BRIDGE: Building Representations in Domain-Guided Program Synthesis

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

Abstract

Large language models (LLMs) are good at generating code, but remain brittle for formal verification in systems like LEAN4. A core scalability challenge is that verified synthesis requires consistent outputs across multiple artifacts: executable code, precise specifications, theorem statements, and ultimately proofs. Existing approaches rarely treat these as a unified pipeline. We present **BRIDGE**, a structured prompting framework that decomposes verification into three interconnected domains: Code (implementations), Specifications (formal intent), and Theorem Statements (constructive correctness claims), and elicits domain-specific intermediate reasoning to connect them. In LEAN4, BRIDGE often adopts a code-first workflow, using the generated implementation as a semantic anchor for downstream specification and theorem statement generation. Across 178 algorithmic problems and five LLMs, BRIDGE improves Lean executable correctness by nearly $1.5\times$ over direct baselines and can be $2\times$ more sample-efficient at inference time, requiring fewer samples per verified solution at comparable generation lengths. We further find that specification-driven prompting improves Python pass rates by up to 17.5 percentage points. Beyond inference-time prompting, supervised fine-tuning on BRIDGE-style reasoning traces yields nearly $1.5\times$ higher Lean pass success than code-only SFT, indicating that these intermediate representations are learnable. BRIDGE provides a practical foundation for scaling verified synthesis while leaving full semantic proof completion as an open problem. The code and datasets are available here: github.com/lean-dojo/BRIDGE.

[1]California Institute of Technology [2]Amazon. Correspondence to: Robert Joseph George <rgeorge@caltech.edu>.

The 3^{rd} *AI for Math Workshop* at the 43^{rd} International Conference on Machine Learning (ICML), Seoul, South Korea, 2026. Copyright 2026 by the author(s).

1. Introduction

Program verification aims for guarantees stronger than passing tests: an implementation should satisfy a precise specification for *all* valid inputs and executions (Hoare, 1969; Floyd, 1967). In modern proof assistants, this goal requires a collection of coupled artifacts rather than a single generated object: executable code, a formal specification, theorem statements connecting the code to the specification, and proof scripts checked by the kernel (de Moura et al., 2015). Treating these artifacts independently makes semantic drift easy. A model can emit code that passes tests, a specification that typechecks but is vacuous, a theorem statement that looks correct but fails to capture the intended property, or a proof of a weak claim that says little about the original program. Current LLM code generation systems struggle in this setting because Lean requires more than an algorithm sketch (Chen et al., 2021; Li et al., 2022). Recursive calls must be justified, types must agree with the intended invariants, and specifications must be strong enough to make the theorem meaningful without becoming impossible to prove (de Moura et al., 2015; mathlib Community, 2020). Verification therefore exposes a representation problem: the model must translate informal intent into a sequence of artifacts whose syntax, types, recursion structure, contracts, and proof obligations agree with one another.

Interactive theorem provers such as LEAN4 make this coupling especially sharp because Lean unifies programming, specification, and proof inside a single dependent type theory. Generated programs must elaborate, satisfy typing constraints, and respect Lean's termination and totality discipline (de Moura et al., 2015; mathlib Community, 2020). We therefore use *Lean executable correctness* as a scalable code metric: a solution must elaborate, satisfy termination and totality checks, and pass the benchmark tests. Full formal verification remains the long term goal. We propose **BRIDGE**, a structured prompting framework for generating and checking linked verification artifacts. BRIDGE fixes a three domain decomposition, **Code**, **Specifications**, and **Theorem/Proof Artifacts**, but allows flexible ordering and conditioning between them. In the Lean experiments, code/scaffold-first routes are often effective: the model gen-

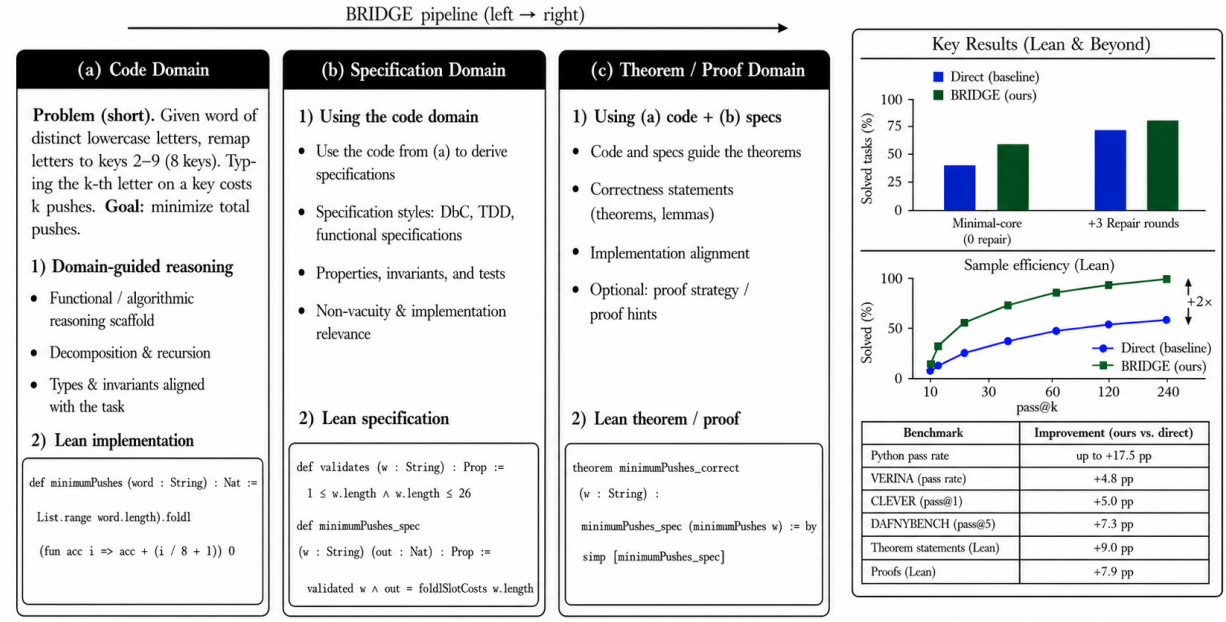


Figure 1. BRIDGE methodology and key results. The figure shows one left-to-right BRIDGE instantiation over Code, Specification, and Theorem/Proof artifacts; in general, arrows denote conditioning/checking dependencies, not a mandatory order. Lean feedback can localize repair to the failed artifact. Appendix G gives the detailed prompts.

erates Lean-compatible code using an aligned scaffold, then uses that implementation as one semantic anchor for specifications, theorem statements, and targeted proof attempts.

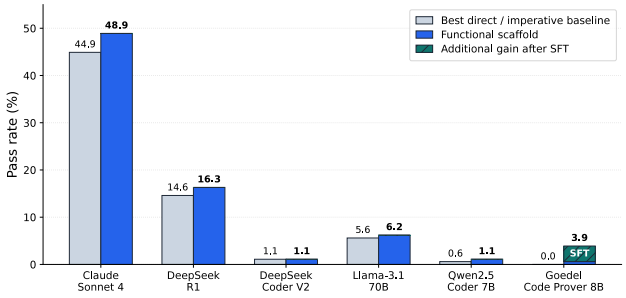
BRIDGE is not a verified compiler from OCAML, HASKELL, PYTHON, or C++ into Lean. The intermediate program is a reasoning scaffold, not a certified source artifact. Semantic confidence instead comes from tests, elaboration, termination checks, nonvacuity checks, implementation relevance, proof attempts, and human inspection. BRIDGE studies the front end of verified synthesis, not arbitrary proof completion at scale. Its purpose is to make the code, specification, theorem, and proof attempt pipeline easier to inspect and easier for downstream proof search to use.

Contributions. We introduce BRIDGE, which decomposes verifiable coding into linked Code, Specification, and Theorem/Proof domains to reduce semantic drift across formal artifacts. On a 178 problem Lean benchmark, BRIDGE improves executable correctness by up to $1.5\times$ over direct prompting and reaches comparable success with up to roughly $2\times$ fewer Lean evaluations. Specification guided prompting improves code generation by up to 17.5 percentage points, while BRIDGE also improves public Lean and solver based verification benchmarks, including VERINA, CLEVER, and DAFNYBENCH. Finally, supervised fine tuning on BRIDGE traces outperforms matched direct trace fine tuning, with Goedel-Code-Prover-8B improving on VERINA from 20.3% to 29.4%.

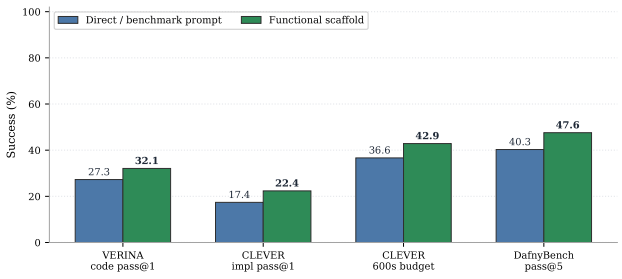
2. Methodology

BRIDGE treats verifiable coding as an artifact alignment problem rather than a single code generation task. A standard code prompt asks a model to jump directly from an informal problem statement to a final implementation. In a proof assistant this is often the wrong abstraction: even a good algorithm sketch must be expressed as a total Lean definition, the intended behavior must be stated as a formal predicate, the theorem statement must connect the two, and the proof must be accepted by the kernel. BRIDGE therefore represents the task as a graph of coupled artifacts generated with domain-specific reasoning, conditioned on one another, and checked for consistency.

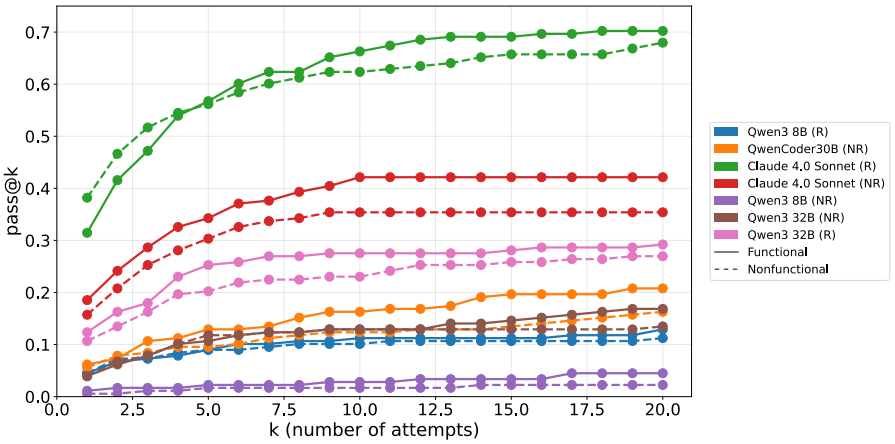
The framework has three domains. The *Code* domain produces the executable implementation. The *Specification* domain states assumptions, edge cases, and intended behavior as formal constraints. The *Theorem/Proof* domain turns those constraints into correctness claims and bounded proof attempts. BRIDGE does not require a fixed order among the domains: one can generate code first, use specification reasoning to constrain code, or ask theorem/proof prompts to expose obligations before repair. In our main Lean experiments, a scaffold/code-first route is especially effective: the model constructs a functional scaffold and Lean implementation, and downstream prompts use it as a semantic anchor.



(a) BRIDGE suite.



(b) Public transfer.



(c) Pass@k efficiency.

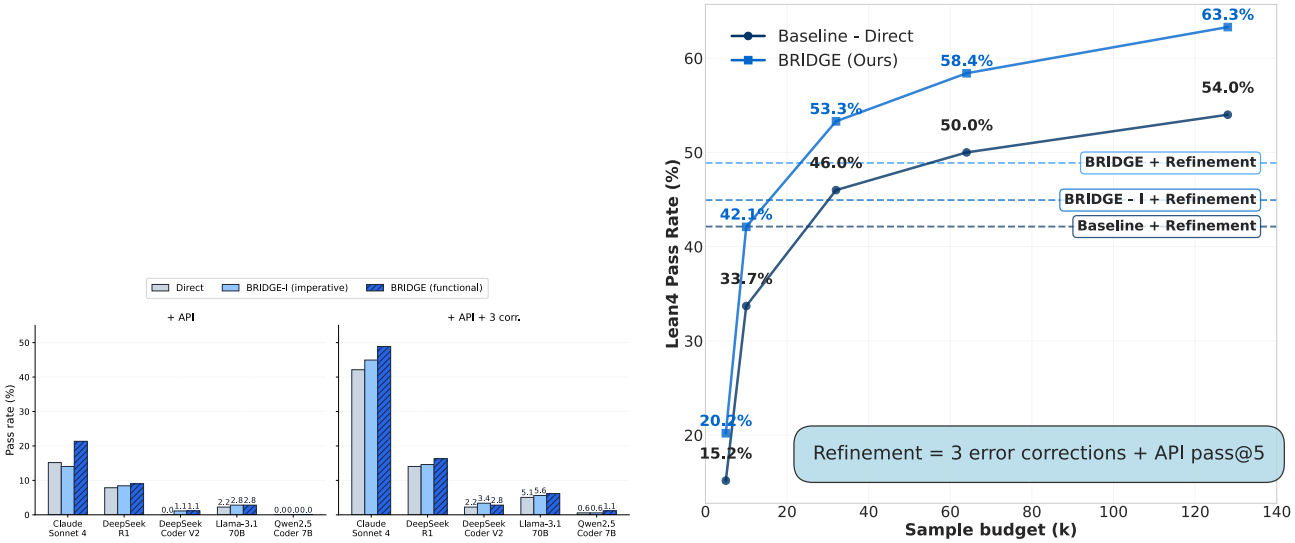
Figure 2. Main quantitative results. Functional scaffolding improves executable Lean synthesis, transfers to public verification settings, and reaches comparable success with fewer Lean evaluations. Functional denotes the recursive/immutable scaffold; direct/imperative baselines omit it. R/NR denote reasoning/no-reasoning traces, and "best direct/imperative" is the best non-functional baseline at matched budget. Details are in Appendix A.1.

Code-domain scaffolds. The Code domain compares direct Lean generation with intermediate reasoning styles. Functional scaffolds, inspired by OCaml and Haskell, emphasize immutable data, pattern matching, recursion, and local helpers; we denote this condition as BRIDGE-F. Imperative scaffolds, inspired by Python and C++, emphasize loops, mutable state, arrays, and updates; we denote this non-functional variant as BRIDGE-I. These scaffolds are not executed, trusted, or treated as source programs. They are reasoning artifacts that guide the model toward a Lean-compatible implementation. All reported code results are evaluated only on the final Lean output.

Specification-domain scaffolds. The Specification domain makes task intent explicit. We ask for design-by-contract preconditions and postconditions, property-oriented specifications, test-derived edge cases, defensive assumptions, and algorithmic invariants. These styles expose different failure modes. A contract may miss an optimality property; a property theorem may be true but too weak; a defensive specification may assume away the hard cases. BRIDGE therefore evaluates specifications using multiple signals rather than elaboration alone: nonvacuity, acceptance of intended implementations, rejection of wrong implementations, and usefulness for theorem/proof generation.

Theorem/proof scaffolds. The Theorem/Proof domain asks whether generated code and specifications can support formal claims. We distinguish three levels of success: an elaborating theorem statement, a proof script accepted without `sorry`, and a faithful semantic proof of the original natural-language task. The present work focuses on the first two as scalable diagnostics. A completed implementation-aligned proof can still be weaker than the task-level mathematical specification, but it reveals whether the artifact chain is structured enough for downstream proof search.

Artifact consistency loop. BRIDGE is built around cross-artifact consistency. Code generation is followed by Lean checking; specification generation conditions on both the natural-language task and candidate implementation; theorem generation asks for claims that connect the implementation to the specification; proof attempts are inspected for elaboration, nonvacuity, and use of the implementation. This gives a practical feedback loop: a failure in one arti-



(a) Ablations with API context and repair.

(b) Lean evaluation efficiency.

Figure 3. Additional main results. Functional scaffolds remain strong under ablations and reduce Lean evaluations needed for a passing implementation. BRIDGE-F denotes the functional variant; BRIDGE-I denotes the imperative/non-functional variant.

Table 1. BRIDGE domains and the checks used in this paper. The checks are deliberately layered: executable success, specification usefulness, theorem elaboration, and proof completion provide complementary views of artifact alignment.

Domain	Generated artifact	Main checks
Code	Lean implementation	Elaboration, totality, termination, tests
Specification	Formal predicates and contracts	Nonvacuity, acceptance, wrong-code rejection
Theorem/Proof	Theorem statements and scripts	Statement TC, proof TC, no-`sorry` completion

fact can be localized and repaired without discarding the entire artifact graph. For example, a Lean implementation may compile but fail a boundary test; a specification may elaborate but reject the intended implementation; a theorem statement may typecheck but never mention the generated function; or a proof may complete only because the statement is vacuous. BRIDGE surfaces these failures as separate diagnostics and repair targets.

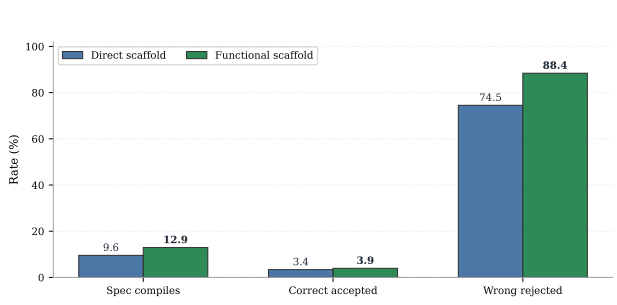
This loop is intentionally modest. It does not certify that the generated program satisfies the full informal task unless the final theorem and proof express that task faithfully. Instead, it gives a checked path from an informal problem to a set of formal objects whose failures can be localized. The main empirical question is whether the intermediate representation changes the distribution of generated Lean programs and downstream proof artifacts in a useful way.

3. Experimental Setup

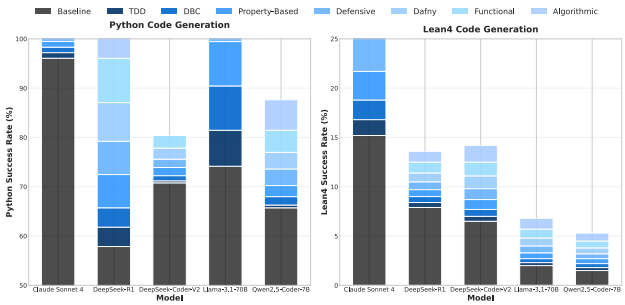
We evaluate BRIDGE in three settings: executable Lean synthesis, specification-guided code generation, and theorem/proof artifact generation. The primary benchmark is a 178-problem Lean suite derived from algorithmic programming tasks. Each problem contains a natural-language task statement, a Lean function signature or harness, and 100 oracle tests. A candidate solution is counted as executable-correct only if it elaborates, satisfies Lean's termination and totality checks, and passes the tests. This criterion is stricter than ordinary unit-test accuracy in an untyped language because Lean rejects partial definitions, ill-typed recursion, and many informal algorithm sketches before tests can even

Algorithm 1 BRIDGE-style verified synthesis loop, shown for one scaffold/code-first instantiation

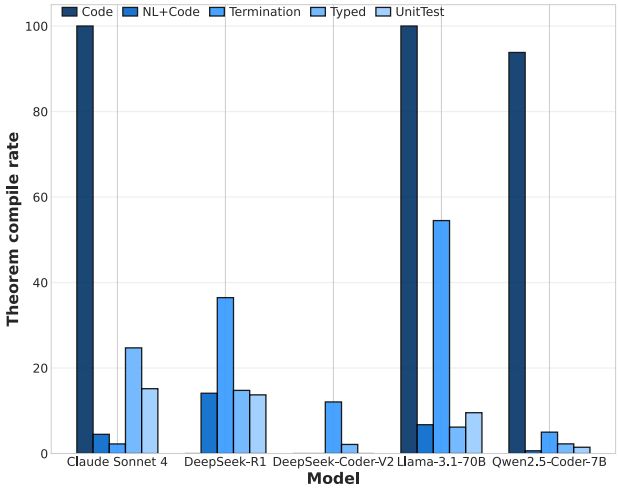
Input: natural-language task x, Lean harness H, sample budget k
for $i = 1, \ldots, k$ **do**
  Generate a domain scaffold r_i using a chosen reasoning style
  Generate Lean implementation c_i conditioned on (x, r_i, H)
  Check c_i for elaboration, totality, termination, and tests
  if c_i passes executable checks **then**
    Generate specification candidates s_i conditioned on (x, c_i)
    Generate theorem/proof candidates t_i, p_i conditioned on (x, c_i, s_i)
    Score nonvacuity, implementation relevance, theorem TC, proof TC
  end if
end for
Return the best executable artifacts and their diagnostic proof/spec traces



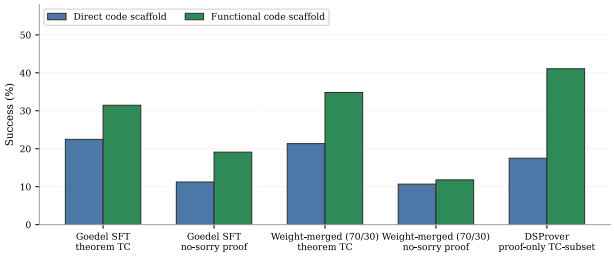
(a) Spec discriminators.



(b) Spec reasoning.



(c) Theorem statements.



(d) Proof diagnostics.

Figure 4. Specification and theorem/proof diagnostics. Specifications can reject wrong code, specification reasoning improves code generation, and functional scaffolds improve theorem/proof artifacts. TDD means test-driven development; DbC means design by contract. Details are in Appendix B and Appendix C.

run. It is still not full semantic verification.

We also test transfer to public verification settings. VERINA contains Lean tasks focused on implementation and specification generation, while CLEVER evaluates Lean code synthesis over implementation tasks. We additionally run a solver-backed DAFNYBENCH pilot to test whether the same reasoning bias helps outside interactive theorem proving. These transfer settings are useful because they differ in prompt format, harness style, and checker backend; a scaffold that only overfits our internal harness should not consistently improve across them.

The main Lean study uses five model families: CLAUDE SONNET 4, DEEPSEEK CODER V2, DEEPSEEK R1, LLAMA-3.1-70B, and QWEN2.5 CODER. For each strategy, we draw independent samples and report pass@k, where a task is solved if at least one sample passes the relevant checker. We compare functional scaffolds against direct Lean prompting, imperative scaffolds, and a length-controlled Double Lean baseline. Double Lean gives the model an extra Lean-oriented reasoning step with comparable generation length, so it tests whether the gain is simply due to producing more tokens. We additionally evaluate repair rounds, in which failed candidates are regenerated or patched using checker feedback.

Table 2. Evaluation settings used in this paper. The main metric is executable correctness for Lean code; specification and proof metrics evaluate downstream verification artifacts.

Setting	Size	Primary measurement
BRIDGE Lean suite	178	Elaborates, total/terminating, passes tests
VERINA	187	Public Lean implementation/specification tasks
CLEVER	161	Public Lean implementation tasks
DAFNYBENCH	782	Solver-backed verified-code pilot
Specification discriminator	178	Correct-code acceptance and wrong-code rejection
Proof pilot	178	Theorem TC and proof completion without `sorry`

Metrics. For code, the main metric is pass@k under the Lean harness. For specifications, we report elaboration, non-vacuity, correct implementation acceptance, wrong imple-

Table 3. Core Lean executable results on the 178-problem BRIDGE suite. Functional scaffolding improves over direct prompting and over a length-controlled Double Lean baseline.

Prompting strategy	Core pass	With 3 repairs
Direct Lean	44/178	77/178
Double Lean	49/178	81/178
Functional scaffold	59/178	83/178

mentation rejection, and downstream usefulness. For theorem/proof artifacts, we report statement typechecking, strict nonvacuity, proof typechecking, proof completion without `sorry`, and average implementation length. These metrics are intentionally separated. A single scalar verification score would hide important distinctions: a proof can complete for a weak statement; a specification can elaborate but fail to constrain behavior; a program can pass tests while not satisfying the intended theorem.

4. Executable Lean Results

Our first result is that the representation used before Lean generation changes executable correctness. On the 178-problem Lean suite, functional scaffolds solve more tasks than direct prompting and more than the length-controlled Double Lean baseline. In the minimal core setting, functional reasoning solves 59/178 tasks, compared with 44/178 for direct prompting and 49/178 for Double Lean. With three repair rounds, it solves 83/178 tasks, compared with 77/178 for direct prompting and 81/178 for Double Lean. The effect is largest before repair, but it remains visible after repair, suggesting that the scaffold improves the initial candidate distribution rather than merely adding feedback rounds. Repair is complementary: checker feedback patches failed artifacts, while the scaffold changes the artifacts proposed initially.

The length-controlled comparison is important. Functional prompts are often longer than direct prompts, but Double Lean has a similar token budget and does not close the gap. The qualitative difference is that functional scaffolds express recursive decomposition, immutable data flow, and helper-function structure before the model commits to Lean syntax. These are the same design choices that Lean's termination checker and type checker prefer. By contrast, an imperative scaffold can describe a correct algorithm while leaving the model to translate mutable loop state into recursive or fold-based Lean code, a transformation that frequently introduces indexing or termination errors.

BRIDGE also improves sample efficiency. In the pass@k regime, functional prompting reaches comparable success with roughly half as many Lean evaluations in several settings. This matters because the expensive part of a verified-synthesis workflow is not only generating text; it is also checking candidates, collecting counterexamples, and running repair loops. A scaffold that produces fewer but more checker-compatible candidates can reduce the number of failed Lean evaluations required to find one accepted implementation.

Table 4. Public transfer counts. Functional scaffolding improves most matched prompt and budget settings, but gains remain model and benchmark dependent.

Benchmark	Model/setting	Direct	Functional	Δ
VERINA	Qwen2.5 Coder pass@1	51/187	60/187	+4.8 pp
VERINA	Base Goedel pass@1	14/187	12/187	-1.1 pp
CLEVER	Qwen pass@1	28/161	36/161	+5.0 pp
CLEVER	Qwen larger budget	59/161	69/161	+6.3 pp
CLEVER	Base Goedel larger budget	66/161	70/161	+2.5 pp
DAFNYBENCH	Solver-backed pass@5	315/782	372/782	+7.3 pp

The public transfer experiments support the same interpretation. Functional scaffolding improves Qwen on VERINA from 51/187 to 60/187 and on CLEVER from 28/161 to 36/161 at pass@1. At larger CLEVER budgets, it improves Qwen from 59/161 to 69/161 and base Goedel from 66/161 to 70/161. In the solver-backed DAFNYBENCH pilot, functional scaffolding improves pass@5 from 315/782 to 372/782. The main exception is base Goedel on VERINA, where functional prompting slightly underperforms direct prompting. We view this as a useful caution: BRIDGE is an inductive bias, not a uniformly dominant decoding rule.

5. Specification Results

The Specification domain asks whether generated formal constraints have semantic bite. A specification that merely elaborates is not enough: it can be vacuous, too weak, unrelated to the task, or shaped so closely to the implementation that it fails to reject wrong code. BRIDGE therefore evaluates specifications with a discriminator-style protocol. For each task, we compare whether the generated specification accepts an intended implementation and rejects plausible wrong implementations. This gives a more informative signal than typechecking alone.

Functional scaffolding improves the rejection of wrong implementations. In our discriminator experiment, functional specifications reject 61/69 wrong variants, or 88.4%, compared with 38/51, or 74.5%, for direct specifications. Correct implementation acceptance remains low for both settings: 7/178 for functional and 6/178 for direct. This negative result is important. It shows that broad, faithful, automatically generated semantic specifications remain difficult even when code generation improves. BRIDGE should therefore be read as a way to organize and diagnose the artifact pipeline, not as a complete replacement for specification engineering.

Specification reasoning also helps ordinary code generation. In Python generation, specification-guided prompting improves pass rates by up to 17.5 percentage points. The

Table 5. Specification discriminator results. Functional scaffolding improves wrong-implementation rejection, while correct-implementation acceptance remains a hard bottleneck.

Specification source	Correct accept	Wrong reject
Direct	6/178	38/51
Functional	7/178	61/69

mechanism is simple but useful: explicit contracts force the model to name boundary cases, output invariants, and hidden assumptions before writing code. Even when the final program is not formally verified, this planning step can prevent common edge-case errors. In Lean, the same planning benefit is present, but the checker imposes additional constraints: the program must elaborate, avoid partiality, and justify recursive structure.

A recurring pattern is that useful specifications are often relational rather than local. For a sorting task, output length and element type are weak properties; permutation and sortedness are stronger. For an optimization task, achievability without optimality is incomplete; optimality without a construction can be hard to prove. For a dynamic program, the natural specification may be over subproblem values rather than over only the final output. BRIDGE's multi-style specification prompts are designed to surface these different views, after which Lean checks and wrong-code rejection expose whether a candidate specification is meaningful.

6. Theorem/Proof and Fine-Tuning Results

The Theorem/Proof domain evaluates whether the generated implementation and specification artifacts support downstream proof search. We run an implementation-aligned proof pilot on all 178 tasks. For each task, we select matched direct and functional Lean candidates, ask a prover to generate a specification and theorem tied to the candidate implementation, and then attempt to complete the proof. This is not a full task-level certification experiment. It is a diagnostic for whether the implementation is structured enough to support checkable claims; strict nonvacuity and task relevance remain necessary before interpreting proofs semantically.

Functional implementations are shorter and yield more theorem/proof successes. With a functional SFT prover, functional code produces 56/178 typechecked theorem statements and 34/178 proofs completed without `sorry`, compared with 40/178 statements and 20/178 completed proofs for direct code. With a direct SFT prover, functional code still improves theorem statement typechecking from 11/178 to 25/178 and no-`sorry` proofs from 5/178 to 12/178. These results suggest that the structure of the implementation itself matters: once the code is shorter, more recursive, and better aligned with Lean idioms, theorem statements are easier to state and proof attempts are easier to discharge.

Table 6. Implementation-aligned theorem/proof pilot on all 178 tasks. Functional code gives provers shorter implementations and more completed proof artifacts under matched budgets.

Prover	Code style	Stmt TC	Proof TC	No `sorry`	Avg chars
Direct SFT	Direct	11/178	5/178	5/178	532.7
Direct SFT	Functional	25/178	12/178	12/178	343.5
Functional SFT	Direct	40/178	20/178	20/178	532.7
Functional SFT	Functional	56/178	34/178	34/178	343.5
Weight merged 70/30	Direct	38/178	19/178	19/178	532.7
Weight merged 70/30	Functional	62/178	21/178	21/178	343.5

We also isolate proof search with DEEPSEEK-PROVER-V2-7B on theorem statements that already elaborate. The direct-code subset contains 40 typechecked statements and the functional-code subset contains 56. The prover completes 7/40 direct proofs and 23/56 functional proofs without `sorry`. This setting removes theorem-statement generation as the only explanation: the prover is given statements that already elaborate, yet functional code still yields a higher proof completion rate. The result is consistent with the BRIDGE hypothesis that code structure affects downstream verification, not only executable success.

Beyond prompting, we test whether BRIDGE-style traces can be learned. We construct matched 2k-row OpenCodeInstruct-style datasets from the same task distribution. One contains direct code traces; the other contains BRIDGE-style functional traces. Starting from GOEDEL-CODE-PROVER-8B, supervised fine-tuning on functional traces produces higher Lean pass success than code-only fine-tuning in our transfer setting, improving VERINA from 20.3% to 29.4%. This suggests that the intermediate representation is not merely an inference-time trick. It provides a learnable inductive bias that can be internalized by a model. **Why functional traces help.** The fine-tuning result aligns with the prompting result. Functional traces teach the model to decompose tasks into structurally recursive pieces, name helper functions, avoid partial operations, and preserve data-flow invariants. These habits are valuable in Lean because executable definitions and proof obligations share the same syntax and semantics. A model that writes Lean code in a proof-friendly style gives both the checker and the downstream prover less accidental complexity to handle.

7. Related Work and Positioning

BRIDGE connects several strands of work in program synthesis, formal verification, and LLM reasoning. Linking code, specifications, and proofs is standard in verification; our contribution is to instantiate artifact alignment as an LLM prompting, training, and evaluation framework for Lean-based synthesis. Classic verification methods separate the implementation from the proof obligation through assertions, invariants, and Hoare-style triples (Hoare, 1969; Floyd, 1967). Modern proof assistants such as Lean and Isabelle make these obligations kernel-checked, but they

also expose the modeling burden: the user must express code, specifications, theorem statements, and proof scripts in a language where all four artifacts share the same formal semantics (de Moura et al., 2015; Nipkow et al., 2002; mathlib Community, 2020). Verified systems such as CompCert and seL4 show the value of this approach, while also illustrating the amount of expert engineering needed for full functional correctness (Leroy, 2009; Klein et al., 2009).

LLM code generation has made rapid progress, but most standard code benchmarks primarily reward executable behavior under tests (Chen et al., 2021; Li et al., 2022). That setting is useful but incomplete for verification: a model can pass examples while failing to expose the invariants needed for a proof. Prompting methods such as chain-of-thought, self-consistency, tree-of-thought, ReAct, Reflexion, and self-refinement show that intermediate reasoning can improve model behavior (Wei et al., 2022; Wang et al., 2022; Yao et al., 2023a;b; Shinn et al., 2023; Madaan et al., 2023). BRIDGE specializes this idea to formal artifacts. The intermediate reasoning is not only explanatory text; it is intended to shape a Lean implementation, a specification, a theorem statement, and a proof attempt that can be checked.

Recent verification-oriented LLM systems often focus on one part of this stack. Solver-backed systems such as Dafny-based pipelines can succeed when the right annotations are generated and an SMT backend can discharge the remaining obligations (Leino, 2010; Loughridge et al., 2024; Wen et al., 2024; Sun et al., 2024). Lean-focused benchmarks and tools emphasize theorem proving, tactic generation, or specification synthesis in an interactive theorem prover (Yang et al., 2023; Wang et al., 2024; Ye et al., 2025; 2026; Thakur et al., 2025). BRIDGE is complementary: it studies how the representation of the implementation affects all downstream artifacts. The same task is evaluated not only by whether code passes tests, but also whether the generated code gives specification and proofs prompts a better semantic anchor.

8. Discussion and Limitations

BRIDGE improves the front end of verified program synthesis, but it does not solve full formal verification. Passing executable Lean tests is not the same as proving semantic correctness. A generated theorem can elaborate while stating a weak property. A proof can complete because the theorem is implementation-shaped or because the specification omits a hard obligation. Even wrong-code rejection is only as strong as the wrong variants used in the discriminator protocol. These limitations are not incidental; they are the reason BRIDGE reports multiple diagnostics rather than a single verification claim.

The main lesson is that artifact alignment is a useful intermediate target. Direct prompting often fails because it treats Lean code, specifications, theorem statements, and proof scripts as separate text-generation problems. BRIDGE instead makes their dependencies explicit. The implementation supplies concrete recursive structure; specifications expose intended behavior and hidden assumptions; theorem statements connect those objects; proof attempts reveal whether the chain is strong enough for a kernel-checked artifact. This decomposition makes failures more interpretable: failed tests indicate implementation errors, wrong-code acceptance indicates weak specifications, theorem nonvacuity checks reveal statement failures, and failed proof obligations identify missing lemmas or mismatched definitions.

Several extensions are natural. First, stronger semantic specification benchmarks are needed. The low correct-implementation acceptance rate shows that specification synthesis remains a bottleneck. Second, theorem generation should be evaluated against task-level gold specifications, not only implementation-aligned contracts. Third, future systems should close the loop with proof search: a failed proof should propose either a code repair, a stronger specification, or an auxiliary lemma, rather than ending the pipeline. Finally, our strongest evidence is on single-function algorithmic Lean tasks and selected public verification benchmarks; BRIDGE should be tested on longer, multi-module and library-heavy programs where artifact drift is more severe.

9. Conclusion

BRIDGE frames verified program synthesis as a representation problem over coupled formal artifacts. Rather than asking a model to jump directly from an informal problem statement to a final proof, BRIDGE decomposes generation into executable code, specifications, theorem statements, and proof attempts, then uses checker feedback and cross-artifact diagnostics to keep these objects aligned. Across our Lean suite and public transfer settings, functional BRIDGE scaffolds improve executable correctness, sample efficiency, specification discrimination, theorem statement generation, proof completion diagnostics, and supervised fine-tuning transfer. The results do not imply end-to-end semantic verification at scale. They show that intermediate functional artifacts can serve as semantic anchors that make formal artifacts easier to generate, check, and repair.

10. Acknowledgements

Robert Joseph George is supported by the DARPA expMath, Harmonic AI Fellowship and AA2.AIMING-1-NSF.2522494 (AIMing: Synergistic Advancement of AI and Mathematics with Dr. Sergei Gukov). Anima Anandkumar is supported in part by the Bren endowed chair, ONR (MURI grant N00014-18-12624), DARPA expMath and the 2050 senior fellow program at Schmidt Sciences. We also thank Modal and Amazon for providing compute credits and hardware resources used to conduct these experiments.

References

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. A benchmark for vericoding: Formally verified program synthesis, 2025. URL https://arxiv.org/abs/2509.22908.

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. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

de Moura, L., Kong, S., Avigad, J., van Doorn, F., and von Raumer, J. The lean theorem prover (system description). In *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction*, volume 9195 of *Lecture Notes in Computer Science*, pp. 378–388. Springer, 2015. doi: 10.1007/978-3-319-21401-6_26. URL https://doi.org/10.1007/978-3-319-21401-6_26.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, 2020. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139/.

Floyd, R. W. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*, volume 19, pp. 19–32. American Mathematical Society, 1967. doi: 10.1090/psapm/019/0235771. URL https://doi.org/10.1090/psapm/019/0235771.

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. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR)*, 2021. URL https://openreview.net/forum?id=jLoC4ez43PZ.

Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al. sel4: Formal verification of an operating-system kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 207–220. ACM, 2009. doi: 10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596.

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

Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, volume 6355 of *Lecture Notes in Computer Science*, pp. 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20. URL https://doi.org/10.1007/978-3-642-17511-4_20.

Leroy, X. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://doi.org/10.1126/science.abq1158.

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

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. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

mathlib Community, T. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International*

Conference on Certified Programs and Proofs, pp. 367–381, New Orleans, LA, USA, 2020. ACM. doi: 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824.

Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. doi: 10.1007/3-540-45949-9. URL https://doi.org/10.1007/3-540-45949-9.

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023. URL https://arxiv.org/abs/2303.11366.

Sun, C., Sheng, Y., Padon, O., and Barrett, C. Clover: Closed-loop verifiable code generation, 2024. URL https://arxiv.org/abs/2310.17807.

Thakur, A., Lee, J., Tsoukalas, G., Sistla, M., Zhao, M., Zetzsche, S., Durrett, G., Yue, Y., and Chaudhuri, S. Clever: A curated benchmark for formally verified code generation, 2025. URL https://arxiv.org/abs/2505.13938.

Wang, R., Zhang, J., Jia, Y., Pan, R., Diao, S., Pi, R., and Zhang, T. Theoremllama: Transforming general-purpose llms into lean4 experts. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 11953–11974, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.667. URL https://aclanthology.org/2024.emnlp-main.667/.

Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models, 2022. URL https://arxiv.org/abs/2203.11171.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models, 2022. URL https://arxiv.org/abs/2201.11903.

Wen, C., Cao, J., Su, J., Xu, Z., Qin, S., He, M., Li, H., Cheung, S.-C., and Tian, C. Enchanting program specification synthesis by large language models using static analysis and program verification, 2024. URL https://arxiv.org/abs/2404.00762.

Xu, X., Li, X., Qu, X., Fu, J., and Yuan, B. Local success does not compose: Benchmarking large language models for compositional formal verification, 2025. URL https://arxiv.org/abs/2509.23061.

Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. Leandojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/4441469427094f8873d0fecb0c4e1cee-Paper-Datasets_and_Benchmarks.pdf.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models, 2023a. URL https://arxiv.org/abs/2305.10601.

Yao, S., Zhao, J., Yu, D., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. In *ICLR*, 2023b. URL https://arxiv.org/abs/2210.03629.

Ye, Z., Yan, Z., He, J., Kasriel, T., Yang, K., and Song, D. Verina: Benchmarking verifiable code generation, 2025. URL https://arxiv.org/abs/2505.23135.

Ye, Z., Yang, A. Z. H., Su, H., Liao, Z., Tenka, S., Qin, Z., Ghai, U., Song, D., and Kong, S. Intent-aligned formal specification synthesis via traceable refinement, 2026. URL https://arxiv.org/abs/2604.10392.

A. Additional Evaluation Details

This appendix reports additional experimental and methodological details across systems and inference settings. The organization follows the BRIDGE artifact stack: Code-domain results, Specification-domain diagnostics, Theorem/Proof-domain diagnostics, training and robustness analyses, related work, prompt templates, and representative examples.

A.1. Code-Domain Ablations and Public Benchmark Counts

This section reports the Code-domain evidence supporting the executable Lean results. We include ablations, public benchmark counts, and generation-length analyses showing how scaffold style, API context, repair, and transfer settings affect final Lean correctness.

A.2. BRIDGE Suite Ablations

Figure 5 gives the ablation breakdown on the 178 problem BRIDGE suite, including API context and repair-round variants. The length-controlled Double Lean control is important because it has comparable generation length to functional scaffolding but does not close the performance gap.

This supports the interpretation that the gain comes from the reasoning representation rather than verbosity alone.

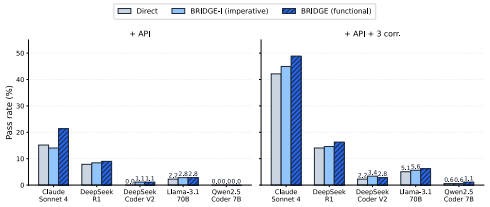


Figure 5. Detailed BRIDGE suite ablations with API context and repair rounds. Functional scaffolding remains the strongest structured prompting condition, while API context and repair rounds narrow but do not remove the representation effect.

A.3. Public Benchmark Transfer Counts

Table 7 gives the transfer counts across public verification benchmarks. Functional scaffolding improves most matched prompt and budget settings, including Qwen on VERINA and CLEVER, larger-budget CLEVER, and solver-backed DAFNYBENCH. The main exception is base Goedel on VERINA, where functional prompting slightly underperforms direct prompting. We interpret this as evidence that the scaffold is a useful inductive bias whose gains still depend on model calibration and benchmark format.

Table 7. Direct versus functional transfer results on public verification benchmarks. Functional scaffolding improves most matched settings, but gains remain model and budget dependent. "Epoch-30" denotes the checkpoint after 30 supervised fine-tuning epochs.

Benchmark	Model and setting	Direct	Functional	Δ
VERINA	Qwen2.5 Coder pass@1	51/187	60/187	+4.8 pp
VERINA	Base Goedel pass@1	14/187	12/187	-1.1 pp
CLEVER	Qwen pass@1	28/161	36/161	+5.0 pp
CLEVER	Qwen larger budget	59/161	69/161	+6.3 pp
CLEVER	Base Goedel larger budget	66/161	70/161	+2.5 pp
CLEVER	Goedel SFT epoch-30 larger budget	30/161	33/161	+1.9 pp
DAFNYBENCH	Solver backed pass@5	315/782	372/782	+7.3 pp

A.4. Generation Length and Sample Efficiency

Functional scaffolds are longer than direct generations, but length alone does not explain the gains. Table 8 reports average word and token counts by strategy. Double Lean is close to the functional methods in length, yet performs worse in the main experiments. This supports the main claim: the useful signal is not simply more text, but a scaffold whose structure better matches Lean.

The practical implication is that functional scaffolds can trade a larger per sample generation for fewer Lean evaluations. Since Lean checking and iterative repair dominate the workflow cost, sample efficiency is a meaningful advantage even when individual generations are longer.

Table 8. Generation lengths by reasoning strategy. Lengths are reported as words and tokens, averaged over all attempts, successful attempts, and failed attempts.

Reasoning Strategy	Average	Successful	Failed
Direct	195 / 270	141 / 196	200 / 278
Double Lean	402 / 555	347 / 480	412 / 570
Python	382 / 526	324 / 446	391 / 540
Haskell	387 / 534	334 / 460	401 / 555
C++	390 / 538	341 / 469	402 / 558
OCaml	408 / 563	343 / 473	421 / 581

B. Specification-Domain Details

This section details how BRIDGE instantiates and evaluates specification reasoning across several formal styles. The goal is not to treat a generated specification as a final certificate of correctness. Instead, specifications act as intermediate formal artifacts that constrain implementations, expose hidden assumptions, and provide targets for theorem statements and proof attempts.

B.1. Programming Paradigms and Specification Structure

In LEAN4, programs are mathematical objects inside dependent type theory, so the structure of the implementation strongly affects the structure of the specification. A functional implementation usually exposes recursive structure, base cases, and compositional definitions directly. This makes it easier to state properties by induction or by rewriting. An imperative implementation may be algorithmically natural, but loops, mutation, and index updates often require additional invariants before the intended behavior can be expressed cleanly.

This distinction matters for specifications. A specification for a recursive function can often mirror the recursive decomposition of the code. A specification for an imperative loop must usually describe the relationship between initial state, loop state, and final state. BRIDGE uses intermediate scaffolds to bias the model toward artifacts that are easier to connect: functional scaffolds expose recursion and invariants early, while specification scaffolds expose assumptions and expected behavior before or after implementation.

Table 9. Specification reasoning styles used in BRIDGE. Each style exposes a different view of task intent and a different failure mode.

Style	Main artifact	Primary risk
Design by Contract	Preconditions and postconditions	Missing relational properties
Property Reasoning	Bounds, monotonicity, preservation	Weak or irrelevant properties
Test Driven Reasoning	Edge case constraints	Overfitting to examples
Defensive Reasoning	Input assumptions and safe behavior	Excessive guards
Algorithmic Reasoning	Decomposition and invariants	Informal constraints only

B.2. Specification Reasoning Styles

Design by Contract. Design by Contract asks the model to identify preconditions, postconditions, and invariants before writing or checking the implementation. In Lean, these usually become hypotheses, predicates over inputs and outputs, or auxiliary invariant definitions. This style is useful for making hidden assumptions explicit. Its main limitation is that local contracts may miss global or relational properties, such as optimality, permutation equivalence, or behavior across multiple calls.

Property reasoning. Property reasoning asks the model to identify universal behavior such as bounds, monotonicity, preservation, symmetry, idempotence, or algebraic laws. These properties often become useful theorem candidates. However, they can also be too weak. For example, a theorem that only states that an output is nonnegative may elaborate and even be provable, while saying very little about the original task.

Test driven reasoning. Test driven reasoning uses examples and edge cases as a way to infer intended behavior. This is useful for tasks with many boundary cases, such as empty inputs, singleton lists, duplicate elements, or degenerate numerical ranges. The limitation is that tests can suggest an incomplete specification. BRIDGE therefore treats test driven reasoning as a source of candidate constraints, not as a replacement for formal semantics.

Defensive reasoning. Defensive reasoning asks the model to surface invalid inputs, hidden assumptions, and safe fallback behavior. This can prevent vacuous specifications that silently assume away difficult cases. The risk is that excessive guards can obscure the core mathematical property or make the specification too narrow.

Algorithmic reasoning. Algorithmic reasoning asks the model to describe the intended computational structure before formalizing the specification. This is useful when the behavior is easiest to characterize through an invariant or recurrence. For example, a dynamic program may require a specification over subproblem values rather than only a final input-output relation.

B.3. Specification Evaluation Signals

A generated specification can elaborate while still being semantically unhelpful. BRIDGE therefore evaluates specifications using multiple signals rather than a single binary label.

The discriminator setting is particularly important. A useful specification should accept a correct implementation while rejecting plausible wrong implementations. This is stricter than elaboration alone. In our specification discriminator experiment, functional scaffolding improves specification compilation and wrong implementation rejection. Functional specifications reject 61/69 wrong variants, or 88.4%, compared with 38/51, or 74.5%, for direct specifications. Correct implementation acceptance remains low for both conditions, 7/178 for functional and 6/178 for direct, which is a useful negative result: broad, faithful semantic specification generation remains substantially harder than executable code generation.

Table 10. Specification evaluation signals. BRIDGE uses these diagnostics to distinguish syntactic validity from semantic usefulness.

Signal	Purpose
Elaboration	Checks whether the specification is valid Lean.
Nonvacuity	Checks whether the specification constrains behavior.
Correct implementation acceptance	Checks whether the intended implementation satisfies the spec.
Wrong implementation rejection	Checks whether plausible incorrect code is ruled out.
Downstream usefulness	Checks whether the spec supports theorem or proof attempts.

B.4. Specification Reasoning as a Code Generation Tool

Specification reasoning also improves code generation itself. In conventional Python generation, specification guided prompting improves pass rates by up to 17.5 percentage points. This suggests that specifications help as a planning tool even when the final output is ordinary executable code. The same effect appears in Lean, but verified Lean success remains the harder bottleneck because the generated code must satisfy typing, totality, termination, and tests.

This separation is important. Specification reasoning has two roles in BRIDGE. First, it helps models plan better implementations by making edge cases and constraints explicit. Second, it produces formal artifacts that can be checked against implementations and later theorem statements. The first role improves code generation; the second supports downstream verification.

B.5. Embedding Analysis of Generated Code

To test whether structured reasoning changes the generated implementation distribution, we embed generated Python implementations using Qwen2.5 Coder embeddings and visualize them with t-SNE. Figure 6 shows that structured strategies cluster together, including Design by Contract, defensive reasoning, functional reasoning, and property based reasoning. Direct baseline generations are more dispersed.

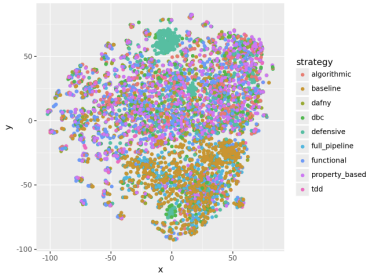


Figure 6. Embedding analysis of generated Python implementations. Structured reasoning strategies cluster more tightly, while direct baseline solutions are more dispersed.

This supports the interpretation that BRIDGE changes the implementation distribution, not only the surrounding explanation. Different scaffolds lead the model toward different implementation families, which then affects whether specifications and theorem statements can attach cleanly.

C. Theorem/Proof-Domain Details

This section reports theorem-statement and proof-attempt diagnostics for downstream verification. We distinguish three levels of success: stating a claim that elaborates in Lean, proving that claim without `sorry`, and proving a faithful semantic specification of the original task. BRIDGE improves the first two signals, while faithful end-to-end semantic verification remains outside the scope of the current system.

C.1. Multi Pathway Theorem Discovery

Theorem generation is difficult because the model must decide which mathematical facts make the implementation correct. BRIDGE uses multiple pathways to reduce the chance that a single prompt produces a shallow or implementation shaped theorem.

Table 11. Theorem discovery pathways used in BRIDGE. Each pathway surfaces a different kind of formal obligation.

Pathway	Typical theorem content
Natural language	Functional correctness and input-output relationships
Unit tests	Generalized edge case properties and invariants
Code analysis	Implementation aligned claims and structural invariants
Type guided reasoning	Properties suggested by Lean types and dependencies
Termination reasoning	Measures, recursion, and complexity bounds

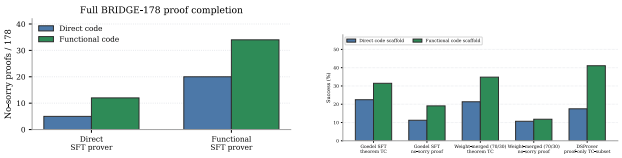
The pathways are complementary. Natural language prompts tend to produce high level correctness claims. Unit test prompts often identify boundary behavior. Code analysis prompts generate implementation-aligned statements that are easier to prove, but may be weaker than the intended task. Type guided prompts can exploit Lean's dependent structure, while termination prompts expose the measures required for recursive definitions.

C.2. Full Implementation Aligned Proof Pilot

To test whether functional scaffolds make downstream verification artifacts easier to build, we run a full 178-task implementation-aligned proof experiment. For each task, we select matched direct and functional Lean candidates. A prover then generates a specification and theorem tied to the candidate implementation and attempts to complete the proof. We score statement typechecking, strict nonvacuity, proof typechecking, proof completion without `sorry`, and implementation length.

Table 12. BRIDGE suite theorem, specification, and proof experiment on all 178 tasks. Implementation aligned specifications condition on the candidate implementation. Functional code is shorter and yields more theorem statements and completed proofs across multiple prover checkpoints.

Prover	Code Style	Stmt TC	Strict Nonvac.	Proof TC	No `sorry` Proof	Avg Code Chars
Direct SFT	Direct	11/178	2/178	5/178	5/178	532.7
Direct SFT	Functional	25/178	1/178	12/178	12/178	343.5
Functional SFT	Direct	40/178	2/178	20/178	20/178	532.7
Functional SFT	Functional	56/178	4/178	34/178	34/178	343.5
Weight merged 70/30	Direct	38/178	1/178	19/178	19/178	532.7
Weight merged 70/30	Functional	62/178	2/178	21/178	21/178	343.5



(a) Implementation aligned proof pilot across provers.

(b) Proof only DSProver success rates.

Figure 7. Proof diagnostics on the BRIDGE suite. Functional code yields more elaborating theorem statements and more completed proofs under matched prover budgets.

The implementation-aligned result is a proof-effort diagnostic: it asks whether a generated implementation gives the prover enough structure to state and discharge checkable claims. Functional code consistently gives the prover shorter implementations and more successful statement/proof attempts. However, implementation-aligned specifications can still be weaker than the natural-language task, so these results should not be read as full semantic verification.

C.3. Proof Only DSProver Check

We also run DEEPSEEK-PROVER-V2-7B as a proof-only model on implementation-aligned theorem statements that already elaborate. This isolates proof search from theorem-statement generation. The direct-code subset contains 40 typechecked statements, and the functional-code subset contains 56. DSProver completes 7/40 direct proofs and 23/56 functional proofs without `sorry`.

Table 13. Proof only DSProver check on implementation-aligned theorem statements that already elaborate. Functional code gives the proof model a larger and easier to prove theorem set.

Code Style	Input Statements	No `sorry` Proofs	Rate
Direct	40	7	17.5%
Functional	56	23	41.1%

This proof-only setting shows that functional scaffolding helps even after theorem statement generation is fixed. The proof model receives statements that already elaborate, and functional code still yields a higher proof completion rate. This suggests that the structure of the implementation itself matters for downstream proof search.

We exclude a raw VERINA DSProver adapter run from the main comparisons because the outputs often violated the expected structured format or contained placeholders, yielding 0/187 accepted proofs. We therefore report DSProver only in the setting where theorem splicing and scoring are explicitly controlled.

C.4. Weight Merging Versus SFT

We also test a simple alternative to supervised fine tuning: linear weight merging. Specifically, we merge GOEDEL-CODE-PROVER-8B, which is closer to the code generation setting, with GOEDEL-PROVER-V2-8B, which has stronger mathematical proof priors, using a 70/30 mixture. The motivation is that Lean sits between programming and mathematics. A model must produce syntactically valid code, but it must also state mathematical claims, use proof idioms, and manipulate propositions in a theorem prover. Merging with a math oriented prover may therefore improve the theorem statement side of the pipeline without requiring new training.

The results support this interpretation, but also show its limits. The merged prover improves theorem statement typechecking for functional code, reaching 62/178 statement typechecks compared with 38/178 for direct code under the same merged prover. This is the strongest statement typechecking result among the prover variants. However, it does not match the functional SFT prover on completed proofs: the merged prover completes 21/178 proofs without `sorry` for functional code, whereas the functional SFT prover completes 34/178. Thus, merging with a math model appears especially useful for producing Lean elaborating theorem statements, while BRIDGE style SFT is more effective for turning those statements into completed implementation-aligned proofs.

This suggests a useful division of labor. Weight merging is a cheap way to inject mathematical proof priors into a code prover, and it may be especially helpful for Lean because Lean artifacts require both programming structure and theorem proving structure. But SFT on functional BRIDGE traces provides a more targeted alignment signal: it teaches the model not only to speak Lean mathematics, but to connect functional implementations, specifications, theorem statements, and proof attempts in the format required by the task.

C.5. Completed Micro Proofs

The following examples illustrate the artifact stack used in the proof experiment. These are implementation-aligned proofs: they show that the generated implementation satisfies the generated contract. They are useful diagnostics for code, specification, and proof alignment, but they should not be interpreted as full semantic proofs of the original tasks.

Listing 1. A compact implementation-aligned proof.

```
def minPushes (w : String) : Nat :=
  (List.range w.length).foldl
    (fun acc i => acc + i / 8 + 1) 0

def ValidWord (w : String) : Prop :=
  1 <= w.length ∧ w.length <= 26 ∧ w.data.
    Nodup

def MinPushesSpec (w : String) (out : Nat)
    : Prop :=
  ValidWord w →
  out =
    (List.range w.length).foldl
      (fun acc i => acc + i / 8 + 1) 0

theorem minPushes_satisfies_spec (w :
    String) :
  MinPushesSpec w (minPushes w) := by
  intro _
  unfold MinPushesSpec minPushes
  rfl
```

The next example gives a lightweight invariant theorem. It is not a full task level semantic specification, but it is nonvacuous and implementation relevant.

Listing 2. A simple invariant proof attached to generated code.

```
def appendOne (xs : List Nat) (x : Nat) :
    List Nat :=
  xs ++ [x]

def AppendOneSpec (xs : List Nat) (x : Nat)
     (out : List Nat) : Prop :=
  out.length = xs.length + 1

theorem appendOne_length (xs : List Nat) (x
     : Nat) :
  AppendOneSpec xs x (appendOne xs x) := by
  unfold AppendOneSpec appendOne
  simp
```

C.6. Task Family Proof Patterns

The following examples show why theorem generation is task dependent. The code blocks are schematic proof targets. Some include `sorry` because they illustrate the obligations BRIDGE discovers rather than completed verification certificates.

Sorting and searching. Sliding window and sorting tasks often require both achievability and optimality statements.

```
def validSubseq (nums : List Int) (idxs :
    ↪ List (Fin nums.length)) : Prop :=
  idxs.Pairwise (· < ·)

def canTransform (nums : List Int) (k :
    ↪ Int)
    (idxs : List (Fin nums.length)) : Prop
    ↪ :=
  ∃ c : Int, ∀ i ∈ idxs, Int.natAbs
    ↪ (nums.get i - c) ≤ k.natAbs

theorem maximumBeauty_achievable (nums :
    ↪ List Int) (k : Int) :
  ∃ idxs : List (Fin nums.length),
    validSubseq nums idxs ∧
    canTransform nums k idxs ∧
    idxs.length = maximumBeauty nums k :=
    ↪ by
  sorry

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. Maximum Beauty theorem targets.

Combinatorial counting. Counting tasks require an exact correspondence between the implementation and a finite set of valid objects.

```
def validRemovals (nums : Array Int) :
    ↪ Finset (Nat × Nat) :=
  let n := nums.size
  let pairs := (Finset.range (n +
    ↪ 1)).product (Finset.range (n + 1))
  pairs.filter (fun p => isValidRemoval
    ↪ nums p.1 p.2)

theorem incremovable_exact_count (nums :
    ↪ Array Int) :
  incremovableSubarrayCount nums =
    ↪ (validRemovals nums).card := by
  sorry

theorem incremovable_upper_bound (nums :
    ↪ Array Int) :
  incremovableSubarrayCount nums ≤
    ↪ (nums.size + 1) ^ 2 := by
  sorry
```

Listing 4. Combinatorial theorem targets.

Tree dynamic programs. Tree algorithms often expose the hardest gap between algorithm discovery and Lean accepted verification. Models may find the right recursive decomposition, but still fail to provide a termination measure acceptable to Lean.

```
theorem dfs_correct
    (adj : Array (Array Nat)) (val : Array
    ↪ Int) (root : Nat) :
  let p := dfs adj val root none
  p.1 ≥ 0 ∧ p.2 ≥ 0 ∧ p.1 = optimalScore
    ↪ adj val root := by
  sorry
```

Listing 5. Tree dynamic programming theorem target.

These examples illustrate why theorem/proof artifacts are useful diagnostics. They identify the mathematical obligations that a future prover or tool-using agent must discharge, even when the current system cannot complete the full semantic proof.

D. Fine-Tuning and Training Details

This section reports the supervised fine-tuning protocol used for the learnability results. The experiment tests whether BRIDGE-style reasoning can be internalized by a model under a matched training setup.

D.1. Training Data

We constructed two matched 2k row OpenCodeInstruct style datasets from the same task distribution. The direct dataset pairs problem prompts with direct Lean generations. The functional dataset pairs analogous prompts with functional intermediate reasoning followed by the final Lean answer. Both datasets use the same formatting, filtering rules, train and validation split policy, and held out evaluation protocol.

D.2. Training Setup

Both variants fine tune GOEDEL-CODE-PROVER-8B with LoRA on one H100. We use LoRA rank 32, LoRA alpha 16, maximum sequence length 2048, learning rate 10^{-4}, per device batch size 1, gradient accumulation 8, bf16 precision, seed 42, and 30 total epochs. The direct and functional runs use the same base checkpoint and are evaluated on held out BRIDGE suite, VERINA, and CLEVER tasks.

Table 14. Goedel-Code-Prover-8B SFT results. Functional trace SFT improves over matched direct trace SFT on public benchmarks, with the strongest gain on VERINA. "Epoch-30" denotes the checkpoint after 30 supervised fine-tuning epochs.

Benchmark	Setting	Direct	Functional	Δ
VERINA	Base Goedel pass@1	14/187	12/187	-1.1 pp
VERINA	SFT epoch-30 pass@1	38/187	55/187	+9.1 pp
CLEVER	Base Goedel larger budget	66/161	70/161	+2.5 pp
CLEVER	SFT epoch-30 larger budget	30/161	33/161	+1.9 pp

D.3. Operational Alignment

Functional SFT also changes the shape of generated Lean code. On BRIDGE suite pass@5 generations, functional SFT outputs are shorter and cleaner than direct SFT outputs: 514.4 average extracted Lean characters versus 833.2, zero detected imports versus 876, zero `sorry`s versus 7, and 17 mutation or loop like artifacts versus 128. On VERINA, functional SFT is less imperative, with 5 detected mutation or loop like artifacts versus 40, and is substantially more accurate. On CLEVER, functional attempts solve more tasks, use fewer attempts, exhaust fewer budgets, and contain no detected imports, `sorry`s, or imperative artifacts.

E. Agentic and Robustness Analyses

This section reports additional analyses across systems and inference settings, including tool-enabled agents, temperature robustness, cross-domain repair, and theorem-guided code generation.

E.1. Tool-Enabled Agent Evaluation

We evaluate a tool-enabled coding agent on the BRIDGE suite as an exploratory upper-bound setting. Unlike the fixed-budget prompting experiments, the agent can iteratively edit files, inspect Lean errors, run tests, and use verifier feedback before producing a final answer. Under this adaptive interaction model, all 178 tasks are solved in the direct, imperative, and functional styles.

Because this setting allows repeated tool use and adaptive repair, it is not directly comparable to fixed-budget prompting. Its role is diagnostic: it shows that the benchmark is solvable by a strong interactive system, while highlighting the value of structured artifacts. Functional-style workflows reached completion about $4\times$ faster than direct workflows and produced stronger specification and theorem drafts under a combined model-judge and human-review rubric.

E.2. Temperature Robustness

Functional prompting remains consistently stronger across temperatures, suggesting that the effect is not merely a sampling artifact. It remains useful across moderate changes in decoding randomness.

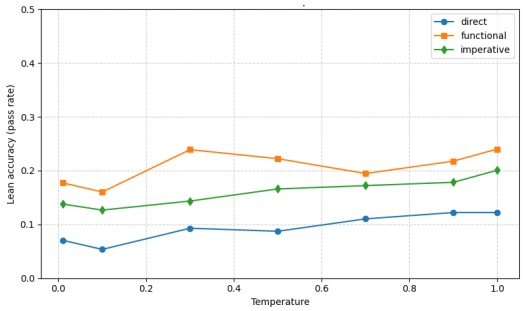


Figure 8. Temperature robustness of reasoning paradigms. Pass@5 accuracy of CLAUDE SONNET 4 as a function of sampling temperature for direct Lean, imperative Python bridge, and functional OCaml bridge prompting.

E.3. BRIDGE Inference Time Framework

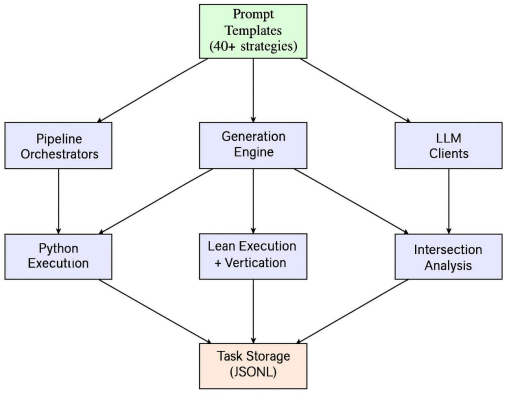


Figure 9. BRIDGE inference time framework. Code domain scaffolds produce Lean implementations, Specification domain prompts derive contracts and invariants, Theorem/Proof domain prompts generate correctness claims and proof attempts, and verifier feedback can be used for refinement.

Figure 9 gives the full inference-time view of BRIDGE and makes explicit where code, specifications, theorem statements, proof attempts, and verifier feedback enter the loop.

E.4. Error Informed Python Code Refinement

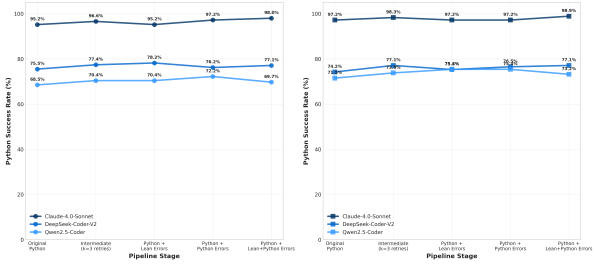


Figure 10. Python code success rates across refinement stages. Functional paradigms outperform imperative ones, and incorporating Lean error feedback improves upstream Python generation, although final Lean executable success remains the bottleneck.

We also study a pipeline where Python generation benefits

from retry mechanisms and cross-domain error signals from Lean. Lean derived errors can improve Python code quality even when Lean translation ultimately fails. The gap between high Python success and lower Lean executable success highlights the central difficulty: the bottleneck is often translating an algorithm into a total, typed, Lean accepted definition.

E.5. Failure Patterns

Functional scaffolds reduce several common Lean failure modes. Direct prompting often fails through syntax errors, missing type annotations, partial functions, and loops that do not translate cleanly into Lean. Imperative scaffolds can recover algorithmic intent, but they often require the model to invent loop invariants and state transition arguments after the fact. Functional scaffolds instead bias the model toward recursive definitions, immutable data, and compositional structure, which are closer to Lean's accepted programming style.

The most important failure classes are syntax failures, type failures, and termination or totality failures. Functional reasoning reduces syntax and type failures relative to direct generation, and it tends to produce more structurally recursive candidates. This does not eliminate termination errors, especially for graph and tree algorithms, but it makes the generated definitions closer to the form Lean can check.

E.6. Theorem Reasoning as Code Guidance

We also evaluate whether theorem oriented reasoning can improve code generation before proof search. A theorem first strategy reaches higher pass@5 than direct Lean generation in a subset of tasks, especially for mathematically heavy problems such as geometry, combinatorial counting, and bit manipulation. When both theorem guided and direct strategies succeed, their final algorithms are often similar, suggesting that theorem reasoning helps the model find the same core algorithm through a more constrained route. This supports the BRIDGE view that theorem/proof artifacts serve two roles: they identify formal obligations, and they can guide implementation search.

F. Additional Related Work

Large scale formal verification. Large scale verification projects show that full functional correctness proofs are possible for realistic systems, but they require extensive expert effort. Systems such as the SEL4 microkernel and the COMPCERT verified compiler demonstrate the strength of machine checked proofs, while also illustrating why scalable formal verification remains difficult in practice. Solver based tools such as DAFNY, BOOGIE, WHY3, and VERUS (Lattuada et al., 2023) lower this burden by translating annotated programs into verification conditions that can often be discharged automatically. These systems motivate BRIDGE's focus on the artifacts that appear before automation succeeds: executable code, formal specifications, theorem statements, and proof attempts must be structured well enough for a verifier or prover to use them.

Solver based verification with LLMs. Solver based verification is often more forgiving than interactive theorem proving because many local proof obligations can be handled automatically once the right annotations and specifications are present. Benchmarks such as DAFNYBENCH and systems such as AUTOSPEC show that LLMs can perform well in this setting, especially when the task is to produce code and contracts that an SMT backend can finish (Loughridge et al., 2024; Wen et al., 2024). CLOVER uses a closed loop pipeline in which an LLM generates code, documentation, and annotations, then applies consistency checks and solver based verification to decide acceptance (Sun et al., 2024). These results suggest that when the backend can close the proof gap, the main difficulty shifts toward generating correct, complete, and nonvacuous specifications. BRIDGE complements this line of work by studying how the reasoning scaffold changes the generated code and formal artifacts before the solver or prover takes over.

Specification synthesis and intent alignment. Recent work has begun to treat formal specification generation as a first class task rather than a byproduct of code generation. VERISPECGEN targets Lean specification synthesis on VERINA by decomposing natural language intent into atomic requirements, generating requirement targeted tests, and using traceability maps to localize and repair failed clauses (Ye et al., 2026). This is closely related to BRIDGE's Specification domain. The difference is emphasis: specification synthesis systems focus on improving the specification itself, while BRIDGE studies how specifications interact with executable code, theorem statements, and proof attempts through a shared reasoning scaffold. In particular, BRIDGE evaluates specifications not only by elaboration, but also by nonvacuity, correct implementation acceptance, wrong implementation rejection, and downstream usefulness.

Interactive theorem proving with LLMs. Interactive theorem provers expose sharper limits for LLM based verification. Public Lean benchmarks such as VERINA and CLEVER show that models can generate plausible code, specifications, theorem statements, and proof attempts, but still struggle to connect them into reliable verified artifacts (Ye et al., 2025; Thakur et al., 2025). In these settings, success requires more than local annotation generation: the implementation must be expressible in Lean, the specification must capture the intended behavior, the theorem state-

ment must connect the two, and the proof must be accepted by the kernel. Prior benchmark results show that code and specification generation are substantially easier than full proof completion, and that performance drops sharply when proofs must compose across richer artifacts. BRIDGE is designed around this bottleneck. Rather than treating proof failure as an isolated prover problem, it treats the entire code, specification, theorem, and proof chain as an artifact alignment problem.

Compositional and cross system verification benchmarks. Recent benchmarks also show that local verification success does not automatically scale to larger verification tasks. DAFNYCOMP studies compositional specification problems in Dafny, while the VERICODING BENCHMARK studies verification oriented coding across systems (Xu et al., 2025; Bursuc et al., 2025). These settings emphasize that even when an LLM can generate locally plausible code or annotations, composing specifications and proofs across modules remains difficult. This limitation is directly relevant to BRIDGE. The present work focuses on the front end of verified synthesis and mostly algorithmic tasks, but the same artifact alignment problem becomes more important in long horizon, multi module, or cross system verification settings.

Multi view program representations. A related line of work represents programs through multiple semantic views, including source code, natural language descriptions, syntax trees, data flow, documentation, and generated explanations. Models such as CODEBERT and GRAPHCODEBERT show that combining code with structural or textual views can improve program understanding and generation (Feng et al., 2020; Guo et al., 2021). BRIDGE specializes this multi view idea to formal verification. The views are not only code and text, but executable implementations, specifications, theorem statements, and proof attempts. Unlike ordinary program representations, these artifacts must survive formal checks and remain mutually meaningful inside a proof assistant or solver-backed verification system.

G. Prompt System and Instantiations

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* (Lean implementations), *Specification Domain* (formal contracts), and *Theorem/Proof Domain* (mathematical property discovery and proof attempts). We present unified templates with strategy-specific content marked as `[SWAP]` placeholders.

Prompt Construction Procedure. To construct a complete prompt from a template, we follow a systematic substitution process. First, the natural language problem description (from the LeetCode dataset) is prepended to the template. Next, `[SWAP:STRATEGY]` is replaced with one of the strategy options listed in the template (e.g., "Direct", "Python Bridge", "Haskell Functional"). The `[SWAP:REASONING_STYLE]` placeholder is replaced with the corresponding reasoning approach name (e.g., "Functional Reasoning" for Haskell, "Imperative Reasoning" for C++). The `[SWAP:REASONING_DETAILS]` placeholder is filled with strategy-specific reasoning guidelines provided in the subsequent template boxes. Metadata placeholders such as `InputType`, `OutputType`, and `{{ function_name }}` are extracted from the problem statement: `InputType` and `OutputType` are inferred from the problem's input/output specifications, while `{{ function_name }}` is derived from the problem's function signature or a canonical name based on the problem description. All placeholders are replaced before sending the prompt to the model, ensuring complete and consistent prompt construction across all strategies.

G.1. Code Domain Strategies (Lean)

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

BRIDGE Code Domain Template

Instructions. Solve the task by applying the reasoning paradigm in `[SWAP:STRATEGY]` and translate the solution to LEAN4.

`[SWAP:STRATEGY]` options:

- **Direct**: Solve directly in LEAN4
- **Python Bridge**: Reason in Python logic first, then translate
- **Haskell Functional**: Pattern matching, recursion, type-driven development
- **OCaml Type-Guided**: Type safety and mathematical invariants
- **Double Lean**: Mathematical reasoning in Lean style, then implement
- **C++ Imperative**: Procedural logic, then translate to functional LEAN4

```
Step 1: [SWAP:REASONING_STYLE] — [SWAP:REASONING_DETAILS]
```

Note: `[SWAP:REASONING_STYLE]` is replaced with the strategy name (e.g., "Functional Reasoning", "Imperative Reasoning") corresponding to the selected `[SWAP:STRATEGY]` option.

```
Step 2: Lean4 Translation

import Std
import Mathlib
```

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

BRIDGE-I (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 and built-ins
- Simple, readable solutions

BRIDGE (Haskell / OCaml / Double Lean)

[SWAP:REASONING_DETAILS] for Haskell:

- Pattern matching and structural recursion
- Functional composition and list processing
- Type signatures and algebraic data types

[SWAP:REASONING_DETAILS] for OCaml:

- Immutable data structures and recursion
- Algebraic data types, variants, records
- Tail recursion and compositional design

[SWAP:REASONING_DETAILS] for Double Lean:

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

G.2. Specification Domain Strategies (Python)

Unified Specification Domain Template

Instructions. Solve the problem using [SWAP:STRATEGY] reasoning, then implement the solution in Python with formal contracts.

[SWAP:STRATEGY] options:

- **Direct**: Write Python solution directly without intermediate reasoning
- **Design by Contract**: Define preconditions, postconditions, and invariants
- **Dafny-style**: Use requires/ensures clauses and verification conditions
- **Property-Based Testing**: Identify universal mathematical properties
- **Functional Programming**: Use immutable data and pure functions
- **Defensive Programming**: Validate inputs and handle errors robustly
- **Algorithmic Thinking**: Decompose into algorithmic subproblems
- **Test-Driven Development**: Design by reasoning about test cases first

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

Step 2: Python Implementation — Translate reasoning into 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 [SWAP:STRATEGY] reasoning to
    create robust code with contracts

[SWAP:CONTRACT_DECORATORS]
def function_name(parameters):
    """
    [SWAP:DOCSTRING_WITH_CONTRACTS]
    """
    # Implementation guided by [SWAP:
    STRATEGY] reasoning
    pass
```

G.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, inspired by 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
    """
    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, emphasizing algebraic invariants and bounds.

```
import deal
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
    - Monotonicity: adding positives
    increases result
    - Non-negativity: result >= 0
    """
    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 first, following the Red–Green–Refactor cycle.

```
import deal
from typing import List

@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: define failing tests
    - Green: minimal implementation
    - Refactor: optimize safely
    """
    if not isinstance(data, list) or len
    (data) == 0:
        return 0

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

def comprehensive_test_suite():
    assert tdd_solution([1, 2, 3]) == 6
    assert tdd_solution([]) == 0
    assert tdd_solution([-1, -2]) == 0
    assert tdd_solution([1, "2", None])
    == 1
```

Prompt — Defensive Programming Reasoning

Reasoning Focus. Implement comprehensive input validation, error handling, and fail-safe behaviors to ensure robustness.

```
import deal
import logging
from typing import List, Union
```

```python
@deal.chain(
    deal.safe,
    deal.post(lambda result: isinstance(
    result, int) and result >= 0)
)
def defensive_solution(data: Union[List[
    int], None]) -> int:
    """
    Defensive Strategies:
    - Validate inputs
    - Handle malformed data gracefully
    - Provide safe fallback behavior
    """
    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)]
        return sum(x for x in valid_ints
     if x >= 0)
    except Exception as e:
        logging.error(f"Computation
    error: {e}")
        return 0
```

Prompt — Algorithmic Thinking Reasoning

Reasoning Focus. Systematically decompose problems into algorithmic subcomponents, emphasizing structure, correctness, and complexity analysis.

```python
import deal
from typing import List

@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
    2. Core computation
    3. Post-processing validation

    Complexity: O(n) time, O(1) space
    """
    if not data:
        return 0

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

G.3. Theorem/Proof Domain Strategies

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

Unified Theorem Statements Template

`[SWAP:PATHWAY]` options:

- **Natural Language**: Extract properties directly from the problem description
- **Unit Tests**: Generalize mathematical claims from test patterns
- **Code Analysis**: Reverse-engineer properties from implementation structure
- **Type-Guided**: Leverage refined 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]

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
```

G.3.1. Concrete Pathway Examples

Prompt — Natural Language → Lean Theorems

Analysis Focus. Extract key mathematical properties and correctness conditions from the natural language statement and express them as Lean theorems about `{{ function_name }}`. 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 relationships and invariants they enforce, then express these as Lean theorems about `{{ function_name }}`. Implement `{{ function_name }}` and generate *Theorem 1* (a property generalized from tests) and *Theorem 2* (an invariant relating multiple calls).

```
-- 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 }}`. Derive theorems that verify algorithmic correctness and implementation-specific invariants.

```
-- 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 about termination and computational complexity for `{{ function_name }}`. Implement `{{ function_name }}` with termination in mind and produce termination-focused statements.

```
-- 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 leverage refined types and mathematical structure to prove stronger correctness and invariants.

```
-- 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
```

G.4. Iterative Improvement Strategies

BRIDGE implements iterative repair through structured error diagnosis and targeted correction. This enables systematic refinement of failed solutions while keeping the error signal tied to the generated artifact.

Prompt — Structured Error-Guided Repair

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, and edge case failures.
- **Step 2: Root Cause Analysis.** Determine why the errors occurred and which 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 re-verify correctness.

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

# Your previous solution failed.
# Analyze the errors and generate an
   improved solution.

# Previous Python / Lean Code (FAILED)
{{ previous_python_or_lean_code }}

# Error Messages and Feedback
{{ python_or_lean_errors }}
```

Meta-Analysis Process

Analysis Methodology. Examine theorems generated across all five proof 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": "
import Std
import Mathlib

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

theorem solution\_bounds (input :
    InputType) :
  0 \\le solution input \\wedge solution
     input \\le upper\\_bound input := by
```

```
  sorry
  -- omitted for brevity
"
}
```

H. Theorem and Proof Examples

We present selected examples illustrating why generating good theorem/proof artifacts is hard and how BRIDGE addresses them. Each highlights how different algorithmic categories require distinct reasoning strategies, motivating our three-domain approach to formal verification. All code examples and theorem statements shown in this section were generated by CLAUDE SONNET 4 using the BRIDGE framework; proof obligations are shown with `sorry` when they are intended targets rather than completed proofs.

H.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 6. 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 7. Correctness Theorems

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

H.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 8. 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 9. Combinatorial Theorems

Correctness in this setting depends on exhaustive case reasoning, contrasting with the structural reasoning used for sorting.

H.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 : Nat) (parent : Option Nat) :
    ↪ Prod Int Int :=
  let children := (adj[node]!).filter (fun
    ↪ c => parent != some c)
  if children.isEmpty then
    (0, val[node]!)
  else
    let res    := children.map (fun c =>
    ↪ dfs adj val c (some node))
    let scores := (res.map (fun x =>
    ↪ x.1)).toList.sum
```

```
    let costs  := (res.map (fun x =>
    ↪ x.2)).toList.sum
    (max (scores + val[node]!) (scores -
    ↪ costs + val[node]!),
     val[node]! )
```

Listing 10. 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 none
  in p.1 ≥ 0 ∧ p.2 ≥ 0 ∧
     p.1 = optimalScore adj val root := by
  sorry
```

Listing 11. Tree DP Theorems

Listing 11 gives 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 a target specification rather than a completed verification certificate.

I. Qualitative Examples Across Benchmarks

This appendix provides a small, human-readable gallery of generated artifacts. We include examples where functional scaffolding changes (i) whether specifications elaborate, (ii) whether downstream theorem/proof artifacts are completed, and (iii) behavior on public benchmarks. Examples are illustrative and are not intended to replace the aggregate metrics.

I.1. BRIDGE-suite: specification discriminative power

Task 2: `minimumPushes`. Direct spec: compiled=False, correct-accept=None, wrong-rejected=0. Functional spec: compiled=True, correct-accept=True, wrong-rejected=3.

Direct compile error (first):

```
type mismatch
  fun x => ?m.5246
has type
  (x : ?m.5242) → ?m.5247 x : Sort (imax ?
    u.5241 ?u.5244)
but is expected to have type
  Int : Type
```

Functional spec (excerpt):

```
def postcondition (word : String) (result :
     Int) : Bool :=
  let chars := word.toList
  let counts := chars.foldl (fun acc c =>
    acc.insert c (acc.findD c 0 + 1)) (
    HashMap.mk String Nat)
  let key1 := counts.findD "a" 0
  let key2 := counts.findD "b" 0
  let key3 := counts.findD "c" 0
  let key4 := counts.findD "d" 0
  let key5 := counts.findD "e" 0
  let key6 := counts.findD "f" 0
  let key7 := counts.findD "g" 0
  let key8 := counts.findD "h" 0
  let key9 := counts.findD "i" 0
  let key10 := counts.findD "j" 0
  let key11 := counts.findD "k" 0
  let key12 := counts.findD "l" 0
  let key13 := counts.findD "m" 0
  let key14 := counts.findD "n" 0
  let key15 := counts.findD "o" 0
  let key16 := counts.findD "p" 0
  let key17 := counts.findD "q" 0
  let key18 := counts.findD "r" 0
  let key19 := counts.findD "s" 0
  let key20 := counts.findD "t" 0
  let key21 := counts.findD "u" 0
  let key22 := counts.findD "v" 0
  let key23 := counts.findD "w" 0
  let key24 := counts.findD "x" 0
  let key25 := counts.findD "y" 0
  let key26 := counts.findD "z" 0
  let cost := key1 + key2 + key3 + key4 +
    key5 + key6 + key7 + key8 + key9 +
    key10 + key11 + key12 + key13 + key14 +
     key15 + key16 + key17 + key18 + key19 +
     key20 + key21 + key22 + key23 + key24 +
     key25 + key26
  decide (cost = result)
```

Task 8: `maxFrequencyElements`. Direct spec: compiled=False, correct-accept=None, wrong-rejected=0. Functional spec: compiled=True, correct-accept=True, wrong-rejected=3.

Direct compile error (first):

```
Invalid field `findD`: The environment does
     not contain `List.findD`
  acc
has type
  List (Int × Nat)
```

Functional spec (excerpt):

```
def postcondition (nums : List Int) (result
    : Int) : Bool :=
  let freqs : List Nat := nums.map (fun
    value => (nums.filter (fun other =>
    other = value)).length)
  let maxFreq : Nat := freqs.foldl (fun
    best freq => if freq > best then freq
    else best) 0
  let countMax : Nat := (freqs.filter (fun
    freq => freq = maxFreq)).length
  decide (countMax = result.toNat)
```

I.2. BRIDGE-suite: implementation-aligned theorem/proof artifacts

The examples below come from the full BRIDGE-suite proof pilot. These are *implementation-aligned* statements (the spec is conditioned on the candidate implementation), so they primarily diagnose proof tractability rather than semantic specification fidelity.

Task 128. Functional (completed no-`sorry` proof; excerpt):

```
def furthestDistanceFromOrigin (moves :
    String) : Int :=
  let right := (moves.toList.filter (fun c
    => c = 'R')).length
  let left := (moves.toList.filter (fun c
    => c = 'L')).length
  let underscore := (moves.toList.filter (
    fun c => c = '_')).length
  Int.ofNat (right + underscore - left)

def furthestDistanceFromOrigin_spec (moves
    : String) (result : Int) : Prop :=
  let right := (moves.toList.filter (fun c
    => c = 'R')).length
  let left := (moves.toList.filter (fun c
    => c = 'L')).length
  let underscore := (moves.toList.filter (
    fun c => c = '_')).length
  result = Int.ofNat (right + underscore -
    left)

theorem furthestDistanceFromOrigin_correct
    (moves : String) :
    furthestDistanceFromOrigin_spec moves (
    furthestDistanceFromOrigin moves) := by
  unfold furthestDistanceFromOrigin
    furthestDistanceFromOrigin_spec
  simp
```

Direct (statement; excerpt):

```
def furthestDistanceFromOrigin (moves :
    String) : Int :=
  let chars := moves.toList
  let (total, current) :=
    chars.foldl (fun (total, current) c =>
      match c with
      | 'L' => (total, current - 1)
      | 'R' => (total, current + 1)
      | _ =>
          let newTotal := total + 1
          let newCurrent := if newTotal % 2
     == 0 then newTotal / 2 else newTotal /
     2 + 1
          (newTotal, newCurrent)) (0, 0)
  total.natAbs.toInt

# Test cases
-- #eval furthestDistanceFromOrigin "
    L_RL__R"  -- Expected: 3
-- #eval furthestDistanceFromOrigin "
    _R__LL_"  -- Expected: 5
-- #eval furthestDistanceFromOrigin "
    _______"  -- Expected: 7

def furthestDistanceFromOrigin_spec (moves
    : String) (result : Int) : Prop :=
  let chars := moves.toList
  let (total, current) :=
    chars.foldl (fun (total, current) c =>
      match c with
      | 'L' => (total, current - 1)
      | 'R' => (total, current + 1)
      | _ =>
          let newTotal := total + 1
          let newCurrent := if newTotal % 2
     == 0 then newTotal / 2 else newTotal /
     2 + 1
          (newTotal, newCurrent)) (0, 0)
  result = total.natAbs.toInt

theorem furthestDistanceFromOrigin_correct
    (moves : String) :
    furthestDistanceFromOrigin_spec moves (
    furthestDistanceFromOrigin moves) := by
  sorry
```

Task 71. Functional (completed no-`sorry` proof; excerpt):

```
def maxBalancedSubsequenceSum (nums : List
    Int) : Int :=
  match nums with
  | [] => 0
  | x :: _ => x

def maxBalancedSubsequenceSum_spec (nums :
    List Int) (result : Int) : Prop :=
  match nums with
  | [] => result = 0
  | x :: _ => result = x
```

```
theorem maxBalancedSubsequenceSum_correct (
    nums : List Int) :
    maxBalancedSubsequenceSum_spec nums (
    maxBalancedSubsequenceSum nums) := by
  cases nums with
  | nil => simp [maxBalancedSubsequenceSum,
     maxBalancedSubsequenceSum_spec]
  | cons x xs => simp [
    maxBalancedSubsequenceSum,
    maxBalancedSubsequenceSum_spec]
```

Direct (statement; excerpt):

```
/-- Transform the problem into a form
    suitable for applying `longest non-
    decreasing subsequence`
algorithms by transforming the array into a
     list of pairs \((i, nums[i] - i)\).
    This transformation
turns the condition \(nums[i] - nums[j] \
    geq i - j\) for \(i > j\) into \(nums[i
    ] - i \geq nums[j] - j\),
which can be handled by standard
    subsequence algorithms. -/
def transform (nums : List Int) : List (Nat
     × Int) :=
  nums.enum.map (fun (i, x) => (i, x - Int.
    ofNat i))

/-- Maximum sum of a non-decreasing
    subsequence in terms of the transformed
     values. -/
def maxBalancedSum (nums : List Int) : Int
    :=
  let transformed := transform nums
  let rec dp : List (Nat × Int) → List (
    Nat × Int) → Int → Int
    | [], _, best => best
    | x :: xs, seen, best =>
        let currentValue := x.2
        let bestSoFar := seen.foldl (fun
    acc p => max acc (p.2 + currentValue))
    currentValue
        let newBest := max best bestSoFar
        dp xs (x :: seen) newBest
  dp transformed [] (List.foldl (fun acc x
    => max acc x.2) (-1000000000)
    transformed)

def maxBalancedSubsequenceSum (nums : List
    Int) : Int :=
  maxBalancedSum nums

-- #eval maxBalancedSubsequenceSum [3, 3,
    5, 6] -- Expected: 14
-- #eval maxBalancedSubsequenceSum [5, -1,
    -3, 8] -- Expected: 13
-- #eval maxBalancedSubsequenceSum [-2, -1]
     -- Expected: -1

def maxBalancedSubsequenceSum_spec (nums :
    List Int) (result : Int) : Prop :=
  result = maxBalancedSubsequenceSum nums
```

```
theorem maxBalancedSubsequenceSum_correct (
    nums : List Int) :
    maxBalancedSubsequenceSum_spec nums (
    maxBalancedSubsequenceSum nums) := by
  rfl
```

I.3. VERINA: Lean code generation artifacts

verina_advanced_17. Direct compile=False; functional compile=True.

Direct generation (excerpt):

```
def insertSorted (x : Int) (l : List Int) :
     List Int :=
  match l with
  | [] => [x]
  | y :: ys =>
    if x ≤ y then
      x :: y :: ys
    else
      y :: insertSorted x ys

def insertionSortHelper (l : List Int) (
    sorted : List Int) : List Int :=
  match l with
  | [] => sorted
  | x :: xs => insertionSortHelper xs (
    insertSorted x sorted)

insertionSortHelper l []
```

Functional generation (excerpt):

```
def insertAtSorted (x : Int) (l : List Int)
     : List Int :=
  match l with
  | [] => [x]
  | y :: ys => if x ≤ y then x :: y :: ys
    else y :: insertAtSorted x ys

def insertAll (l : List Int) (acc : List
    Int) : List Int :=
  match l with
  | [] => acc
  | x :: xs => insertAll xs (insertAtSorted
     x acc)

insertAll l []
```

verina_advanced_42. Direct compile=False; functional compile=True.

Direct generation (excerpt):

```
def minPrice (xs : List Nat) : Nat :=
  xs.foldl (. min .) (if xs.isEmpty then 0
    else xs.head)

def scanMin (xs : List Nat) : List Nat :=
  xs.scanl min (if xs.isEmpty then 0 else
    xs.head)
```

```
if prices.isEmpty then
    0
  else
    let mins := scanMin prices
    let zipped := List.zip prices mins
    let profits := zipped.map (fun (p, m)
    => p - m)
    profits.foldl max 0
```

Functional generation (excerpt):

```
match prices with
  | [] => 0
  | _ =>
    let rec loop xs (minPrice : Nat) (
    maxProfit : Nat) : Nat :=
      match xs with
      | [] => maxProfit
      | p::ps =>
        let newProfit := if p > minPrice
    then p - minPrice else maxProfit
        let newMin := if p < minPrice then
    p else minPrice
        loop ps newMin newProfit
    loop prices 99999999 0
```

I.4. CLEVER: budgeted Lean implementation search

Problem 158. Direct: ok=False, error=`Timeout`. Functional: ok=`True` (compiled; no `sorry`).

Functional implementation (winning attempt; excerpt):

```
match words with
  | [] => ""
  | x :: xs =>
    let step := fun (bestSoFar: String) (
    current: String) =>
      let bestUnique := bestSoFar.toList.
    toFinset.card
      let currentUnique := current.toList.
    toFinset.card
      if bestUnique < currentUnique then
        current
      else if bestUnique > currentUnique
    then
        bestSoFar
      else
        if bestSoFar < current then
          bestSoFar
        else
          current
    words.foldl step x
```

I.5. DafnyBench: solver-backed verification (subset pilot)

Test 038. Direct: ok=False; functional: ok=True.

Functional Dafny attempt (excerpt):

```
method double_array_elements(s: array<int>)
  modifies s
  ensures forall i :: 0 <= i < s.Length ==>
     s[i] == 2 * old(s[i])
{
  var i := 0;
  var s_old := s[..]; // Capture the
    initial state of the array
  while (i < s.Length)
    invariant 0 <= i <= s.Length
    invariant forall j :: 0 <= j < i ==> s[
    j] == 2 * s_old[j]
    invariant forall j :: i <= j < s.Length
     ==> s[j] == s_old[j]
  {
    s[i] := 2 * s[i];
    i := i + 1;
  }
}
```

Direct Dafny attempt (excerpt):

```
method double_array_elements(s: array<int>)
  modifies s
  ensures forall i :: 0 <= i < s.Length ==>
     s[i] == 2 * old(s[i])
{
  var i := 0;
  while (i < s.Length)
    invariant 0 <= i <= s.Length
    invariant forall j :: 0 <= j < i ==> s[
    j] == 2 * old(s[j])
    decreases s.Length - i
  {
    s[i] := 2 * s[i];
    i := i + 1;
  }
}
```