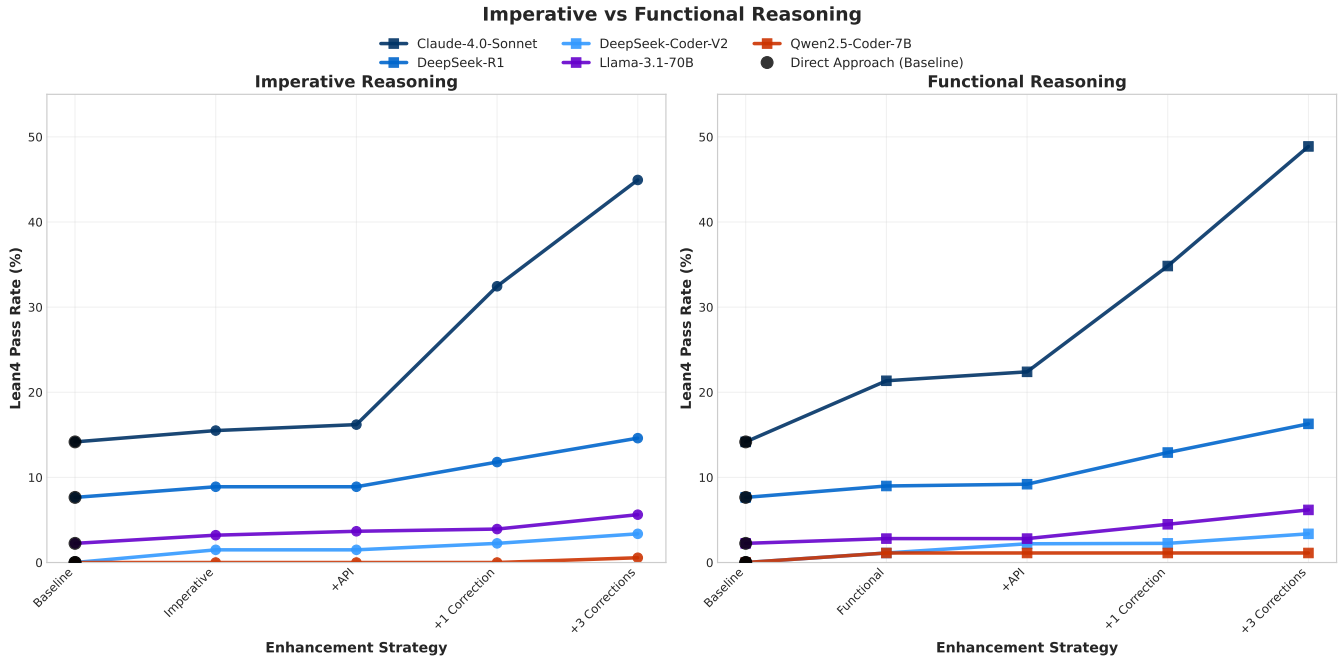


Figure 7: Imperative vs Functional Reasoning. Line graphs show each model's pass@5 performance trajectory across enhancement strategies, with functional paradigms consistently outperforming imperative approaches.

As shown in Figure 1, functional reasoning achieves up to 2x greater efficiency when combined with error-correction and nearly 4x higher verification success compared to direct pass@64 baselines, even at comparable generation lengths ($\approx 573 - 603$ tokens). The widening gap is not driven by verbosity: functional, imperative, and direct generations are similarly long. Functional reasoning continues to improve beyond pass@128, revealing a higher asymptotic ceiling and greater representational stability, while imperative and direct methods plateau early due to accumulated inconsistencies. The functional paradigm advantages remain stable across temperature settings (0.5-0.9): OCaml and Haskell consistently reach 20-22% success while maintaining 80-100% improvements over baseline approaches (Figure 11 in Appendix A). Error pattern analysis further shows that functional reasoning reduces critical failure modes (i.e., syntax errors) from 45% to 12%, type errors decrease by 20%, and termination failures fall from 15% to 8%, reinforcing that structured reasoning patterns improve verifiable code synthesis.

5.1.1 Inference Time Compute by Reasoning Strategy

Studying the relationship between reasoning verbosity and functional correctness reveals that increased inference-time compute is not the primary driver of improved performance. To quantify this, Table 3 measures average word and token counts (using

the CLAUDE 4.0 SONNET tokenizer) for successful and failed attempts under each reasoning strategy. All reasoning strategies consistently have longer generation lengths (use more FLOPs), with OCAML responses averaging 563 tokens compared to only 270 for direct translation. Double Lean responses average 555 tokens, closely matching functional reasoning, yet achieve only 15% success compared to 21% for OCAML. This shows that mere verbosity does not explain functional advantages—structured reasoning drives the gains beyond raw reasoning effort.

Table 3: Generation lengths by reasoning strategy

Reasoning Strategy	Direct (NL→Lean)	Double Lean Reasoning	Python Reasoning
Average Length (words / tokens)	195 / 270	402 / 555	382 / 526
Success Length (words / tokens)	141 / 196	347 / 480	324 / 446
Failure Length (words / tokens)	200 / 278	412 / 570	391 / 540
Reasoning Strategy	**Haskell Reasoning**	**C++ Reasoning**	**OCaml Reasoning**
Average Length (words / tokens)	387 / 534	390 / 538	408 / 563
Success Length (words / tokens)	334 / 460	341 / 469	343 / 473
Failure Length (words / tokens)	401 / 555	402 / 558	421 / 581

5.2 Specification Reasoning Improves Code Synthesis

We find that specification reasoning strategies provide benefits extending beyond formal verification to general code generation. As shown in Figure 8, these strategies improve performance across both Python and Lean4 domains, demonstrating their broader value as a cognitive scaffold for program synthesis.

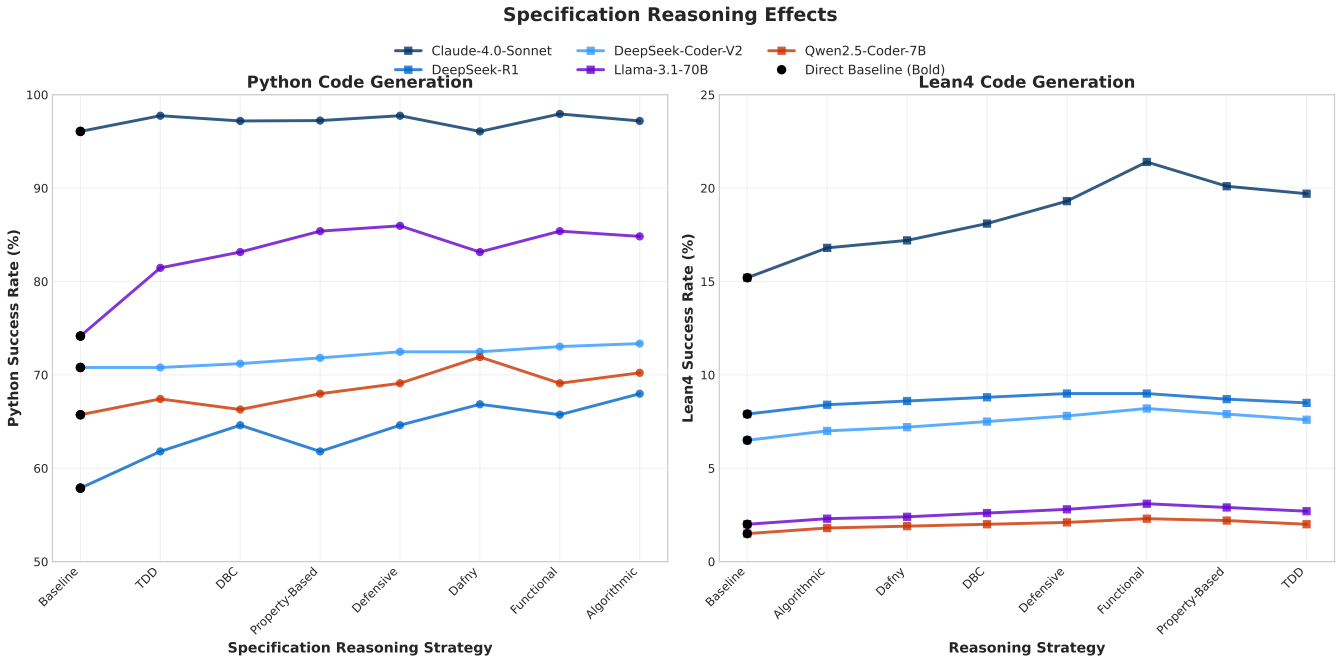


Figure 8: Dual improvement analysis showing that specification reasoning enhances both Python code generation (left) and Lean4 formal verification (right). All models benefit from specification-guided reasoning over direct baseline approaches.

In Python code generation, specification reasoning yields clear gains. Claude 4.0 Sonnet improves from 96.1% to 97.9%, while weaker models see larger boosts: DeepSeek-R1 (57.9%→68.0%, +17.5%), DeepSeek-Coder-V2 (70.8%→73.4%, +3.6%), Llama-3.1-70B (74.2%→86.0%, +15.9%), and Qwen2.5-Coder-7B (65.7%→71.9%, +9.4%). This shows that reasoning about intent before implementation improves code accuracy even outside formal verification frameworks. We also observe that certain specification paradigms are consistently more effective. Algorithmic reasoning achieves an average success rate of 78.7% in Python generation, while Dafny-style formal specifications reach 78.1% and Defensive Programming 77.6%. This clustering of high-performing strategies suggests that structured specification design plays a key role in guiding implementation quality, independent of the target language or framework (see Appendix C for a comprehensive strategy analysis).

5.2.1 Different Strategies produce Semantically Different Code

To understand how different reasoning strategies affect generated Python code, we embedded all implementations using Qwen2.5-Coder embeddings and visualized them in two-dimensional space using t-SNE projection.

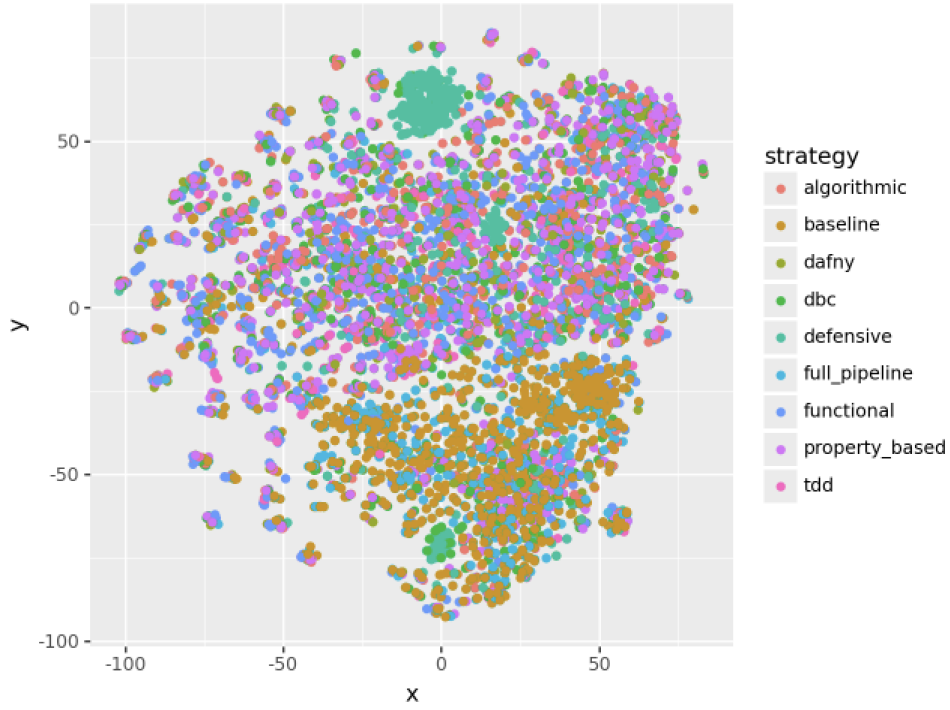


Figure 9: Code Embedding Analysis: Generated Python implementations cluster by strategy, with baseline (orange) showing distinct separation from other approaches.

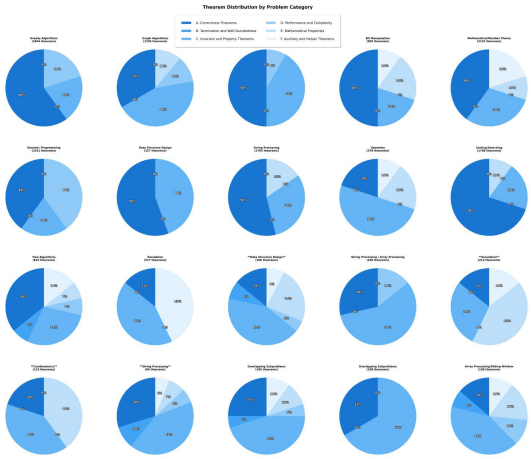
Figure 9 shows that most intermediate reasoning strategies cluster together in semantic space, including Design by Contract, defensive programming, functional reasoning, and property-based testing. The baseline strategy appears as scattered orange points, often separated from these clusters, indicating that direct translation produces qualitatively different implementations. Semantic convergence across strategies supports our cognitive scaffolding theory and the consistent clustering of non-baseline approaches confirms that *thinking differently* e.g., via structured reasoning results in meaningfully *different* code.

5.3 Proof Generation: Multi-Pathway Analysis

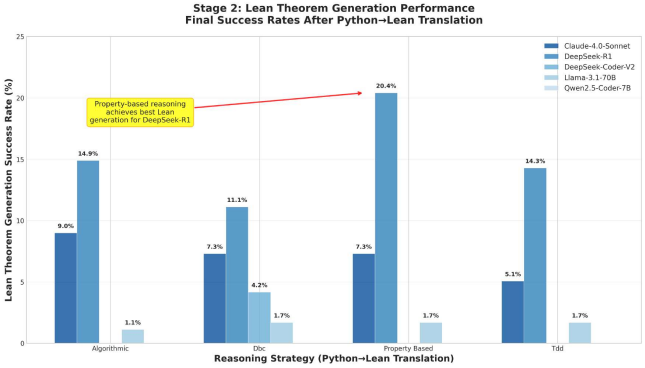
Our multi-pathway intersection analysis reveals the complexity of mathematical proof discovery and highlights the core challenges models face when moving from informal reasoning to formal proofs. Figure 10 illustrates these bottlenecks and performance patterns across different reasoning strategies.

We observe that models perform extremely well on functional correctness when reasoning about proofs in intermediate Python representations (97-98% success) but struggle to formalize these insights in LEAN4. For instance, Claude 4.0 Sonnet maintains only 9% success when converting its Python-level proofs into formal Lean proofs, exposing a critical gap between informal mathematical understanding and formal expression.

Despite this difficulty, cross-strategy intersection analysis reveals that models consistently rediscover certain universal proof categories. Additionally, we perform a meta-analysis combining automated clustering with CLAUDE 4.0 and manual annotation to categorize proof strategies employed. Bounds proofs appear in 78% of problems, monotonicity proofs in 65%, invariant preservation in 52%, optimality proofs in 43%, and termination properties in 89% of recursive problems. These recurring patterns suggest that models can reliably identify fundamental correctness properties even when they struggle to express them in formal proof syntax (see Appendix D for detailed analysis and examples).



(a) Direct reasoning and pipeline bottleneck analysis



(b) Final success rates by reasoning strategy

Figure 10: Proof Generation Analysis: The left panel shows direct proof reasoning performance and reveals a significant bottleneck between Python generation (97-98% success) and Lean translation (9–20% success).

6 Conclusion and Future Work

We introduced BRIDGE, a framework that reframes automated formal verification as achieving *semantic consistency* across three interdependent domains; code, specifications, and proofs, rather than as a single leap from natural language to proof. By using structured intermediate representations, BRIDGE preserves mathematical structure as models move between abstraction levels, aligning their reasoning with the logical foundations of verification frameworks like Lean4. Experiments on five state-of-the-art LLMs and 178 algorithmic problems show that this structured approach yields consistent gains: functional paradigms cut syntax and termination errors, improving functional correctness up to 1.5x. We also found that specification reasoning boosts Python functional correctness by up to 17.5%. Using this foundation, in future work we intend to investigate post-training approaches such as reinforcement learning from verifiable rewards to train models that not only generate code but also *specify and prove* its correctness.

References

[1] AUSTIN, J., ODENA, A., NYE, M., BOSMA, M., MICHALEWSKI, H., DOHAN, D., JIANG, E., CAI, C., TERRY, M., LE, Q. ET AL. (2021). Program synthesis with large language models. `arXiv:2108.07732`.

[2] BARNETT, M., CHANG, B.-Y. E., DELINE, R., JACOBS, B. and LEINO, K. R. M. (2005). Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*. Springer, 364–387.

[3] BERTOT, Y. and CASTÉRAN, P. (2013). *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Science & Business Media.

[4] BOBOT, F., FILLIÂTRE, J.-C., MARCHÉ, C. and PASKEVICH, A. (2011). Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*.

[5] BURSUC, S., EHRENBORG, T., LIN, S., ASTEFANOAEI, L., CHIOSA, I. E., KUKOVEC, J., SINGH, A., BUTTERLEY, O., BIZID, A., DOUGHERTY, Q., ZHAO, M., TAN, M. and TEGMARK, M. (2025). A benchmark for vericoding: Formally verified program synthesis. `arXiv:2509.22908`.

[6] CHEN, M., TWOREK, J., JUN, H., YUAN, Q., DE OLIVEIRA PINTO, H. P., KAPLAN, J., EDWARDS, H., BURDA, Y., JOSEPH, N., BROCKMAN, G., RAY, A., PURI, N., KRUEGER, G., PETROV, M., KHLAAF, H., SASTRY, G., MISHKIN, P., CHAN, B., GRAY, S., RYDER, N., PAVLOV, M., POWER, B., KAISER, L., BAVARIAN, M., WINTER, C., TILLET, P., SUCH, F. P., CUMMINGS, A., PLAPPERT, M., CHANTZIS, F., BARNES, E., HERBERT-VOSS, A., GUSS, W., NICHOL, A., PAINO, I., TEZAK, N., TANG, J., BABUSCHKIN, I., BALAJI, S., JAIN, S., SAUNDERS, W., HESSE, C., CARR, A., LEIKE, J., ACHIAM, J., MISRA, V., MORIKAWA, E., RADFORD, A., KNIGHT, M., BRUNDAGE, M., MURATI, M., MAYER, S., WELINDER, P., MCGREW, B., AMODEI, D. and SUTSKEVER, I. (2021). Evaluating large language models trained on code. `arXiv:2107.03374`.

[7] CLARKE, E. M. and WING, J. M. (1996). Formal methods: State of the art and future directions. *ACM Computing Surveys* **28** 626–643.

[8] DE MOURA, L., KONG, S., AVIGAD, J., VAN DOORN, F. and VON RAUMER, J. (2015). The lean theorem prover (system description). In *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction*. Springer.

[9] DIJKSTRA, E. W. (1976). *A discipline of programming*. Prentice Hall.

[10] FENG, Z., GUO, D., TANG, D., DUAN, N., FENG, X., GONG, M., SHOU, L., QIN, B., LIU, T., JIANG, D. and ZHOU, M. (2020). Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*.

[11] FLOYD, R. W. (1967). Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*, vol. 19. American Mathematical Society, 19–32.

[12] GAO, L., MADAAN, A., ZHOU, S., ALON, U., LIU, P., YANG, Y., CALLAN, J. and NEUBIG, G. (2022). Pal: Program-aided language models. arXiv:2211.10435.

[13] GUO, D., REN, S., LU, S., FENG, Z., TANG, D., DUAN, N., SVYATKOVSKIY, A., FU, S., TUFANO, M., DENG, S. K., CLEMENT, C., DRAIN, D., SUNDARESAN, N., YIN, J., JIANG, D. and ZHOU, M. (2021). Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR)*.

[14] HOARE, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM* **12** 576–580.

[15] HOARE, C. A. R. (1971). Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*.

[16] HUDAK, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys* **21** 359–411.

[17] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M. ET AL. (2009). sel4: Formal verification of an operating-system kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM.

[18] LATTUADA, A., HANCE, T., CHO, C., BRUN, M., SUBASINGHE, I., ZHOU, Y., HOWELL, J., PARNO, B. and HAWBLITZEL, C. (2023). Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.* **7**.

[19] LEINO, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer.

[20] LEROY, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM* **52** 107–115.

[21] LOHN, E. and WELLECK, S. (2024). minicodeprops: a minimal benchmark for proving code properties. arXiv:2406.11915.

[22] LOUGHRIDGE, C., SUN, Q., AHRENBACH, S., CASSANO, F., SUN, C., SHENG, Y., MUDIDE, A., MISU, M. R. H., AMIN, N. and TEGMARK, M. (2024). Dafnybench: A benchmark for formal software verification. arXiv:2406.08467.

[23] MADAAN, A., TANDON, N., GUPTA, P., HALLINAN, S., GAO, L., WIEGREFFE, S., ALON, U., DZIRI, N., PRABHUMOYE, S., YANG, Y., GUPTA, S., MAJUMDER, B. P., HERMANN, K., WELLECK, S. and CLARK, P. (2023). Self-refine: Iterative refinement with self-feedback. arXiv:2303.17651.

[24] MATHLIB COMMUNITY, T. (2020). The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, New Orleans, LA, USA.

[25] MIRANDA, B., ZHOU, Z., NIE, A., OBBAD, E., ANIVA, L., FRONSDAL, K., KIRK, W., SOYLU, D., YU, A., LI, Y. and KOYEJO, S. (2025). Veribench: End-to-end formal verification benchmark for ai code generation in lean 4. In *AI4Math@ICML25*.

[26] SHINN, N., CASSANO, F., BERMAN, E., GOPINATH, A., NARASIMHAN, K. and YAO, S. (2023). Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*.

[27] SUN, C., SHENG, Y., PADON, O. and BARRETT, C. (2024). Clover: Closed-loop verifiable code generation. arXiv:2310.17807.

[28] THAKUR, A., LEE, J., TSOUKALAS, G., SISTLA, M., ZHAO, M., ZETZSCHE, S., DURRETT, G., YUE, Y. and CHAUDHURI, S. (2025). Clever: A curated benchmark for formally verified code generation. arXiv:2505.13938.

[29] VIDELA, A. (2018). Are functional programs easier to verify?
URL https://semantic-domain.blogspot.com/2018/04/are-functional-programs-easier-to.html

[30] WANG, R., ZHANG, J., JIA, Y., PAN, R., DIAO, S., PI, R. and ZHANG, T. (2024). Theoremllama: Transforming general-purpose llms into lean4 experts. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Miami, Florida, USA.

[31] WANG, X., WEI, J., SCHUURMANS, D., LE, Q. V., CHI, E. H., NARANG, S., CHOWDHERY, A. and ZHOU, D. (2022). Self-consistency improves chain of thought reasoning in language models. arXiv:2203.11171.

[32] WEI, J., WANG, X., SCHUURMANS, D., BOSMA, M., ICHTER, B., XIA, F., CHI, E. H., LE, Q. V. and ZHOU, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. arXiv:2201.11903.

[33] WEN, C., CAO, J., SU, J., XU, Z., QIN, S., HE, M., LI, H., CHEUNG, S.-C. and TIAN, C. (2024). Enchanting program specification synthesis by large language models using static analysis and program verification. arXiv:2404.00762.

[34] XU, X., LI, X., QU, X., FU, J. and YUAN, B. (2025). Local success does not compose: Benchmarking large language models for compositional formal verification. arXiv:2509.23061.

[35] YAO, S., YU, D., ZHAO, J., SHAFRAN, I., GRIFFITHS, T. L., CAO, Y. and NARASIMHAN, K. (2023). Tree of thoughts: Deliberate problem solving with large language models. arXiv:2305.10601.

[36] YAO, S., ZHAO, J., YU, D., SHAFRAN, I., NARASIMHAN, K. and CAO, Y. (2023). React: Synergizing reasoning and acting in language models. In *ICLR*.

[37] YE, Z., YAN, Z., HE, J., KASRIEL, T., YANG, K. and SONG, D. (2025). Verina: Benchmarking verifiable code generation. arXiv:2505.23135.

A Extended Analysis

A.1 Literate Programming for Formal Verification

Literate programming presents programs as readable mathematical exposition with embedded, executable code. In this section we ask whether large language models can solve verification problems in that style. Rather than emitting Lean code only, we prompt the model to produce a short manuscript that explains the problem, gives the mathematical context, derives the algorithm, and presents an executable Lean implementation together with its verification status. The goal is to make the generated artifacts useful to both people and proof checkers, and to test whether a more formal narrative helps the model organize its reasoning.

Study design and setup. We evaluated three concisely defined styles using CLAUDE 4.0 SONNET. The *tutorial* style prioritizes pedagogical explanation, the *proof* style foregrounds formal justification, and the *mathematical* style adopts a compact, theorem oriented exposition. Each manuscript follows the same structure: brief abstract and problem restatement, minimal mathematical context, Lean implementation, and a short verification note.

Table 4: Literate programming styles and verification outcomes on 178 tasks

Style	Verified Works	Pass Rate
Tutorial	8	4.5%
Proof	14	**7.9%**
Mathematical	13	7.3%

The proof style attained the highest verification rate, which suggests that asking the model to think in terms of claims and justifications can improve the precision of its implementations. The mathematical style was a close second. The tutorial style was most readable but least successful, which indicates that explanations optimized for novice readability do not always align with the logical structure that Lean requires.

A compact example. Below is a shortened example that illustrates the format. We present a brief narrative followed by a Lean function. The full listing is omitted for space.

```
/-
Problem: decide if two length-4 strings can be made equal by swapping characters in each
    ↪ pair of even and odd indices.

Idea: characterize the relation as equality up to permutation within {0,2} and within
    ↪ {1,3}.
We then implement the decision procedure and record a short verification note.
-/

import Std

def canBeEqual (s1 s2 : String) : Bool :=
  let a := s1.data; let b := s2.data
  if a.length != 4 || b.length != 4 then false
  else
```

```
    let evenOk :=
      (a[0]! = b[0]! ∧ a[2]! = b[2]!) ∨ (a[0]! = b[2]! ∧ a[2]! = b[0]!)
    let oddOk  :=
      (a[1]! = b[1]! ∧ a[3]! = b[3]!) ∨ (a[1]! = b[3]! ∧ a[3]! = b[1]!)
    decide (evenOk ∧ oddOk)

/- Verification note: passes provided tests; types check; no partial definitions.
   Full manuscript and extended proofs are omitted due to space. -/
```

Listing 1: Literate solution skeleton, truncated for brevity

Findings and implications. The style that encourages the model to express claims and sketch justifications produced more verifiable Lean, even when the proofs were brief. This supports the broader result that representation matters: when the model articulates invariants and intended properties, the subsequent implementation is more likely to be precise and checkable. Literate generation also yields human readable artifacts that can be refined into full proofs, which makes it a useful bridge between informal reasoning and formal verification. Future work includes interactive notebooks that couple Lean code, minimal proofs, and concise exposition, and a study of which narrative prompts best support proof discovery.

A.2 Temperature Robustness Analysis

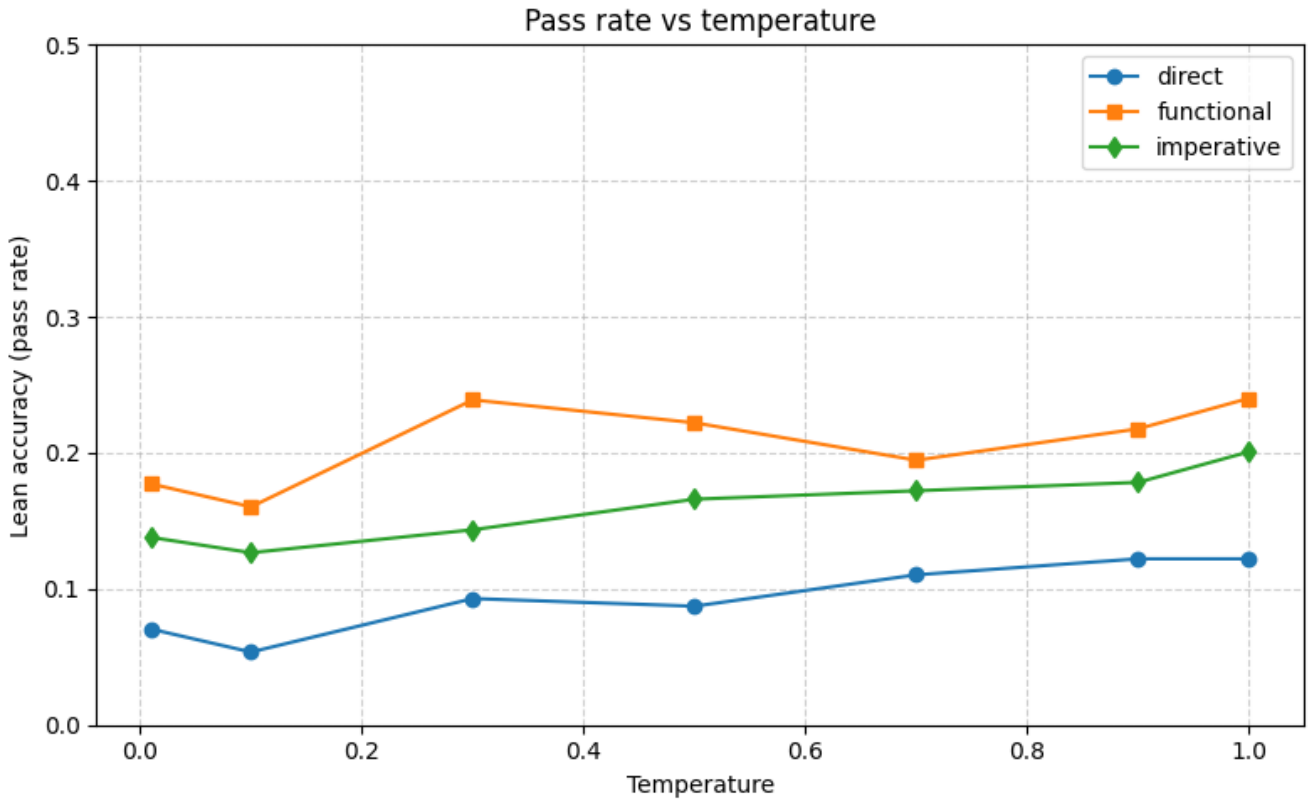


Figure 11: Temperature robustness of reasoning paradigms. Pass@5 accuracy of Claude 4.0 Sonnet as a function of sampling temperature for three prompting strategies: direct natural-language→Lean (blue), imperative Python-bridge (green), and functional OCaml-bridge (orange).

Functional paradigms remain consistently stronger across all temperatures. Their success rates improve slightly with higher temperatures, suggesting robust structured reasoning benefits independent of sampling randomness.

A.3 Error-Informed Python Code Refinement Pipeline

Understanding how error feedback from downstream verification can improve upstream code generation reveals critical dynamics in iterative refinement. We implemented a multi-stage pipeline where Python generation benefits from both retry mechanisms

($k = 3$ attempts) and cross-domain error signals from LEAN4 verification. These Lean-derived errors improve Python quality even when Lean translation ultimately fails. QWEN2.5-CODER gains the most from Lean feedback (+3.9% in the functional paradigm), while DEEPSEEK-CODER-V2 benefits mainly from intermediate retries (+2.9%). CLAUDE 4.0 SONNET, already near ceiling (97–98%), shows minimal gains, indicating diminishing returns for stronger models.

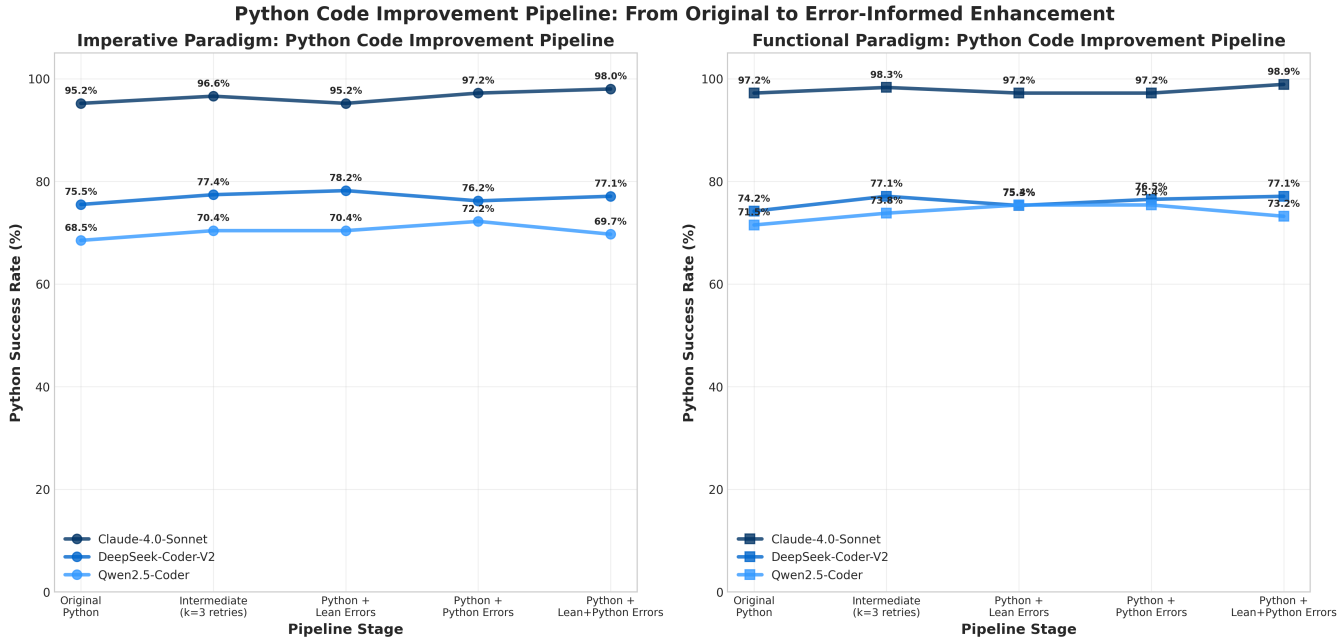


Figure 12: Python code success rates across improvement stages. Functional paradigms consistently outperform imperative ones, and incorporating Lean error feedback boosts Python success, though the gap to final verification remains large.

Functional paradigms consistently outperform imperative ones across all pipeline stages, maintaining this advantage even with error feedback. Combining Python and Lean error signals yields the strongest results, with CLAUDE 4.0 SONNET reaching 98.9% success. However, the steep drop from high Python success (70–98%) to low Lean verification rates (0–14%) highlights a persistent bottleneck: the core challenge lies not in code quality but in translating algorithmic solutions into formally verified proofs.

A.4 Error Pattern Analysis Across Paradigms

Systematic analysis of failure modes across different reasoning approaches exposes where structured reasoning provides specific cognitive advantages. The error distribution patterns illuminate why functional paradigms excel in formal verification tasks.

Syntax errors represent the most immediate failure mode, occurring in 45% of direct translation attempts due to incorrect Lean4 syntax and missing type annotations. Functional reasoning approaches reduce syntax errors to 12%, showing better alignment with Lean4's mathematical syntax patterns. Functional paradigms naturally guide models toward syntactic structures compatible with dependently typed programming languages.

Type errors account for 30% of overall failures, with functional approaches showing 20% fewer type errors. Common issues include incorrect dependent type usage and mismatched function signatures. The reduced incidence in functional approaches confirms that functional reasoning patterns align naturally with Lean4's type theoretic foundations, where types can encode mathematical properties and proof obligations.

Most significantly, termination and totality failures occur in 15% of imperative approaches versus only 8% for functional approaches. Functional reasoning naturally leads to structurally recursive solutions that Lean4 can automatically prove terminating, while imperative approaches often produce iterative algorithms requiring complex termination arguments that current models struggle to provide. This pattern directly supports our theoretical claim that functional paradigms provide cognitive scaffolding aligned with mathematical proof construction requirements.

A.5 Termination Error Analysis and Functional Paradigm Benefits

Understanding termination failures provides crucial insights into why functional paradigms excel in formal verification. Our analysis across multiple models and reasoning strategies reveals that functional approaches provide measurable benefits in reducing termination-related failures during Lean4 code generation.

The Python→Lean strategy consistently demonstrates lower termination error rates compared to direct translation approaches. Claude 4.0 Sonnet shows a 10.8% reduction in termination errors (29.88% vs 40.68% for direct Lean generation), while the Double Lean reasoning strategy achieves an average termination error rate of 33.5%. This pattern suggests that intermediate functional reasoning helps models develop structurally recursive solutions that align naturally with Lean4's termination checking requirements. However, we observe an interesting trade-off between verification success rates and termination error management. Higher-performing strategies like OCaml→Lean achieve the best pass rates (21.35% for Claude 4.0 Sonnet) but encounter significant termination errors (34.57%). This suggests that sophisticated reasoning approaches tackling complex verification challenges naturally encounter more termination-related difficulties, but the overall success rate improvement outweighs these specific failures.

Model capability remains the primary determinant of success, with weaker models showing both poor pass rates (0-2%) and high termination error rates (28-49%) regardless of reasoning strategy. This indicates that while functional reasoning strategies provide structural benefits for termination management, underlying model capability determines whether these benefits can be effectively utilized.

A.6 Proof Generation Impact on Code Quality

Meta-analysis of formal proof reasoning within the BRIDGE Framework demonstrates measurable benefits for Lean4 code synthesis. We compared two distinct approaches: Strategy 1 involves reasoning about proofs from natural language descriptions before generating code, while Strategy 2 employs direct natural language to Lean4 translation without explicit proof consideration.

Strategy 1's explicit proof-driven approach achieves higher overall success (14.6% vs. 10.1%), indicating that formal reasoning scaffolding improves implementation quality. When both strategies succeed on the same problems, their solutions share over 95% algorithmic similarity, suggesting that proof generation both verifies correctness and guides the discovery of good algorithms. The two strategies also exhibit complementary strengths: Strategy 1 performs best on mathematically heavy tasks (e.g., geometry, combinatorial counting, bit manipulation), where explicit proofs provide the necessary scaffolding for correct Lean4 implementations, while Strategy 2 does better on dynamic programming and stateful computations, where full formal reasoning is less critical. Overall, these patterns support our three-domain view of proof generation as serving two roles: it enforces correctness by pinning down key mathematical properties, and it acts as an algorithmic discovery tool that systematically filters out incorrect solution paths.

B Comprehensive Prompt System for BRIDGE

Our BRIDGE framework uses a modular prompt system to evaluate structured reasoning approaches across the three domains of formal verification. Over 40 strategies are organized into three families: *Code Domain* (Lean4 implementations), *Specification Domain* (formal contracts), and *Proof Domain* (mathematical property discovery). We present unified templates with strategy-specific content marked as `[SWAP]` placeholders.

B.1 Code Domain Strategies (Lean4)

The Code Domain tests our hypothesis that reasoning strategies affect LEAN4 generation success. We use a unified template with 6 reasoning strategies:

Unified Code Domain Template

Instructions: Solve the task by applying the reasoning paradigm in `[SWAP:STRATEGY]` and translate to Lean4.
[SWAP:STRATEGY] options:

- **Direct**: Solve directly in Lean4
- **Python Bridge**: Reason in Python logic first, then translate
- **Haskell Functional**: Use pattern matching, recursion, type-driven development
- **OCaml Type-Guided**: Emphasize type safety and mathematical invariants
- **Double Lean**: Reason mathematically in Lean style, then implement
- **C++ Imperative**: Use procedural logic, then translate to functional Lean4

```
Step 1:  [SWAP:REASONING_STYLE] — [SWAP:REASONING_DETAILS]
Step 2:  Lean4 Translation:
```

```
import Std
import Mathlib

def solution (input : InputType) : OutputType :=
  -- Implementation from [SWAP:STRATEGY]
  sorry
```

B.1.1 Strategy-Specific Reasoning Details

Imperative Paradigms (C++ / Python)

[SWAP:REASONING_DETAILS] for C++:

- Loops, arrays, pointers, algorithmic thinking
- Data structures (vectors, maps, sets)
- Memory management, efficiency, complexity

[SWAP:REASONING_DETAILS] for Python:

- Python-like logic, built-ins, simple readable solutions

Functional Paradigms (Haskell / OCaml/ Double Lean)

[SWAP:REASONING_DETAILS] for Haskell:

- Pattern matching, recursion, functional composition
- List operations, higher-order functions, immutable data
- Type signatures and algebraic data types

[SWAP:REASONING_DETAILS] for OCaml:

- Pattern matching, recursion, immutable data structures
- Algebraic data types, variants, record types
- Tail recursion, functional composition, rich standard library

[SWAP:REASONING_DETAILS] for Double Lean:

- Mathematical properties, invariants
- Inductive definitions, recursive structures
- Proof strategies and type theory

B.2 Specification Domain Strategies (Python)

Unified Specification Domain Template

Instructions: Solve this problem using `[SWAP:STRATEGY]` reasoning, then implement in Python with formal contracts.
[SWAP:STRATEGY] options:

- **Direct**: Write Python solution directly without intermediate reasoning steps
- **Design by Contract**: Systematically define preconditions, postconditions, and invariants using DbC methodology
- **Dafny-style**: Use formal specification with requires/ensures clauses and verification conditions
- **Property-Based Testing**: Identify universal mathematical properties that must hold for all valid inputs
- **Functional Programming**: Use immutable data paradigms and pure functions for mathematical correctness
- **Defensive Programming**: Implement comprehensive input validation and robust error handling patterns
- **Algorithmic Thinking**: Break down into algorithmic subproblems with complexity analysis
- **Test-Driven Development**: Design by reasoning about comprehensive test cases first

`Step 1: [SWAP:REASONING_APPROACH]` — Apply strategy-specific reasoning methodology

`Step 2: Python Implementation` — Translate reasoning to Python with formal contracts

```
**APPROACH:** First reason using [SWAP:STRATEGY] principles, then implement in Python.

# Solution Process
**Step 1: [SWAP:STRATEGY] Reasoning**
[SWAP:DETAILED_REASONING_GUIDELINES]

**Step 2: Python Implementation**
Apply your [SWAP:STRATEGY] reasoning to create robust Python code with contracts

# Output Format
<python>
[SWAP:CONTRACT_DECORATORS]
def function_name(parameters):
    """[SWAP:DOCSTRING_WITH_CONTRACTS]"""
    # Implementation guided by [SWAP:STRATEGY] reasoning
    pass
</python>
```

B.2.1 Concrete Prompt Examples for Each Strategy

Prompt — Design by Contract Reasoning

Reasoning Focus: Define rigorous preconditions, postconditions, and invariants that govern function behavior. This approach emphasizes formal contracts that specify exactly what conditions must hold before and after execution.

```
import deal
from typing import List

@deal.pre(lambda data: isinstance(data, list) and len(data) > 0)
@deal.pre(lambda data: all(isinstance(x, int) for x in data))
@deal.post(lambda result: result >= 0)
@deal.ensure(lambda data, result: result <= sum(x for x in data if x > 0))
def contract_solution(data: List[int]) -> int:
    """
    Precondition: non-empty list of integers
    Postcondition: result >= 0 and bounded by positive sum
    Invariant: input data structure remains unchanged
    """
    assert isinstance(data, list) and len(data) > 0
    assert all(isinstance(x, int) for x in data)
    result = sum(x for x in data if x >= 0)
    assert result >= 0
    return result
```

Prompt — Dafny-Style Specification Reasoning

Reasoning Focus: Apply formal verification principles with requires/ensures clauses and mathematical assertions. This strategy emphasizes rigorous specification techniques from the Dafny verification language.

```
import deal
from typing import List

@deal.chain(
    deal.pre(lambda data: all(isinstance(x, int) for x in data)),
    deal.pre(lambda data: data == sorted(data)),
    deal.post(lambda result: result == sorted(result)),
    deal.ensure(lambda data, result: all(x >= 0 for x in result))
)
def dafny_style_solution(data: List[int]) -> List[int]:
    """
    Requires: input sorted list of ints
    Ensures: output sorted and non-negative
    Invariant: sorted order maintained throughout transformation
    """
    assert all(isinstance(x, int) for x in data)
    assert data == sorted(data)
    result = [abs(x) for x in data]
    assert result == sorted(result)
    return result
```

Prompt — Property-Based Testing Reasoning

Reasoning Focus: Identify universal mathematical properties that must hold across all valid input domains. This approach emphasizes discovering and testing algebraic properties like commutativity, associativity, and invariants.

```
import deal
from hypothesis import given, strategies as st
from typing import List

@deal.chain(
    deal.pre(lambda data: isinstance(data, list)),
    deal.post(lambda result: result >= 0),
    deal.ensure(lambda data, result: result <= sum(abs(x) for x in data))
)
def property_solution(data: List[int]) -> int:
    """
    Universal Properties:
    - Commutativity: Order independence of input elements
    - Monotonicity: Adding positive elements increases result
    - Non-negativity: Result always >= 0 regardless of input
    """
    return sum(x for x in data if x >= 0)
```

Prompt — Test-Driven Development Reasoning

Reasoning Focus: Design solutions by reasoning about comprehensive test cases and validation requirements first. This strategy follows the Red-Green-Refactor cycle, emphasizing test coverage and incremental development.

```
@deal.chain(
    deal.pre(lambda data: isinstance(data, list)),
    deal.post(lambda result: isinstance(result, int) and result >= 0)
)
def tdd_solution(data: List[int]) -> int:
    """
    TDD Design Process:
    - Red Phase: Tests designed to validate specific behaviors
    - Green Phase: Minimal implementation to pass all tests
    - Refactor Phase: Optimize while maintaining test passage
    """
    # Implementation designed to pass comprehensive test suite
    if not isinstance(data, list) or len(data) == 0:
        return 0  # Handles edge case tests

    positive_sum = sum(x for x in data if isinstance(x, int) and x > 0)
    return positive_sum

# Test suite that guides implementation
def comprehensive_test_suite():
    assert tdd_solution([1, 2, 3]) == 6      # Basic functionality
    assert tdd_solution([]) == 0             # Empty input edge case
    assert tdd_solution([-1, -2]) == 0       # All negative numbers
    assert tdd_solution([1, "2", None]) == 1 # Mixed types handling
```

Prompt — Defensive Programming Reasoning

Reasoning Focus: Implement comprehensive input validation, error handling, and fail-safe behaviors. This strategy emphasizes robustness through systematic validation and graceful error recovery mechanisms.

```
@deal.chain(
    deal.safe,  # Never raises exceptions
    deal.post(lambda result: isinstance(result, int) and result >= 0)
)
def defensive_solution(data: Union[List[int], None]) -> int:
    """
    Defensive Strategies:
    - Comprehensive input validation with type checking
    - Graceful handling of malformed inputs and edge cases
    - Exception isolation with safe fallback behaviors
    """
    if data is None or not isinstance(data, list):
        logging.warning("Invalid input, using safe default")
        return 0

    try:
        valid_ints = [x for x in data if isinstance(x, int)]
        result = sum(x for x in valid_ints if x >= 0)
        return result
    except Exception as e:
        logging.error(f"Computation error: {e}")
        return 0  # Safe fallback
```

Prompt — Algorithmic Thinking Reasoning

Reasoning Focus: Systematically decompose problems into algorithmic subcomponents with complexity analysis. This approach emphasizes structured problem breakdown, optimal data structure selection, and efficiency considerations.

```
import deal
from typing import List
from functools import lru_cache

@deal.chain(
    deal.pre(lambda data: isinstance(data, list)),
    deal.post(lambda result: isinstance(result, int) and result >= 0)
)
def algorithmic_solution(data: List[int]) -> int:
    """
    Algorithm Design:
    1. Preprocessing: Input validation and data preparation
    2. Core Algorithm: Optimized computation with memoization
    3. Post-processing: Result validation and verification

    Complexity: O(n) time, O(1) space for this implementation
    """
    # Step 1: Preprocessing
    if not data:
        return 0

    # Step 2: Core algorithm with optimization
    result = sum(x for x in data if isinstance(x, int) and x > 0)

    # Step 3: Post-processing validation
    return max(0, result)
```

B.3 Proof Domain Strategies

The Proof Domain uses five pathways for robust mathematical property discovery and theorem generation. Each pathway approaches mathematical property identification from different perspectives to ensure comprehensive coverage.

Unified Proof Generation Template

[SWAP:PATHWAY] options:

- **Natural Language**: Extract mathematical properties directly from problem descriptions
- **Unit Tests**: Analyze test patterns to generalize mathematical theorems
- **Code Analysis**: Reverse-engineer properties from implementation structure
- **Type-Guided**: Leverage enhanced types for precise theorem generation
- **Termination**: Focus on termination proofs and complexity bounds

```
# Task: Generate Lean implementation AND theorems via [SWAP:PATHWAY]

**PATHWAY:** [SWAP:PATHWAY_SPECIFIC_ANALYSIS]

Your task is to create:
1. **Lean Implementation**: Complete solution to the problem
2. **Formal Theorems**: Mathematical properties discovered via [SWAP:PATHWAY]

# Output Format
import Std
import Mathlib

def solution (input : InputType) : OutputType :=
  -- Implementation guided by [SWAP:PATHWAY] analysis
  sorry

-- [SWAP:THEOREM_TYPE]
theorem [SWAP:THEOREM_NAME] (input : InputType) :
  [SWAP:THEOREM_STATEMENT] := by sorry
```

B.3.1 Concrete Pathway Examples

Prompt — Natural Language → Lean Theorems

Analysis Focus: Extract the key mathematical properties and correctness conditions from the natural language problem statement and express them as Lean theorems about `{{ function_name }}`. The model must implement `{{ function_name }}` and generate *Theorem 1* (functional correctness) and *Theorem 2* (input/output relationship).

```
-- Theorem 1: functional correctness
theorem {{ function_name }}_correctness
    ({{ function_params }}) (h : precondition {{ function_params }}) :
    property ({{ function_name }} {{ param_names }}) := by
  sorry

-- Theorem 2: input/output relationship
theorem {{ function_name }}_input_output_relationship
    ({{ function_params }}) :
    relation {{ param_names }} ({{ function_name }} {{ param_names }}) := by
  sorry
```

Prompt — Unit Test Pattern → Lean Theorems

Analysis Focus: Study unit test patterns to infer the mathematical relationships and invariants they enforce, and express these as Lean theorems about `{{ function_name }}`. The model must implement `{{ function_name }}` and generate *Theorem 1* (a property generalized from the tests) and *Theorem 2* (an invariant relating multiple calls to `{{ function_name }}`).
The code and theorems should capture: 1. **Functional implementation**: a working solution that passes the tests 2. **Pattern recognition**: the mathematical patterns the tests verify 3. **Invariant properties**: relationships that always hold between inputs/outputs 4. **Boundary behavior**: how the function behaves on edge cases 5. **Mathematical correctness**: properties that ensure the solution is correct

```
-- Theorem 1: property generalized from test patterns
theorem {{ function_name }}_test_pattern_theorem
    ({{ function_params }}) :
    property_from_tests {{ param_names }} ({{ function_name }} {{ param_names }}) := by
  sorry

-- Theorem 2: invariant identified from tests
theorem {{ function_name }}_test_invariant_theorem
    (input1 input2 : {{ input_type }}) :
    invariant_property ({{ function_name }} input1) ({{ function_name }} input2) := by
  sorry
```

Prompt — Code Implementation → Lean Theorems

Analysis Focus: Reverse-engineer mathematical properties from an existing Lean implementation of `{{ function_name }}`. The model analyzes the code structure and generates theorems that verify algorithmic correctness and implementation-specific invariants.
The theorems should capture: 1. **Functional correctness**: the implementation produces correct results 2. **Algorithmic properties**: what the algorithm guarantees 3. **Mathematical relationships**: core relationships between inputs and outputs 4. **Invariants**: properties that hold throughout execution (loops/recursion) 5. **Boundary behavior**: how edge cases are handled

```
-- Theorem 1: functional correctness of {{ function_name }}
theorem {{ function_name }}_correctness
    ({{ function_params }}) (h : valid_input {{ param_names }}) :
    satisfies_specification ({{ function_name }} {{ param_names }}) {{ param_names }} := by
  sorry

-- Theorem 2: mathematical property preservation
theorem {{ function_name }}_property_preserved
    ({{ function_params }}) :
    property_holds {{ param_names }} ({{ function_name }} {{ param_names }}) := by
  sorry

-- Theorem 3: algorithmic invariant
theorem {{ function_name }}_algorithm_invariant
    ({{ function_params }}) :
    forall_steps_satisfy_invariant ({{ function_name }} {{ param_names }}) := by
  sorry
```

Prompt — Termination → Complexity Proofs

Analysis Focus: Generate Lean theorems specifically about termination and computational complexity for `{{ function_name }}`. The model must implement `{{ function_name }}` with termination in mind and produce termination-focused theorems.
The code and theorems should capture: 1. **Functional implementation**: a working solution with termination guarantees 2. **Recursive/loop termination**: all recursive calls and loops terminate 3. **Well-founded recursion**: decreasing measures or well-founded relations 4. **Complexity bounds**: upper bounds on computation steps

```
-- Theorem 1: termination of {{ function_name }}
theorem {{ function_name }}_terminates
    ({{ function_params }}) :
    ∃ output : {{ return_type }}, {{ function_name }} {{ param_names }} = output := by
  sorry

-- Theorem 2: well-founded recursion (if applicable)
theorem {{ function_name }}_recursion_well_founded
    ({{ function_params }}) :
    WellFounded (recursion_relation {{ param_names }}) := by
  sorry

-- Theorem 3: complexity bound for {{ function_name }}
theorem {{ function_name }}_complexity_bound
    ({{ function_params }}) :
    computation_steps ({{ function_name }} {{ param_names }}) ≤
    complexity_function (size {{ param_names }}) := by
  sorry
```

Prompt — Type-Guided → Lean Theorems

Analysis Focus: Given a strongly typed Lean implementation of `{{ function_name }}`, generate theorems that explicitly leverage its refined types and mathematical structure. The goal is to use the enhanced types to prove stronger correctness and invariant properties.
The theorems should capture: 1. **Type correctness**: the implementation respects all type-level constraints 2. **Mathematical properties**: properties enabled by the underlying structures 3. **Enhanced invariants**: stronger guarantees derived from refined types 4. **Structural verification**: properties of algebraic / data structures 5. **Precision benefits**: results that would be impossible with weaker types

```
-- Theorem 1: type-level correctness for {{ function_name }}
theorem {{ function_name }}_type_level_correctness
    ({{ function_params }}) :
    TypeConstraintSatisfied ({{ function_name }} {{ param_names }}) := by
  sorry

-- Theorem 2: preservation of mathematical structure
theorem {{ function_name }}_structure_preservation
    ({{ function_params }}) :
    PreservesStructure ({{ function_name }} {{ param_names }}) {{ param_names }} := by
  sorry

-- Theorem 3: enhanced invariant using refined types
theorem {{ function_name }}_enhanced_invariant
    ({{ function_params }}) :
    StrongerInvariant {{ param_names }} ({{ function_name }} {{ param_names }}) := by
  sorry
```

B.4 Iterative Improvement Strategies

The framework implements iterative improvement mechanisms using chain-of-thought error analysis. This enables systematic refinement of failed solutions through structured error diagnosis and targeted corrections.

Prompt — Chain-of-Thought Error Analysis

Goal: Systematically improve failed solutions through structured error analysis and reasoning refinement. The approach follows a four-phase methodology to identify, analyze, and correct solution failures.
Process: Four Steps

- **Step 1: Error Classification** Identify syntax errors, semantic errors, logic errors, type mismatches, edge case failures.
- **Step 2: Root Cause Analysis** Determine why the errors occurred and what concepts were misunderstood or mismodeled.
- **Step 3: Solution Strategy Design** Plan algorithmic changes, type corrections, edge case handling, and specification fixes.
- **Step 4: Systematic Implementation** Apply improvements based on the analysis and verify correctness.

```
# RETRY ATTEMPT {{ retry_attempt }}/{{ max_retries }}

Your previous solution failed. Please analyze the errors and generate an improved solution.

## Previous Python/Lean Code (FAILED)
{{ previous_python/Lean_code }}

## Error Messages and Feedback
{{ python/Lean_errors }}
```

Meta-Analysis Process

Analysis Methodology: Examine theorems from all five pathways to identify robust mathematical properties. The process synthesizes pathway-specific insights into integrated theorem sets with enhanced verification coverage.

```
{
  "intersection_analysis": {
    "common_concepts": ["monotonicity", "bounds", "optimality", "correctness"],
    "shared_properties": ["invariant_preservation", "edge_case_handling"],
    "robust_theorems": ["most_frequently_appearing", "highest_mathematical_rigor"],
    "pathway_specific_insights": ["unique_contributions_per_pathway"]
  },
  "final_theorem_selection": [
    "solution_correctness: Essential functional correctness property",
    "solution_bounds: Mathematical bounds and constraint verification",
    "solution_termination: Computational termination guarantees"
  ],
  "complete_Lean_file": "-- Complete integrated Lean file
import Std
import Mathlib

def solution (input : InputType) : OutputType :=
  complete_implementation


theorem solution_bounds (input : InputType) :
  0 ≤ solution input ∧ solution input ≤ upper_bound input := by sorry
[..omitted for brevity]
}
```

C BRIDGE Framework: Extended Analysis

C.1 Programming Paradigms and Verification

In LEAN4, programs are mathematical objects within dependent type theory. The structure of code directly shapes verification success. Paradigms emphasizing *immutability*, *structural recursion*, and *compositionality* align with Lean's requirements, while those built on *mutation* and *control effects* introduce complex proof obligations.

Key Paradigm Characteristics:

- **Functional**: Pure functions, pattern matching, structural recursion. *Pros*: Compositional semantics, fewer aliasing hazards, invariants encoded in types. *Cons*: Unfamiliarity for models trained on imperative code.
- **Imperative**: Mutable variables, loops, state transitions. *Pros*: Hardware-close, familiar patterns. *Cons*: Requires loop invariants, complex aliasing reasoning.
- **Object-Oriented**: Encapsulation, subtyping, dispatch. *Pros*: Modularity in large systems. *Cons*: Frame conditions, ownership models complicate specifications.

BRIDGE Approach: We use *structured intermediate representations* to guide models toward Lean-friendly structures. For algorithmic tasks, OCaml/Lean-style recursive solutions yield code with appropriate measures and invariants, while Python/Java-style loops require models to recover invariants post-hoc.

C.2 Specification Paradigms

Specifications define *what* programs must guarantee, forming the bridge between informal requirements and provable implementations. BRIDGE supports multiple specification paradigms to prevent vacuity and improve coverage.

Design by Contract (DbC):

- **Core concept**: Every component has explicit preconditions, postconditions, and invariants
- **In Lean**: Preconditions $\rightarrow$ hypotheses; postconditions $\rightarrow$ dependent types; invariants $\rightarrow$ reusable properties
- **Limitations**: Cannot express temporal/relational properties; provides no proof construction guidance

Property-Based Testing (PBT):

- **Core concept**: Universal behavioral properties over broad input spaces (monotonicity, commutativity, bounds)
- **BRIDGE usage**: Models articulate general properties conceptually before implementation, not executable tests
- **Validation**: We use PLAUSIBLE library to test generated properties on randomized inputs, filtering vacuous specifications

Defensive Programming:

- **Core concept**: Validate all inputs, check invariants, handle errors explicitly
- **BRIDGE usage**: Pre-specification filter to surface implicit assumptions that cause vacuous contracts
- **Trade-off**: Improves robustness but can obscure core logic with excessive checks

Test-Driven Development (TDD):

- **Core concept**: Tests as executable specifications capturing intended behavior and edge cases
- **BRIDGE usage**: Specification discovery mechanism—models reason about comprehensive test cases before formal specification
- **Effectiveness**: Particularly useful for combinatorial problems requiring exhaustive case analysis

C.3 Multi-Pathway Proof Discovery

Proof generation is the most challenging verification stage. Unlike code or specifications, theorems must capture deep mathematical relationships that make programs correct.

BRIDGE Approach: Five Complementary Pathways

1. **Natural Language → Code+Proofs**: Direct extraction from problem descriptions
2. **Unit Tests → Code+Proofs**: Pattern analysis from test cases to hypothesize correctness properties
3. **Code → Proofs**: Two-stage approach—first generate correct Lean code, then derive proofs from structure
4. **Typed Code → Proofs**: Leverage rich dependent types to guide proof statement discovery
5. **Termination → Proofs**: Focus on termination analysis and well-founded measures before full correctness

Intersection Analysis: We cluster candidate proofs across pathways and identify properties appearing consistently. This multi-perspective approach surfaces robust mathematical insights rather than pathway-specific artifacts.

Universal Proof Categories Discovered:

- **Bounds proofs**: Lower/upper bounds on outputs (78% of problems)
- **Monotonicity proofs**: How outputs vary with parameters (65% of problems)
- **Invariant preservation**: Properties maintained during execution (52% of problems)
- **Optimality proofs**: Solutions are maximal/minimal (43% of problems)
- **Termination properties**: Recursive algorithms terminate (89% of recursive problems)

C.4 Domain-Specific Proof Patterns

Different algorithmic categories require distinct proof strategies:

Sorting/Searching (e.g., Maximum Beauty): Correctness depends on range constraints and sliding window invariants. Verification requires proving: (i) algorithm returns achievable subsequence, (ii) result is optimal, (iii) sliding window invariant preserved. Benefits from functional reasoning due to natural alignment with structural recursion.

Combinatorics (e.g., Incremovable Subarrays): Stress counting correctness and exhaustive enumeration. Typical theorems ensure exact counting of valid cases, trivial upper bounds, and closed forms for special inputs. Correctness depends on exhaustive case reasoning rather than structural induction.

Tree Algorithms (e.g., Tree DP): Require structural induction and recursive invariant maintenance. Theorems establish correctness by induction on tree structure. Natural alignment with functional paradigms—both algorithm and proof are structurally recursive.

D Proof Generation Examples and Analysis

We present selected examples illustrating the proof generation challenges our BRIDGE framework addresses. Each highlights how different algorithmic categories require distinct reasoning strategies, motivating our three-domain approach to formal verification.

D.1 Sorting and Searching: Maximum Beauty

The `maximumBeauty` problem exemplifies the sorting/searching category, where correctness depends on range constraints and sliding window invariants. The task is to find the longest subsequence whose elements can all be transformed to the same value within a tolerance k.

```
def maximumBeauty (nums : List Int) (k : Int) : Int :=
  let arr := nums.toArray.qsort (· ≤ ·)
  -- crude but safe upper bound: each step increments l or r,
  -- and both are ≤ arr.size, so ≤ 2 * arr.size steps.
  let fuel0 := 2 * arr.size.succ

  -- structurally recursive on ‘fuel‘
  let rec go : Nat → Nat → Nat → Nat → Int
  | 0,      l, r, best => best
```

```
  | fuel+1, l, r, best =>
      if h : r = arr.size then
        -- window has reached the end
        best
      else
        if arr[r]! - arr[l]! ≤ 2 * k then
          -- expand window to the right
          go fuel l (r + 1) (max best (r - l + 1))
        else
          -- shrink window from the left
          go fuel (l + 1) r best

  go fuel0 0 0 0
```

Listing 2: Maximum Beauty Implementation (truncated)

Verification relies on proving (i) that the algorithm returns an achievable subsequence, (ii) that it is optimal, and (iii) that the sliding window invariant is preserved:

```
/-
  Specifications for subsequences and the "can be transformed" property.
  We model a subsequence by a list of indices `Fin nums.length`,
  in strictly increasing order.
-/

/-- `idxs` is a (strictly increasing) subsequence of the positions of `nums`. -/
def validSubseq (nums : List Int) (idxs : List (Fin nums.length)) : Prop :=
  idxs.Pairwise (· < ·)

/--
  There exists some center value `c` such that every chosen element
  is within `k` (in absolute value) of `c`.
-/
def canTransform (nums : List Int) (k : Int) (idxs : List (Fin nums.length)) : Prop :=
  ∃ c : Int, ∀ i ∈ idxs, |nums.get i - c| ≤ k

/--
  Correctness: `maximumBeauty` returns the size of *some* valid subsequence
  that can be transformed within radius `k`.
-/
theorem maximumBeauty_correct (nums : List Int) (k : Int) :
  ∃ idxs : List (Fin nums.length),
    validSubseq nums idxs ∧
    canTransform nums k idxs ∧
    idxs.length = maximumBeauty nums k := by
  sorry


/--
  Optimality: no valid transformable subsequence is longer than `maximumBeauty`.
-/
theorem maximumBeauty_optimal (nums : List Int) (k : Int) :
  ∀ idxs : List (Fin nums.length),
    validSubseq nums idxs →
    canTransform nums k idxs →
    idxs.length ≤ maximumBeauty nums k := by
  sorry
```

Listing 3: Correctness Theorems

This domain benefits from functional reasoning because the recursive sliding window structure aligns naturally with structural recursion and proof mechanisms.

D.2 Combinatorics: Incremovable Subarray Count

Combinatorial problems stress counting correctness and exhaustive enumeration. The `incremovableSubarrayCount` problem counts subarrays whose removal leaves a strictly increasing sequence.

```
/-- Check if an array of `Int`s is strictly increasing. -/
def isStrictlyIncreasing (nums : Array Int) : Bool :=
  Id.run do
    let n := nums.size
    if n ≤ 1 then
      return true
    let mut ok := true
    let mut prev := nums[0]!
    for i in [1:n] do
      let x := nums[i]!
      if prev < x then
        prev := x
      else
        ok := false
    return ok

/--
`isValidRemoval nums i j` is true iff removing the subarray `nums[i .. j)`
(half-open interval) leaves a strictly increasing array.
We assume `0 ≤ i ≤ j ≤ nums.size`; otherwise we return `false`.
-/
def isValidRemoval (nums : Array Int) (i j : Nat) : Bool :=
  let n := nums.size
  if h : i ≤ j ∧ j ≤ n then
    let left  := nums.extract 0 i   -- [0, i)
    let right := nums.extract j n   -- [j, n)
    let remaining := left ++ right
    isStrictlyIncreasing remaining
  else
    false

/--
Count the number of subarrays whose removal leaves a strictly increasing sequence.
-/
def incremovableSubarrayCount (nums : Array Int) : Nat :=
  Id.run do
    let n := nums.size
    let mut count := 0
    for i in [0:n] do
      for j in [i:n] do
        if isValidRemoval nums i j then
          count := count + 1
    return count
```

Listing 4: Incremovable Subarray Count (core loop)

Typical theorems ensure (i) exact counting of valid removals, (ii) a trivial size-squared upper bound, and (iii) a closed form for strictly increasing inputs:

```
/--
The finite set of all valid removals, as pairs `(i,j)`.
We enumerate all `0 ≤ i, j < n` pairs, and filter by `validRemovalPred`.
-/
def validRemovals (nums : Array Int) : Finset (Nat × Nat) :=
  let n := nums.size
  let allPairs : Finset (Nat × Nat) :=
    (Finset.range n).product (Finset.range n)
  allPairs.filter (validRemovalPred nums)
/--
```

```
Correctness: ‘incremovableSubarrayCount‘ equals the number of valid removals.
-/
theorem incremovable_correct (nums : Array Int) :
  incremovableSubarrayCount nums =
    (validRemovals nums).card := by
  sorry

/--
Trivial upper bound: at most ‘size^2‘ many removals.
-/
theorem incremovable_upper (nums : Array Int) :
  incremovableSubarrayCount nums ≤ nums.size ^ 2 := by
  sorry
```

Listing 5: Combinatorial Theorems

Here, correctness depends on exhaustive case reasoning, contrasting with the structural reasoning used for sorting.

D.3 Tree Algorithms: Structural Induction

Tree dynamic programming problems require structural induction and recursive invariant maintenance. The `maximumScoreAfterOperations` problem computes the optimal score while preserving tree health.

```
partial def dfs (adj : Array (Array Nat)) (val : Array Int)
    (node parent : Nat) : Int × Int :=
  let children := (adj[node]!).filter (fun c => c ≠ parent)
  if children.isEmpty then
    (0, val[node]!)
  else
    let res    := children.map (fun c => dfs adj val c node)
    let scores := (res.map (·.1)).toList.sum
    let costs  := (res.map (·.2)).toList.sum
    (max (scores + val[node]!) (scores - costs + val[node]!),
     val[node]!)
```

Listing 6: Tree DP Core (simplified)

In contrast to the array and combinatorial cases, producing a fully verified version of this tree DP (with an explicit `termination_by` proof) remained challenging for the model, even under functional prompting. The model consistently discovered the correct recursive structure over children and the score/cost decomposition, but it struggled to synthesize a well-founded measure acceptable to Lean's termination checker. In particular, it attempted several plausible termination arguments (e.g., measures based on the number of unvisited nodes or the height of the remaining subtree), yet could not turn these into a fully checked LEAN4 termination proof. Nevertheless, the functional reasoning stage still led to a substantially better design than direct Lean generation, which often produced non-structural recursion or cyclic calls that could not even type-check.

Theorems establish correctness by induction on the tree structure:

```
theorem dfs_correct
    (adj : Array (Array Nat)) (val : Array Int) (root : Nat) :
  let p := dfs adj val root (-1)
  in p.1 ≥ 0 ∧ p.2 ≥ 0 ∧
     p.1 = optimalScore adj val root := by
  sorry
```

Listing 7: Tree DP Theorems

Theorem 7 illustrates the intended correctness and optimality statement for this DP: the score component of `dfs` should coincide with an abstract specification `optimalScore`, and both score and cost should be nonnegative. In our experiments, models frequently proposed high-level termination and correctness arguments matching this specification (e.g., decreasing measures over visited nodes or subtree sizes), but were unable to fully discharge the corresponding LEAN4 proof obligations; we therefore present this theorem as an aspirational specification rather than a fully verified artifact.