FormalEvolve: Neuro-Symbolic Evolutionary Search for Diverse and Prover-Effective Autoformalization

Haijian Lu [1 2] **Wei Wang** [2 *] **Jing Liu** [1]

Abstract

Autoformalization aims to translate natural-language mathematics into compilable, machine-checkable statements. However, semantic consistency does not imply prover effectiveness: even semantically consistent formalizations can differ substantially in proof-search cost and success rate. In this work, we formulate autoformalization as a budgeted, test-time search for semantically consistent repertoires, and propose FormalEvolve, a compilation-gated neuro-symbolic evolutionary framework. FormalEvolve generates diverse candidates via LLM-driven mutation and crossover with bounded patch repair, while symbolic Abstract Syntax Tree (AST) rewrite operations further inject structural diversity. On CombiBench and ProofNet, under a strict generator-call budget of T=100, FormalEvolve reaches semantic hit rates (SH@100) of 58.0% and 84.9%, and reduces cross-problem concentration of semantic successes (lower Gini). Under a fixed prover budget, FormalEvolve also improves downstream proving performance on CombiBench. Code will be released publicly.

1. Introduction

Interactive theorem provers such as Lean 4 (Moura & Ullrich, 2021) and libraries such as Mathlib (The mathlib Community, 2020) make large-scale machine-checkable mathematics practical. End-to-end systems still hinge on high-quality formal statements: an ill-typed or unfaithful statement can derail downstream proving and waste substantial compute. As LLM-based theorem proving improves (Polu & Sutskever, 2020; Han et al., 2022; Yang et al., 2023; Song et al., 2024), reliable *autoformalization*—translating informal mathematical statements into formal, machine-checkable Lean 4 statements—becomes increasingly important.

[1]School of Artificial Intelligence, Xidian University, Xi'an, China [2]Beijing Institute for General Artificial Intelligence, Beijing, China. Correspondence to: Wei Wang <wangwei@bigai.ai>.

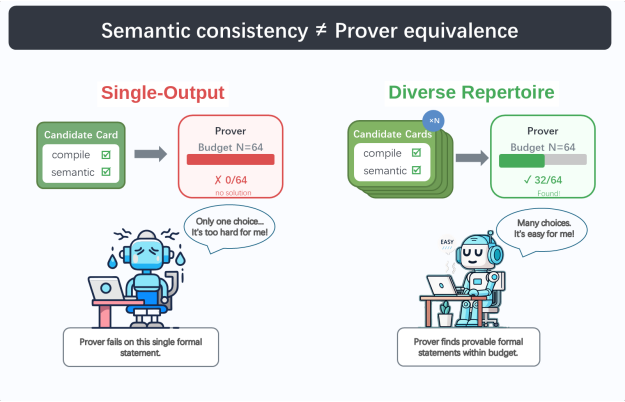


Figure 1. Motivating illustration under a fixed prover and attempt budget (N=64). Even when candidate statements compile and are judged semantically consistent, prover outcomes can vary substantially. Constructing a diverse repertoire hedges against this sensitivity and increases the chance that at least one provable statement is found within budget.

To hedge against this prover fragility under fixed generator-call and prover budgets, we propose FormalEvolve, a compilation-gated evolutionary search framework that explicitly constructs a diverse repertoire of feasible statements. Figure 2 gives a high-level overview.

Large language models can draft Lean 4 statements with non-trivial compilation success (Wu et al., 2022), and benchmarks such as ProofNet provide paired informal statements and ground-truth formalizations (Azerbayev et al., 2023). Yet reliability remains fragile. Typing and library context define a narrow feasible region, and compilation alone does not prevent semantic drift, motivating dedicated judges such as CriticLean (Peng et al., 2025). Many problems admit multiple semantically consistent formalizations; selecting a single output often collapses this space. Even semantics-intended reformulations can change prover behavior (Zhao et al., 2025), and large-scale provers highlight the role of scale and diversity in both training data and model behavior (Lin et al., 2025a;b).

Semantic consistency $\not\Rightarrow$ prover equivalence.

Within the compilation-feasible region, semantic fidelity and proving difficulty are distinct, and improving one does not

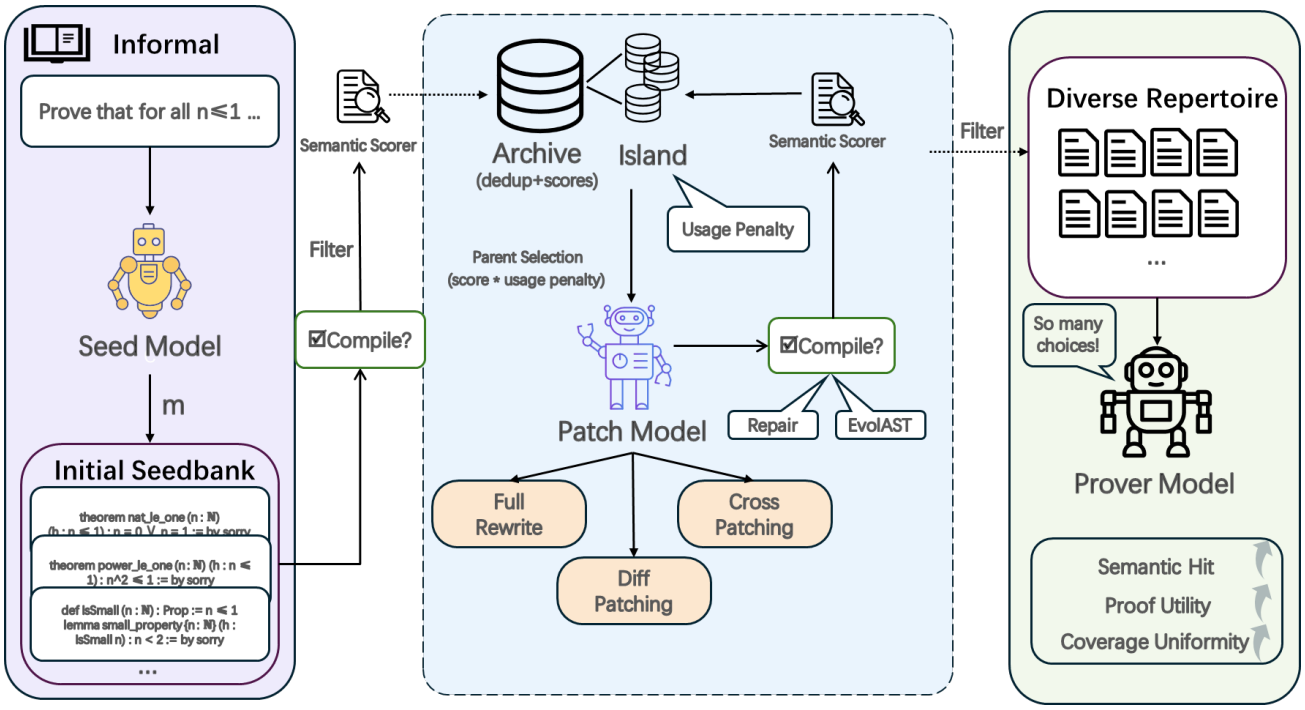


Figure 2. Framework overview of FormalEvolve. A seed model produces an initial seedbank (debited as generator calls) that initializes a compilation-feasible archive. Candidates are stored with semantic scores and selected with usage penalties; a patch model proposes edits (full rewrite, diff patching, and cross patching) and triggers bounded repair or EvolAST fallbacks on compilation failures. Semantic scoring filters the archive into a diverse repertoire for downstream proving and evaluation.

guarantee improving the other. We call candidates *prover-equivalent* if they have comparable proof-search difficulty and success probability under a fixed prover and budget. Under fixed test-time and prover budgets, our goal is therefore a repertoire that spans this trade-off, rather than a single scalarized optimum.

In practice, we treat statement generation as **compilation-gated repertoire search**: within a generator-call budget, we construct a deduplicated set of compilable, judge-consistent candidates, and evaluate downstream proving performance under a fixed prover protocol and attempt budget.

Recent systems mitigate semantic drift by interleaving autoformalization with iteration, tool feedback, and grounding (Chen et al., 2025; Guo et al., 2025; Wang et al., 2025b). We instead study what can be done *within* a strict generator-call budget: optimize *repertoire-level* reliability and coverage rather than refining a small number of trajectories. FormalEvolve maintains a compilation-feasible archive and allocates its budget across candidates using usage-penalized selection and duplicate-aware mechanisms, reducing a common failure mode where semantic successes concentrate on a small subset of easy problems.

FormalEvolve is a **compilation-gated, constrained-diversity search** procedure: it evolves candidate statements via LLM-driven patching with bounded repair, and applies a conservative, semantics-intended AST rewrite operator (EvolAST-style; (Tian et al., 2025)) as a zero-call diversification fallback. Empirically, at $T = 100$ calls, FormalEvolve achieves SH@100=0.58 on CombiBench (versus 0.46 for Kimina Compile+Semantic Repair and 0.53 for a hybrid sampling control with Qwen3 repair), and improves theorem-complete@64 from 8/100 to 13/100 relative to Kimina Compile+Semantic Repair, while also reducing cross-problem concentration (Table 1; Table 2).

We make three primary contributions:

1. **Problem Modeling and Protocol:** We formalize Lean 4 autoformalization as budgeted, compilation-gated repertoire search and introduce budget-auditable metrics that link statement generation (coverage and concentration) to downstream proving performance.
2. **Search Mechanism:** We propose FormalEvolve, a compile-gated evolutionary search procedure that combines archive-based LLM patching with bounded repair and a zero-call EvolAST-style rewrite operator for diversification without consuming additional generator calls.
3. **Empirical Evidence:** We show higher semantic coverage on ProofNet and CombiBench under fixed budgets, improved downstream proving performance on CombiBench, and analyze which components drive gains.

2. Related Work

Autoformalization pipelines and metrics. Large language models have enabled autoformalization at scale, with systematic evaluation on curated benchmarks (Wu et al., 2022; Azerbayev et al., 2023; Zheng et al., 2022). Reliability remains a bottleneck: compilation is a weak proxy for semantic consistency, motivating dedicated semantic judges and alignment evaluators (Peng et al., 2025; Lu et al., 2024). Recent systems further improve semantic fidelity by interleaving iteration, tool feedback, and grounding (Chen et al., 2025; Guo et al., 2025; Wang et al., 2025b; Yang et al., 2023). In contrast to approaches that primarily refine a small number of trajectories or filter a single output, we study the fixed-budget setting and optimize *repertoire-level* reliability and coverage under the same generator-call budget.

Statement sensitivity and evaluator limitations. Even among semantically consistent candidates, proving difficulty can vary dramatically under minor, semantics-intended reformulations (Zhao et al., 2025), and semantic evaluation itself can be imperfect (Chen et al., 2025). These limitations motivate hedging against fragility by constructing diverse statement repertoires and assessing their downstream proving performance under fixed prover budgets, rather than treating statement generation as a single-output decision.

Test-time search and quality-diversity. Test-time search and evolutionary optimization with LLMs have been explored for sample-efficient program and algorithm improvement (Novikov et al., 2025; Lange et al., 2025; Liu et al., 2024). Quality-diversity methods emphasize constructing repertoires of high-quality solutions rather than optimizing a single output (Lehman & Stanley, 2011; Mouret & Clune, 2015), and multi-criteria evolutionary optimization provides a complementary perspective on trade-offs (Deb et al., 2002). EvolProver evolves formalized problems via symmetry and difficulty, including semantics-preserving AST rewrites, to augment training data for provers (Tian et al., 2025). Our focus differs: we treat compilation as a hard feasibility gate, debit all generator-side operators against the same generator-call budget, and use EvolAST-style conservative rewrites as *online* diversity operators that consume no generator calls for *test-time* statement search.

3. Method

FormalEvolve is a per-problem test-time search procedure for Lean 4 statement candidates under a strict generator-call budget T (Section 4). Given an informal statement x, it proposes and evaluates candidates under strict accounting. We now define the search state, selection rule, and operators used in Algorithm 1; Figure 2 provides a visual overview. At a high level, FormalEvolve maintains a compilation-feasible archive, allocates the generator-call budget across candidates via usage-penalized selection, and expands the archive via LLM patching with bounded repair (with a call-free EvolAST fallback for structural diversification). Compilation induces a highly non-convex feasible region; under strict budgets, naive sampling wastes many calls outside feasibility and can still repeatedly rediscover near-duplicates within feasibility. FormalEvolve mitigates this by gating on compilation, salvaging near-feasible candidates via bounded repair, and adding lightweight diversity operators to reduce collapse. Our reported output is the deduplicated semantically consistent repertoire ($C{=}1$ and $J{=}1$) generated within budget, which we use for SH@T and downstream proving.

3.1. Search Procedure and State

Basic objects and gates. We search over Lean 4 candidates of the form $c = (\hat{h}, y)$, where $\hat{h}$ is an import/header context and y is a single Lean 4 declaration with a placeholder proof. In practice, $\hat{h}$ can be taken from the dataset (fixed context) or generated/modified by the model (context diversity); we make this choice explicit in the experimental protocol. We evaluate candidates using compilation feasibility $C(c) \in \{0, 1\}$ and semantic consistency $J(c) \in \{0, 1\}$ (instantiated by an LLM judge in Section 4). Compilation is a hard feasibility gate: only candidates with $C(c) = 1$ are eligible to enter the archive and be reused as parents/inspirations. We treat compilation/elaboration as verifier feedback: it both defines feasibility and provides the primary error feedback for bounded compilation repair. During search, the archive stores unique compilation-feasible candidates regardless of $J(c)$ (i.e., it may include semantically inconsistent stepping stones) and is used only for parent/context sampling. For reporting SH@T and downstream proving, we define the *semantically consistent repertoire* as the deduplicated set of candidates generated within budget that satisfy both $C(c) = 1$ and $J(c) = 1$. FormalEvolve uses C and J during test-time search and bounded repairs; downstream proving is evaluated only when reporting proof utility. All generator-side LLM calls (seeds, patches, and repairs) are debited against the same per-problem generator-call budget T (Section 4).

Seedbank initialization (multi-model). We initialize the search by sampling a small seedbank with a seed model M_{seed} and inserting the compilation-feasible subset into the archive. When a pre-sampled seedbank is reused across methods, we still debit each reused seed as one generator call to preserve strict accounting. We decouple seeding and patching: M_{seed} is domain-finetuned for Lean 4 to increase the yield of compilable starting points, while a general-purpose patch model M_{patch} performs edit-conditioned proposals and bounded repairs (see Section 4). This decoupling reflects a practical capability mismatch: Lean-specialized

Algorithm 1 FormalEvolve (per problem; high-level pseudocode)

$t \leftarrow 0$; $\mathcal{A} \leftarrow \emptyset$
$\mathcal{A} \leftarrow$ GENERATEINITIALCANDIDATES$(x, M_{\text{seed}}, T, t)$ (compile-filtered; debits budget)
while $t < T$ **do**
  $I \leftarrow$ SAMPLEISLAND$(\mathcal{A})$
  $p \leftarrow$ SAMPLEPARENT$(\mathcal{A}_I)$
  $(\mathcal{I}_{\text{arch}}, \mathcal{I}_{\text{top}}) \leftarrow$ SAMPLECONTEXT$(\mathcal{A}_I, p)$
  $c \leftarrow$ PROPOSE$(p, \mathcal{I}_{\text{arch}}, \mathcal{I}_{\text{top}}, x; M_{\text{patch}})$
  $t \leftarrow t + 1$
  $\mathcal{C}, t \leftarrow$ EXPANDANDREPAIR(c, p, x, t, T)
  $\mathcal{A}_I, t \leftarrow$ EVALUATEANDUPDATE$(\mathcal{A}_I, \mathcal{C}, x, p, t, T)$
end while

models tend to yield more compilable seeds from scratch, while general instruction-tuned LLMs more reliably execute edit-conditioned patch/repair prompts but are less stable when generating Lean code from scratch.

Archive, islands, and duplicates. We maintain a per-problem archive $\mathcal{A}$ of unique compilation-feasible candidates that serves both as a parent pool and as prompt context. We use an island model with K semi-isolated subpopulations (default $K = 2$), where $K = 1$ recovers a single population, with periodic migration to mitigate stagnation. Detailed hyperparameters are provided in Appendix Section C.3. Each iteration samples an island and restricts parent selection and context sampling to the sampled island. To reduce collapse, we reject exact duplicates under canonicalization (whitespace/name normalization) and do not reinsert them into the archive.

Detailed pseudocode (including duplicate handling, repair semantics, and budget accounting) is provided in Section C.1.

Scoring and usage-penalized selection. We define a gated score that enforces feasibility before rewarding semantic consistency. In the current protocol, the score depends only on compilation and semantic consistency:

$$s(c) \;=\; C(c) \cdot \big(1 + J(c)\big), \tag{1}$$

so $s(c) \in \{0, 1, 2\}$ for infeasible / compilable / compilable+semantically consistent candidates. The archive retains multiple high-scoring candidates (rather than a single maximizer) to hedge against statement-level variability in downstream proof cost. We do not perform explicit online multi-objective selection (e.g., NSGA-II (Deb et al., 2002)); prover feedback is expensive and prover-dependent, so proof utility is evaluated only downstream after proving. Parent selection samples from the archive using a robust transformation of $s(c)$ and a parent-usage penalty to reduce collapse onto a few templates. Concretely, for each archived candidate c_i with score $\alpha_i = s(c_i)$ and parent-usage count n_i, we compute

$$\alpha_0 = \text{median}(\{\alpha_i\}), \qquad d = \max\big(\text{MAD}(\{\alpha_i\}), \varepsilon\big), \tag{2}$$

$$z_i = \frac{\alpha_i - \alpha_0}{d}, \qquad u_i = \frac{1}{1 + (1+\beta)\, n_i}, \tag{3}$$

$$w_i = \sigma(\lambda z_i) \cdot u_i, \tag{4}$$

and sample parents with probability $p_i = w_i / \sum_j w_j$. Here $\text{MAD}(A) = \text{median}_{a \in A} |a - \text{median}(A)|$ denotes the median absolute deviation and σ is the logistic sigmoid; λ and β are hyperparameters.

3.2. Proposal, Repair, and Diversification

Variation operators. Given a selected parent, the patch model proposes a new candidate using one of three prompt templates. *Full* patching rewrites the entire statement (and, when allowed, its surrounding imports) conditioned on the informal input and parent context. *Diff* patching performs localized edits that make minimal changes relative to the parent, guided by compiler/judge feedback. *Cross* patching additionally conditions on one or more *inspiration* candidates sampled from the current archive, and asks the model to combine useful elements (e.g., import context, binder structure, or type annotations) while attempting to preserve the parent problem's meaning (see Section C.5 for prompt details and Section C.7 for audited examples). All three templates return a complete Lean 4 file (imports plus a single statement) and are budgeted identically as generator calls.

Bounded compilation and semantic repair. We compile the full Lean 4 file and treat successful compilation as a hard feasibility gate. When compilation fails, FormalEvolve invokes a repair prompt that proposes a minimal patch conditioned on compiler feedback. Repair is bounded to a small, fixed number of attempts per proposal; each attempt is debited as a generator call. Bounding repair makes the search budget-auditable under hard cutoffs and prevents pathological cases where a single hard candidate consumes most of the call budget. For a compilable candidate that fails the semantic judge ($C(c) = 1$ and $J(c) = 0$), we optionally apply bounded *semantic repair* by prompting M_{patch} to revise the statement conditioned on the informal input and the judge's rationale (when available), and then re-evaluate; all repair and semantic-repair attempts are debited. We view compilation repair and semantic repair as two instantiations of the same edit-conditioned patch mechanism, differing only in the feedback source (compiler errors vs. judge rationale).

Call-free diversity fallback (EvolAST). Archive-based search can suffer from mode collapse where many candidates become near-duplicates or exploit superficial patterns. FormalEvolve mitigates this primarily via the usage penalty

in parent sampling. We additionally apply a conservative, no-call AST rewrite fallback inspired by EvolProver (Tian et al., 2025) when patching stalls (e.g., repeated duplicates or compile failures). EvolAST rewrites only within binder types and the goal type (imports/preamble unchanged) to produce symmetry/structure variants, which are then filtered by the same compilation gate and semantic judge; full details are in Section C.1.

4. Experiments

Benchmarks. We evaluate on ProofNet (Azerbayev et al., 2023) (Lean 4 port; test split, N=186) and CombiBench (Liu et al., 2025) (N=100, domain/style shift).

Environment. Candidates are compiled under a pinned Lean 4/Mathlib toolchain served by Kimina Lean Server (Dos Santos et al., 2025); full details are in Appendix Section C.3.

Budget. We use a generator-call budget $T = 100$ per problem, debiting each proposal or bounded repair LLM call as one call; repair/semantic-repair attempts are debited as separate calls.

Baselines. Our sampling baselines are budget-matched variants without an archive: **Sample** (no repair), **Compile Repair** (bounded compilation repair), and **Compile+Semantic Repair** (compilation repair plus one bounded semantic-repair call for compilable candidates failing the judge). To separate search effects from patch-model strength, Table 1 also reports a hybrid sampling baseline (Kimina generation with Qwen3 repair) that uses the same repair model as FormalEvolve but no archive-based search.

Models. In our main runs, the sampling baselines use Kimina-Autoformalizer-7B for sampling/repair, while FormalEvolve uses Kimina-Autoformalizer-7B for seeding and Qwen3-30B-A3B for patch/repair. For proof utility, we fix the prover to Goedel-Prover-V2-32B (Lin et al., 2025b) with a fixed prompt template (Appendix Section C.5.6).

Metrics. CH@T and SH@T denote the fraction of problems with at least one compilable candidate (resp. at least one compilable, judge-consistent candidate) within T debited generator calls. Let $s_j(T)$ be the number of deduplicated candidates with $C(c) = 1$ and $J(c) = 1$ for problem j within budget T; we report cross-problem concentration via the Gini coefficient

$$\mathrm{Gini}(T) = \frac{\sum_{i=1}^{N}\sum_{j=1}^{N}|s_i(T) - s_j(T)|}{2N\sum_{j=1}^{N} s_j(T) + \varepsilon}, \qquad (5)$$

and the Top-10% share $\sum_{j\in \mathrm{Top}_{10\%}} s_j(T) / \sum_{j=1}^{N} s_j(T)$ (where $\mathrm{Top}_{10\%}$ selects the problems with the largest $s_j(T)$). We evaluate downstream proof utility under a fixed RR64 prover protocol (pass@64 / complete@64 / theorem-complete@64; Table 2); full definitions are in Appendix Section A.1.

5. Results

Overview at fixed $T = 100$. Table 1 summarizes statement generation at $T = 100$ under strict generator-call accounting. It reports coverage (CH@100, SH@100) and cross-problem concentration (Gini and top-10% share over per-problem `semantic_ok` counts; lower is more uniform). FormalEvolve increases coverage and reduces concentration, with the largest gains on CombiBench. To separate archive-based search from patch/repair model strength, we include a hybrid sampling control (Kimina generation + Qwen3 repair) with the same repair model but no archive; the control raises SH, and FormalEvolve adds further gains.

Statement-level coverage, uniformity, and ablations. On CombiBench, SH@100 increases from 0.460 for the strongest Kimina-only baseline (Compile+Semantic Repair) to 0.530 for the hybrid control, and to 0.580 for FormalEvolve (Table 1); on ProofNet, 0.780 $\rightarrow$ 0.828 $\rightarrow$ 0.849. Most of the improvement comes from the stronger repair model, and archive-based population search adds a further gain. FormalEvolve also reduces concentration relative to both Kimina baselines and the hybrid control (lower Gini and top-10% share), indicating gains beyond a small subset of easy problems (Table 1; Figure 4 and Appendix Figure 9). In ablations, bounded patch-repair calls account for most of the semantic-coverage gain: removing patch repair drops SH@100 on CombiBench from 0.580 to 0.470 and on ProofNet from 0.849 to 0.780 (Table 1). EvolAST and island mechanics have smaller effects and can be non-monotonic under fixed budgets (Table 1; Appendix C.9). Appendix Tables 4 and 5 provide a finer decomposition at $T = 100$ (FY/SD/SY); Figure 3 and Appendix Figure 7 show the budget-sweep curves; Appendix B.2.1 lists representative early-hit instances.

Gains beyond easy instances: cross-problem uniformity. We quantify whether semantic successes concentrate on a small subset of easy problems via coverage (SH@100) and concentration (Gini and top-10% share). At $T = 100$, FormalEvolve increases coverage while reducing concentration on both benchmarks (Table 1). Figure 4 visualizes per-problem semantic-success counts, and Appendix Figure 9 shows concentration versus the call budget. Across budgets, the sampling baselines have comparatively flat concentration curves, while FormalEvolve reduces concentration as T increases, yielding progressively more uniform coverage.

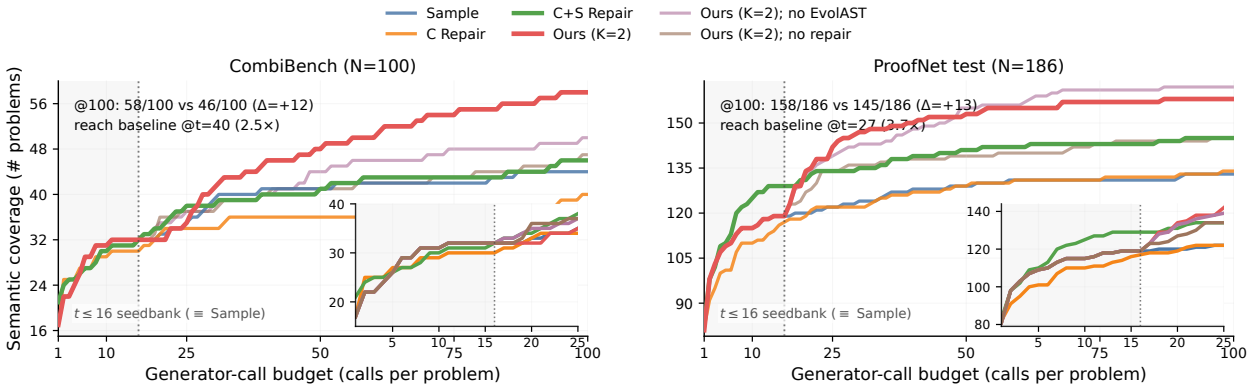
Table 1. Statement-generation summary at $T = 100$ generator calls per problem. We report CH@100, SH@100, and cross-problem concentration of semantic-success counts (Gini and top-10% share over per-problem `semantic_ok` counts). Kimina baselines use Kimina-7B for generation and repair; Qwen3 baselines use Qwen3-30B-A3B. The hybrid control swaps only the repair model to Qwen3 (Kimina generation + Qwen3 repair; no archive). FormalEvolve adds archive-based search on top of the same Qwen3 patch/repair model. Bold indicates the best value among FormalEvolve variants.

	ProofNet (test, $N = 186$)				CombiBench ($N = 100$)			
Method	CH@100	SH@100	Gini↓	Top-10%↓	CH@100	SH@100	Gini↓	Top-10%↓
Baselines (Kimina; no archive)								
Sample	0.903	0.715	0.537	0.243	1.000	0.440	0.816	0.637
Compile Repair	0.909	0.720	0.566	0.263	0.990	0.400	0.825	0.641
Compile+Semantic Repair	0.909	0.780	0.555	0.264	0.990	0.460	0.813	0.609
Baselines (Qwen3; no archive)								
Qwen3 Sample	0.387	0.242	0.896	0.876	0.300	0.200	0.939	0.952
Qwen3 + Compile Repair	0.371	0.237	0.912	0.911	0.900	0.260	0.740	0.385
Qwen3 + Compile+Semantic Repair	0.688	0.548	0.802	0.656	0.940	0.390	0.827	0.655
Hybrid control (no archive)								
Hybrid control	0.973	0.828	0.505	0.241	1.000	0.530	0.790	0.588
FormalEvolve (ours)								
FormalEvolve (K=2)	0.973	0.849	0.443	0.229	1.000	**0.580**	0.759	0.531
FormalEvolve (K=1)	**0.984**	0.866	**0.362**	**0.198**	1.000	0.550	**0.726**	**0.442**
FormalEvolve (K=2; w/o EvolAST)	0.973	**0.871**	0.454	0.236	1.000	0.500	0.776	0.577
FormalEvolve (K=2; w/o patch repair)	0.903	0.780	0.449	0.211	1.000	0.470	0.772	0.494

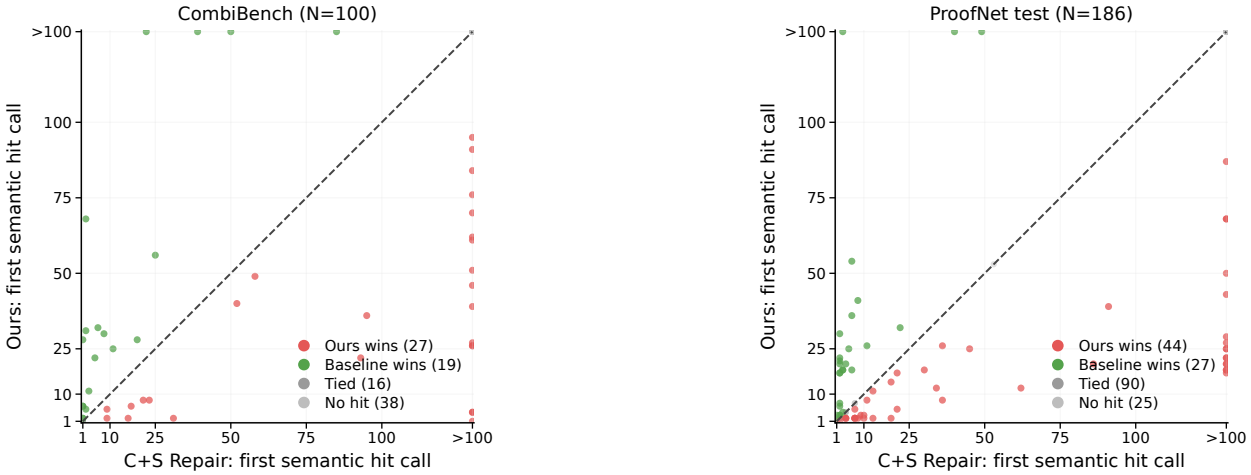
Does the uniformity effect persist with strong seeds? (ReForm-as-seeder). We also ask whether FormalEvolve remains helpful when initialization is already strong. On CombiBench, we treat REFORM (Chen et al., 2025) as a seeder and run a plug-in handover under a tight call budget of 16. We report cross-problem concentration (Gini / top-10% share) alongside $\text{SemOK}_{\text{total}}$ (total judge-consistent statements aggregated across problems) to separate uniformity from total yield (Appendix B.2.2).

Proof utility evaluation. We evaluate downstream proving on the semantically consistent repertoires produced within $T = 100$ generator calls using a fixed prover (Goedel-Prover-V2-32B) and a fixed prompt (Appendix C.5.6). The prover sees only candidates with `compile_ok`$= 1$ and `semantic_ok`$= 1$, deduplicated under canonicalization and capped at 64 statements per problem; problems with an empty repertoire receive zero prover attempts and count as failures (x/N). As an oracle control, we run the same prover on the dataset-provided ground-truth formal statement for each problem, bypassing both the generator and the semantic judge. We report pass@64, complete@64, and theorem-complete@64 in Table 2. The hybrid sampling control (Kimina gen + Qwen3 repair) is used only as a statement-stage control in Table 1 and is not evaluated at the prover stage in our current runs. Figure 5 plots pass@k and complete@k versus the prover attempt budget k, together with the gap relative to Kimina Compile+Semantic Repair.

Interpreting proof utility gains. Table 2 reports solved-problem counts under a fixed prover budget ($B = 64$). The oracle ground-truth-statement control does not necessarily dominate (theorem-complete@64 18/100 on CombiBench; 34/186 on ProofNet), since proof utility measures prover-friendliness under a fixed prover rather than a faithfulness ceiling. On ProofNet, the oracle theorem-complete count is lower than all generator-based methods, suggesting that a diverse semantically consistent repertoire can surface formulations that the prover handles better than the canonical statement. On CombiBench, FormalEvolve improves theorem-complete@64 (13/100) compared to Sample (8/100) and Kimina Compile+Semantic Repair (8/100). This improvement is largely driven by statement coverage: within $T = 100$ calls, FormalEvolve produces a non-empty semantically consistent repertoire on 58/100 problems versus 46/100 for Kimina Compile+Semantic Repair (Table 1). On ProofNet, proof utility is comparable to Kimina Compile+Semantic Repair (pass@64 127/186 vs 119/186; complete@64 52/186 vs 50/186; theorem-complete@64 45/186 vs 46/186), consistent with ProofNet potentially being closer to the training distribution of competition-style autoformalizers (Wang et al., 2025a). In this regime, improvements can be modest and non-monotonic under a fixed prover and attempt budget because semantically consistent statements can still differ in proof-search friendliness. Figure 5 shows how proof utility varies with the prover budget k. To make the coverage vs. prover-sensitivity distinction explicit, Appendix Figure 6 decomposes theorem-complete@64 into (i) whether a method produces any statement to attempt (non-empty repertoire at the prover stage) and (ii) theorem-complete success conditional on being attempted. On CombiBench, FormalEvolve attempts 57/100 problems and achieves 13/100 theorem-complete solves, compared to 44/100 attempted and 8/100 solves for Sample and 46/100 attempted and 8/100 solves for Compile+Semantic Repair; the oracle attempts 100/100 and achieves 18/100 theorem-



(a) Semantic coverage vs debited calls t. The dotted line marks the nominal seedbank boundary (t=16).



(b) First semantic-hit call: FormalEvolve vs Kimina Compile+Semantic Repair ("> 100": no hit within budget).

Figure 3. Coverage and first-hit timing across two benchmarks ($t \leq 100$).

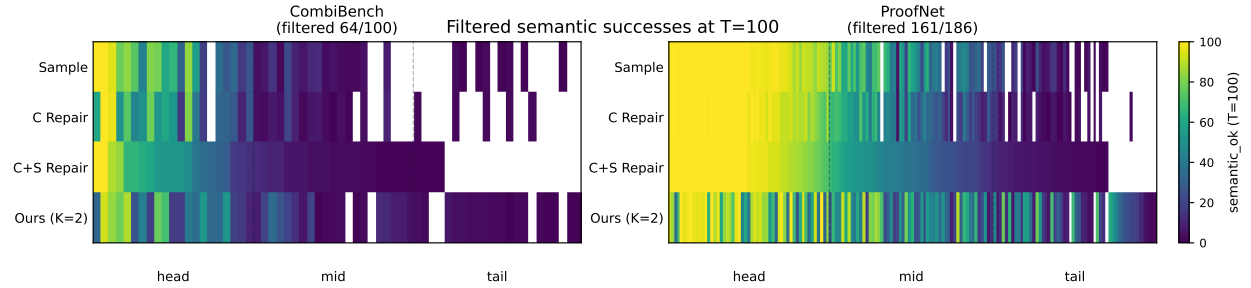


Figure 4. Filtered per-problem semantic-success counts at $T = 100$ under strict budget accounting (`semantic_ok`; 0 shown as white). We keep problems where at least one method has a positive `semantic_ok` count and order columns by Compile+Semantic Repair to keep the ordering independent of ours.

complete. Conditional on being attempted, success is 13/57 (22.8%) for ours versus 8/44 (18.2%) for Sample, 8/46 (17.4%) for Compile+Semantic Repair, and 18/100 (18.0%) for the oracle. On ProofNet, generator-based methods cover most problems (133–158 attempted out of 186), yet theorem-complete remains sensitive: ours achieves 45/186 (45/158 attempted), Compile+Semantic Repair achieves 46/186 (46/145 attempted), Sample achieves 41/186 (41/133 attempted), while the oracle achieves 34/186 (34/186 attempted).

6. Limitations

Our framework and evaluation rely on imperfect proxies. Semantic consistency is assessed via an LLM-based judge (CriticLean-Qwen3-14B), and judgments remain sensitive to model versions and prompting details. We therefore report judge configurations and analyze judge/prover mismatch cases; under our deployed setup, the judge reaches 79.0% and 81.0% accuracy on CONSISTENCYCHECK and CRITICLEANBENCH, respectively (Appendix C.5.7).

Table 2. Downstream proof utility at $B = 64$ prover attempts per problem on the semantically consistent repertoire produced within $T = 100$ generator calls. We report solved-problem counts (x/N) for pass@64, complete@64, and theorem-complete@64 (complete proof of an explicit `theorem`/`lemma`). For generator-based methods, the prover sees only candidates with `compile_ok`$= 1$ and `semantic_ok`$= 1$ (after deduplication; capped to 64 per problem); problems with an empty repertoire receive zero prover attempts and count as failures. The oracle row (Ground truth statement) runs the same prover on dataset-provided ground-truth formal statements (one per problem), bypassing the generator and semantic judge.

Method	pass@64	complete@64	theorem-complete@64	pass@64	complete@64	theorem-complete@64
	CombiBench ($N = 100$)			ProofNet (test, $N = 186$)		
Ours (K=2)	44/100	27/100	13/100	127/186	52/186	45/186
Sample	41/100	23/100	8/100	106/186	46/186	41/186
Compile+Semantic Repair (Kimina)	40/100	23/100	8/100	119/186	50/186	46/186
Ground truth statement (oracle)	96/100	68/100	18/100	146/186	46/186	34/186

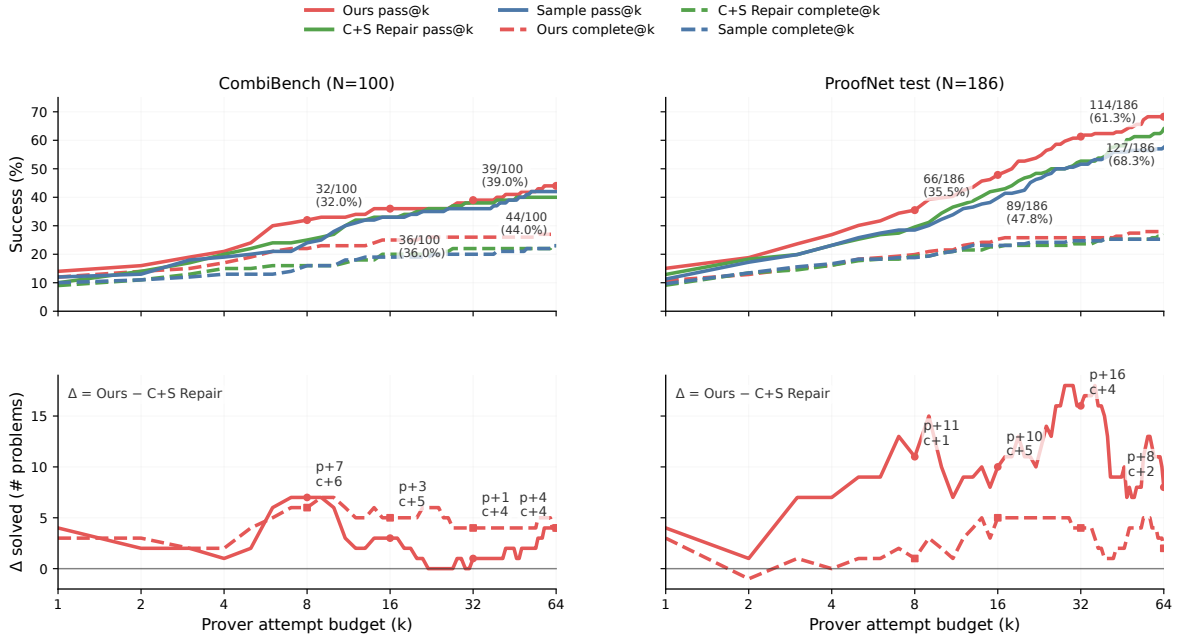


Figure 5. Proof utility as a function of prover attempts k on the semantically consistent repertoire produced within $T = 100$ calls. Solid: pass@k; dashed: complete@k; bottom row shows the gap relative to Kimina Compile+Semantic Repair.

Proof utility depends on the chosen prover, proof budget, and prompting details; pass@64, complete@64, and theorem-complete@64 measure utility under a fixed prover configuration rather than an absolute ceiling on provability. More broadly, semantic consistency is still approximated by an LLM judge, which can yield judge/prover mismatches and makes end-to-end faithfulness hard to certify. When references are available, prover-native equivalence checks (e.g., BEq/BEq+ (Poiroux et al., 2024)) may reduce this noise and better align statement search with prover utility.

Finally, potential benchmark overlap in the training data of strong domain-finetuned generators may reduce observed gains on in-distribution settings; we include CombiBench to probe robustness under domain/style shift.

7. Conclusion

We introduced FormalEvolve, a compile-gated population search method for Lean 4 autoformalization under a fixed generator-call budget. Under fixed budgets, FormalEvolve increases semantic coverage and downstream proof utility by constructing a semantically consistent repertoire rather than committing to a single output. Future work includes reducing judge noise with prover-native equivalence checks (e.g., BEq/BEq+ (Poiroux et al., 2024)) and transferring the same compile-gated population-search recipe, with minimal adaptation, to Coq, Isabelle, and related systems.

Impact Statement

This paper studies LLM-based autoformalization and downstream proof search for machine-checkable mathematics. Potential benefits include improved formal-mathematics tooling, educational support, and stronger verification workflows. Potential risks arise if automatically generated formal statements or successful proofs are over-interpreted as guarantees of faithfulness to the original informal claim. We therefore emphasize fixed-budget evaluation, explicit lim-

itations of semantic judging, and the continued need for human verification in correctness-critical settings.

References

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

Chen, G., Wu, J., Chen, X., Zhao, W. X., Song, R., Li, C., Fan, K., Liu, D., and Liao, M. ReForm: Reflective autoformalization with prospective bounded sequence optimization, 2025. URL https://arxiv.org/abs/2510.24592. arXiv preprint.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6 (2):182–197, 2002. doi: 10.1109/4235.996017. URL https://doi.org/10.1109/4235.996017.

Dos Santos, M., de Saxcé, H., Wang, H., Wang, R., Baksys, M., Unsal, M., Liu, J., Liu, Z., and Li, J. Kimina lean server: A high-performance lean server for large-scale verification, 2025. URL https://arxiv.org/abs/2504.21230.

Guo, Q., Wang, J., Zhang, J., Kong, D., Huang, X., Xi, X., Wang, W., Wang, J., Cai, X., Zhang, S., and Ye, W. Autoformalizer with tool feedback, 2025. URL https://arxiv.org/abs/2510.06857.

Han, J. M., Rute, J., Wu, Y., Ayers, E. W., and Polu, S. Proof artifact co-training for theorem proving with language models. In *The Tenth International Conference on Learning Representations (ICLR 2022)*. OpenReview.net, 2022. URL https://openreview.net/forum?id=rpxJc9j04U.

Lange, R. T., Imajuku, Y., and Cetin, E. ShinkaEvolve: Towards open-ended and sample-efficient program evolution, 2025. URL https://arxiv.org/abs/2509.19349. arXiv preprint.

Lehman, J. and Stanley, K. O. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, 2011. doi: 10.1162/EVCO_A_00025. URL https://doi.org/10.1162/EVCO_A_00025.

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

Lin, Y., Tang, S., Lyu, B., Yang, Z., Chung, J.-H., Zhao, H., Jiang, L., Geng, Y., Ge, J., Sun, J., Wu, J., Gesi, J., Lu, X., Acuna, D., Yang, K., Lin, H., Choi, Y., Chen, D., Arora, S., and Jin, C. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction, 2025b. URL https://arxiv.org/abs/2508.03613.

Liu, F., Zhang, R., Xie, Z., Sun, R., Li, K., Lin, X., Wang, Z., Lu, Z., and Zhang, Q. LLM4AD: A platform for algorithm design with large language model, 2024. URL https://arxiv.org/abs/2412.17287.

Liu, J., Lin, X., Bayer, J., Dillies, Y., Jiang, W., Liang, X., Soletskyi, R., Wang, H., Xie, Y., Xiong, B., Yang, Z., Zhang, J., Zhi, L., Li, J., and Liu, Z. CombiBench: Benchmarking LLM capability for combinatorial mathematics, 2025. URL https://arxiv.org/abs/2505.03171.

Lu, J., Wan, Y., Huang, Y., Xiong, J., Liu, Z., and Guo, Z. FormalAlign: Automated alignment evaluation for autoformalization, 2024. URL https://arxiv.org/abs/2410.10135.

Moura, L. d. and Ullrich, S. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37. URL https://doi.org/10.1007/978-3-030-79876-5_37.

Mouret, J.-B. and Clune, J. Illuminating search spaces by mapping elites, 2015. URL https://arxiv.org/abs/1504.04909.

Novikov, A., Vu, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., and Balog, M. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Peng, Z., Yao, Y., Ma, K., Guo, S., Li, Y., Zhang, Y., Zhang, C., Zhang, Y., Yu, Z., Li, L., Liu, M., Xia, Y., Shen, J., Wu, Y., Cao, Y., Zhang, Z., Huang, W., Liu, J., and Zhang, G. CriticLean: Critic-guided reinforcement learning for mathematical formalization, 2025. URL https://arxiv.org/abs/2507.06181.

Poiroux, A., Weiss, G., Kunčak, V., and Bosselut, A. Reliable evaluation and benchmarks for statement autoformalization, 2024. URL https://arxiv.org/abs/2406.07222.

Polu, S. and Sutskever, I. Generative language modeling for automated theorem proving, 2020. URL https://arxiv.org/abs/2009.03393.

Song, P., Yang, K., and Anandkumar, A. Lean copilot: Large language models as copilots for theorem proving in lean, 2024. URL https://arxiv.org/abs/2404.12534.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 367–381. ACM, 2020. doi: 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824.

Tian, Y., Huang, R., Wang, X., Ma, J., Huang, Z., Luo, Z., Lin, H., Zheng, D., and Du, L. EvolProver: Advancing automated theorem proving by evolving formalized problems via symmetry and difficulty, 2025. URL https://arxiv.org/abs/2510.00732.

Wang, H., Unsal, M., Lin, X., Baksys, M., Liu, J., Dos Santos, M., Sung, F., Vinyes, M., Ying, Z., Zhu, Z., Lu, J., de Saxcé, H., Bailey, B., Song, C., Xiao, C., Zhang, D., Zhang, E., Pu, F., Zhu, H., Liu, J., Bayer, J., Michel, J., Yu, L., Dreyfus-Schmidt, L., Tunstall, L., Pagani, L., Machado, M., Bourigault, P., Wang, R., Polu, S., Barroyer, T., Li, W.-D., Niu, Y., Fleureau, Y., Hu, Y., Yu, Z., Wang, Z., Yang, Z., Liu, Z., and Li, J. Kimina-Prover Preview: Towards large formal reasoning models with reinforcement learning, 2025a. URL https://arxiv.org/abs/2504.11354.

Wang, H., Xie, R., Wang, Y., Gao, G., Yu, X., and Dong, B. Aria: An agent for retrieval and iterative auto-formalization via dependency graph, 2025b. URL https://arxiv.org/abs/2510.04520.

Wu, Y., Jiang, A. Q., Li, W., Rabe, M. N., Staats, C., Jamnik, M., and Szegedy, C. Autoformalization with large language models. In *Advances in Neural Information Processing Systems*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/d0c6bc641a56bebee9d985b937307367-Abstract-Conference.html.

Yang, K., Swope, A. M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/4441469427094f8873d0fecb0c4e1cee-Abstract-Datasets_and_Benchmarks.html.

Zhao, H., Geng, Y., Tang, S., Lin, Y., Lyu, B., Lin, H., Jin, C., and Arora, S. Ineq-Comp: Benchmarking human-intuitive compositional reasoning in automated theorem proving on inequalities, 2025. URL https://arxiv.org/abs/2505.12680.

Zheng, K., Han, J. M., and Polu, S. miniF2F: a cross-system benchmark for formal olympiad-level mathematics. In *The Tenth International Conference on Learning Representations (ICLR 2022)*. OpenReview.net, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

Appendix organization. We organize the appendix into three parts. Appendix A starts with a quick reference for metric definitions (Table 3) and then reports additional quantitative results (tables and coverage curves versus debited calls). Appendix B provides diagnostic analyses of cross-problem concentration and component effects under strict budgets. Appendix C collects supplementary material (pseudocode, configuration/reproducibility, prompt templates, and auditing evidence, including qualitative case studies in Appendix C.7).

Reproducibility pointers. For configuration, budget accounting, prompts, and auditing artifacts, see Appendix C, especially Appendix C.3 and Appendix C.2.

A. Additional Results

Reading guide. This section reports supplementary tables/figures that support the main fixed-budget comparisons. Table 3 is a quick reference for all metrics under strict generator-call accounting; the remaining subsections report decompositions at T=100 and budget-sweep curves.

A.1. Metric definitions

We provide definitions of all reporting metrics under strict generator-call accounting. For a given problem, let E_t denote the set of candidates evaluated up to debited call t, and let $E_t^{\text{feas}} = \{c \in E_t : C(c) = 1\}$. Let $G_t = \{c \in E_t : C(c) = 1 \wedge J(c) = 1\}$, and for dataset-level concentration let $s_j(t)$ be the number of deduplicated semantic successes for problem j at budget t. For proof utility, let $S_t^{(64)}$ denote the first 64 statements in $\tilde{G}_t = \text{Dedup}(G_t)$ used by the fixed prover protocol. Let $P_{\text{pass}}(c) \in \{0, 1\}$ indicate whether the prover returns a Lean 4-accepted proof script for c (warnings allowed; `sorry` permitted), $P_{\text{complete}}(c) \in \{0, 1\}$ indicate a complete proof without `sorry`, and $P_{\text{theorem}}(c) \in \{0, 1\}$ indicate a complete proof of an explicit `theorem`/`lemma`.

Table 3. Metric definitions under strict generator-call accounting.

Metric	Definition (strict accounting)	Interpretation
$\text{FY}(t)$	$\frac{1}{\lvert E_t\rvert}\sum_{c\in E_t} C(c)$	Feasible yield: fraction of calls that compile.
$\text{CH}(t)$	$I[\exists c \in E_t : C(c) = 1]$	Compile hit: at least one compilable candidate.
$\text{SH}(t)$	$I[\exists c \in E_t : C(c) = 1 \wedge J(c) = 1]$	Semantic hit: at least one compilable & judge-consistent candidate.
$\text{SD}(t)$	$\frac{1}{\lvert E_t^{\text{feas}}\rvert}\sum_{c\in E_t^{\text{feas}}} J(c)$ (0 if $\lvert E_t^{\text{feas}}\rvert = 0$)	Semantic density among compilable candidates.
$\text{SY}(t)$	$\frac{1}{\lvert E_t\rvert}\sum_{c\in E_t} J(c)$	Semantic yield per call (treat non-compilable as $J = 0$).
$\text{Div}(t)$	$\lvert\text{Dedup}(G_t)\rvert$	Diversity: deduplicated semantic successes.
$\text{Cov}(t)$	$\frac{1}{N}\sum_{j=1}^{N} I[s_j(t) \geq 1]$ (dataset-level)	Coverage: problems with ≥ 1 semantic success.
$\text{Gini}(t)$	$\frac{\sum_{i,j}\lvert s_i(t)-s_j(t)\rvert}{2N\sum_j s_j(t)+\varepsilon}$ (dataset-level)	Concentration: inequality of success counts (↓).
Top-10% share	$\frac{\sum_{j\in\text{Top}_{10\%}} s_j(t)}{\sum_j s_j(t)}$ (dataset-level)	Top-10% share: share of semantic-success mass on the easiest problems.
$\text{PU}_{\text{pass}}@64(t)$	$I[\exists c \in S_t^{(64)} : P_{\text{pass}}(c) = 1]$	Proof Utility (pass): any proof accepted by Lean 4 under fixed prover budget.
$\text{PU}_{\text{complete}}@64(t)$	$I[\exists c \in S_t^{(64)} : P_{\text{complete}}(c) = 1]$	Proof Utility (complete): at least one sorry-free proof.
$\text{PU}_{\text{theorem}}@64(t)$	$I[\exists c \in S_t^{(64)} : P_{\text{theorem}}(c) = 1]$	Proof Utility (theorem-complete): sorry-free proof of an explicit theorem/lemma.

Notes. Each debited generator call yields one evaluated representative candidate; repair/semantic-repair count as additional calls. Dedup uses conservative whitespace/name canonicalization. Proof utility uses $\tilde{G}_t = \text{Dedup}(G_t)$, takes the first 64 statements by call index, and allocates $B = 64$ prover attempts in a round-robin schedule.

A.2. Proof utility decomposition

Figure 6 decomposes theorem-complete@64 into (i) whether a method produces a non-empty statement repertoire to attempt under the fixed prover protocol, and (ii) whether at least one attempted statement is solved at theorem-complete level under the same attempt budget.

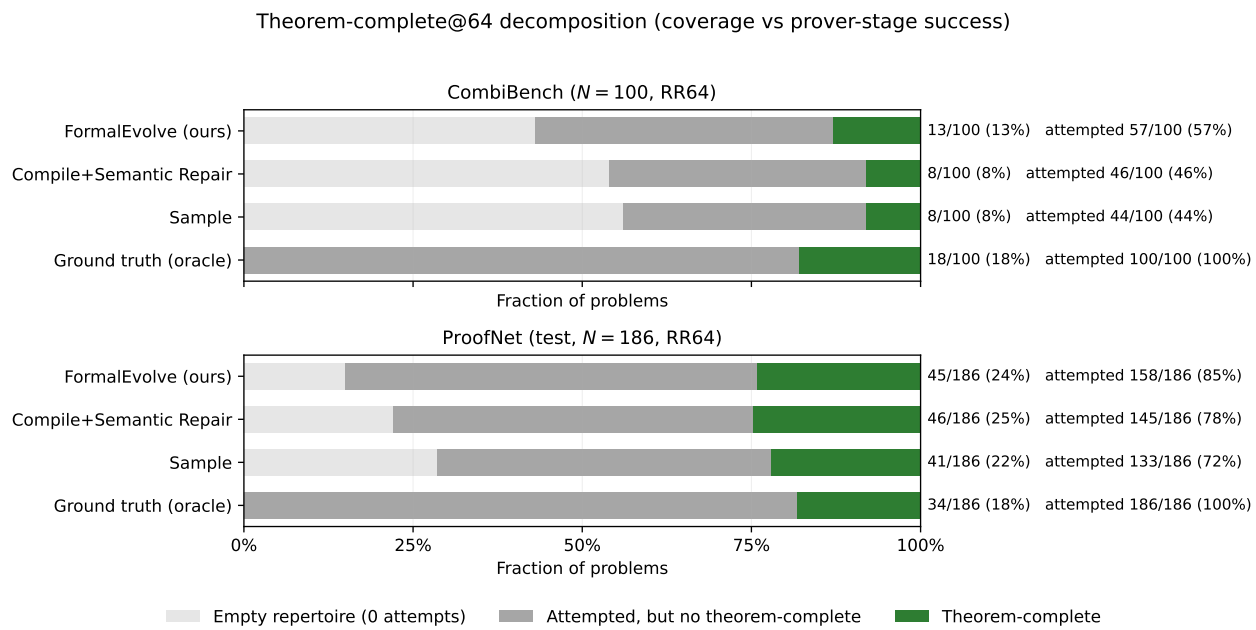


Figure 6. Decomposing theorem-complete@64 under a fixed prover and attempt budget (B=64). Each bar splits the benchmark denominator into problems with an empty statement repertoire for proving (zero prover attempts), problems with a non-empty repertoire but no theorem-complete success, and problems with at least one theorem-complete success.

How to interpret the decompositions. In Tables 4 and 5, FY is compilation yield, SD is semantic density conditional on compilation, and SY is semantic yield per generator call (treating compilation failures as semantic failures). Reporting FY/SH/SD/SY together separates gains from (i) producing more compilable candidates, (ii) improving semantic consistency among compilable candidates, and (iii) reallocating budget between proposal and repair.

A.3. Statement-level decompositions at $T = 100$

Table 4. Summary on CombiBench at a fixed generator-call budget of $T = 100$ (FY: feasible yield, SH: semantic hit rate, SD: semantic density among feasible candidates, SY: semantic yield per call). Downstream proof utility is reported separately in Table 2.

Method	FY	SH	SD	SY
Sample	0.695	0.440	0.189	0.132
Compile Repair	0.555	0.400	0.199	0.110
Compile+Semantic Repair	0.324	0.460	0.371	0.120
FormalEvolve (K=2)	0.407	0.580	0.326	0.133
FormalEvolve (w/o EvolAST fallback)	0.346	0.500	0.313	0.108
FormalEvolve (w/o patch repair)	0.682	0.470	0.250	0.170

Table 5. Summary on ProofNet (HF test) at a fixed generator-call budget of $T = 100$ (FY: feasible yield, SH: semantic hit rate, SD: semantic density among feasible candidates, SY: semantic yield per call). Downstream proof utility is reported separately in Table 2.

Method	FY	SH	SD	SY
Sample	0.702	0.715	0.599	0.421
Compile Repair	0.625	0.720	0.622	0.389
Compile+Semantic Repair	0.509	0.780	0.760	0.387
FormalEvolve (K=2)	0.562	0.849	0.775	0.435
FormalEvolve (w/o EvolAST fallback)	0.539	0.871	0.788	0.425
FormalEvolve (w/o patch repair)	0.694	0.780	0.684	0.475

A.4. Coverage gaps across the budget

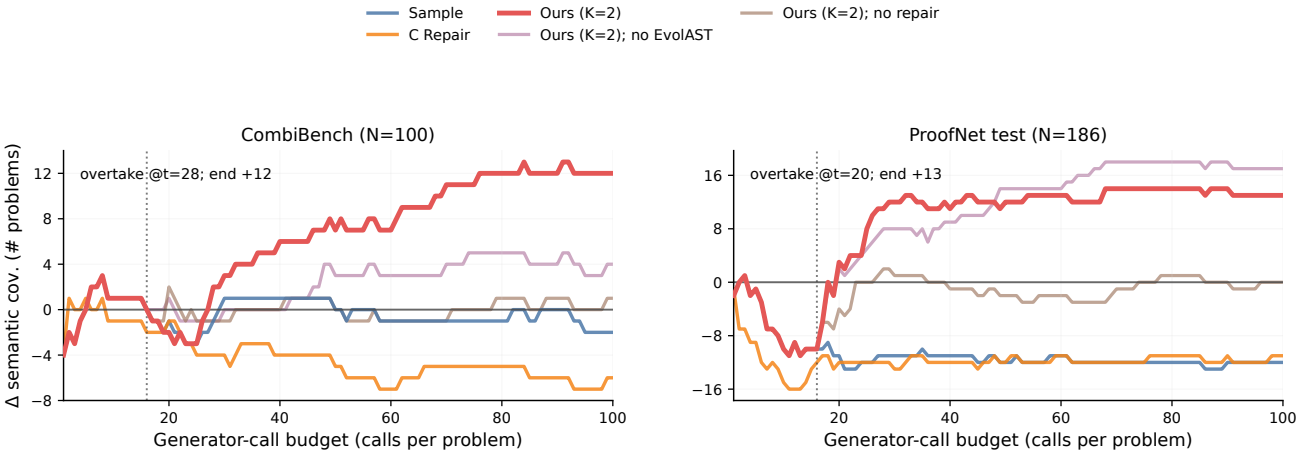


Figure 7. Coverage gap vs debited calls, plotted as Δ semantic coverage relative to the strongest Kimina sampling baseline (Compile+Semantic Repair). The zero line corresponds to the baseline; positive values indicate a method covers more problems at the same generator-call budget. The dotted vertical line marks the nominal seedbank boundary (t=16) used by evolution-based methods (pre-sampled seeds, debited calls); a small subset of problems can require additional debited seeds before evolution starts.

Hybrid baseline: Kimina generation with Qwen3 repair. We include this baseline to separate search effects from patch-model strength; its results are reported in Table 1 under the hybrid baseline group, using the same generator-call accounting and repair protocol as the sampling baselines.

B. Analysis

This section provides diagnostic analyses that help interpret the main fixed-budget results under strict accounting. We report lightweight supplementary statistics (paired uncertainty, call composition, and prover-stage non-dominance), assess cross-problem concentration, and summarize which components drive gains (bounded repair versus online diversity), together with representative failure modes.

B.1. Lightweight supplementary analyses

We report additional lightweight analyses computed from the same strict-budget artifacts used for the main figures and tables (no additional training or new model runs). These diagnostics are intended to help interpret the fixed-budget results: (i) paired uncertainty for SH@100 and theorem-complete@64 (Table 6); (ii) per-problem repair-call composition under strict accounting (Table 7); and (iii) prover-stage non-dominance quadrant counts at theorem-complete@64 (Table 8).

Table 6. Paired uncertainty over problems. We report paired bootstrap 95% CI for the mean difference (FormalEvolve − Compile+Semantic Repair) and an exact sign test on wins/losses (ties excluded).

Dataset	Metric	mean(FE)	mean(C+S)	Δ [95% CI]	wins/loss/tie
CombiBench	SH@100	0.580	0.460	+0.120 [0.040, 0.200]	16/4/80
ProofNet	SH@100	0.849	0.780	+0.070 [0.027, 0.118]	16/3/167
CombiBench	theorem-complete@64	0.130	0.080	+0.050 [-0.010, 0.110]	7/2/91
ProofNet	theorem-complete@64	0.242	0.247	-0.005 [-0.054, 0.043]	11/12/163

Table 7. Debited-call composition under the strict $T = 100$ budget. We report median [q25,q75] of compilation-repair calls (CRep) and semantic-repair calls (SRep) across problems.

Dataset	Method	CRep (median [q25,q75])	SRep (median [q25,q75])
CombiBench	Sample	0.0 [0.0, 0.0]	0.0 [0.0, 0.0]
CombiBench	Compile Repair	29.0 [4.0, 52.0]	0.0 [0.0, 0.0]
CombiBench	Compile+Semantic Repair	14.0 [4.0, 34.5]	38.0 [26.0, 54.0]
CombiBench	FormalEvolve (K=2)	30.5 [20.8, 39.0]	20.5 [14.0, 26.0]
ProofNet	Sample	0.0 [0.0, 0.0]	0.0 [0.0, 0.0]
ProofNet	Compile Repair	16.0 [0.0, 53.0]	0.0 [0.0, 0.0]
ProofNet	Compile+Semantic Repair	9.0 [0.0, 39.8]	11.5 [0.0, 37.8]
ProofNet	FormalEvolve (K=2)	24.0 [8.0, 40.8]	4.0 [0.0, 16.0]

Table 8. Non-dominance at the prover stage (theorem-complete@64): quadrant counts for FormalEvolve vs Compile+Semantic Repair.

Dataset	n	both fail	FE only	baseline only	both succeed
CombiBench	100	85	7	2	6
ProofNet	186	129	11	12	34

B.2. Cross-problem uniformity: coverage and concentration

We assess whether semantic successes concentrate on a small subset of easy problems under a fixed call budget. At $T = 100$, we summarize the per-problem `semantic_ok` count distribution using coverage (SH@100) and concentration (Gini and top-10% share; Table 1). Figure 4 provides a compact per-problem view. For readability, we filter to problems where at least one method attains a positive `semantic_ok` count (otherwise the column is identically zero). We order columns by the strongest Kimina sampling baseline (Compile+Semantic Repair) using its per-problem `semantic_ok` counts at $T = 100$, defining an x-axis independent of ours; Figure 8 uses the same ordering for the delta visualization.

Aggregate pattern under baseline ordering. Under this baseline ordering, gains concentrate in the baseline-ranked tail: on the filtered set, $\Delta > 0$ dominates in the tail segment, while the head segment can favor the baseline (especially on ProofNet). We define head/mid/tail by splitting the baseline-ordered problems into three contiguous segments as evenly as possible within each benchmark. The table below counts how often $\Delta = \#\texttt{semantic\_ok}(\textsc{FormalEvolve}) - \#\texttt{semantic\_ok}(\textsc{Compile+Semantic Repair})$ is positive/negative/zero within each segment (reported as $+/-/0$).

Dataset	head $(+/-/0)$	mid $(+/-/0)$	tail $(+/-/0)$
CombiBench (filtered 64/100)	8/12/1	12/7/2	18/2/2
ProofNet (filtered 161/186)	6/41/6	35/19/0	49/5/0

B.2.1. Representative early semantic hits and baseline-miss instances

On each benchmark, there are 16 instances where Compile+Semantic Repair has zero `semantic_ok` at T=100 while FormalEvolve attains a positive `semantic_ok` count. Conversely, there are 4 such instances where the baseline hits but FormalEvolve misses on CombiBench, and 3 on ProofNet. To ground the aggregate patterns above and illustrate non-dominance, Table 9 lists representative instances in both directions: where FormalEvolve reaches a semantically consistent compilable statement substantially earlier than the baseline (or the baseline has no hit within $T = 100$ calls), and where the baseline hits but FormalEvolve misses within the same call budget. These examples are computed from the debited-call logs and correspond to the below-diagonal and right-edge regions in Figure 3(b).

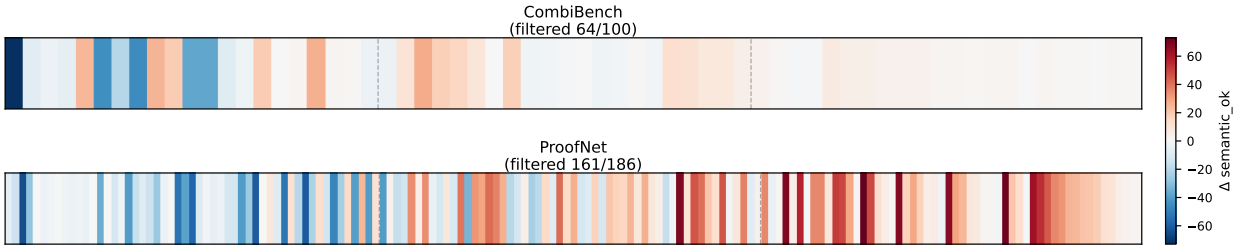


Figure 8. Delta strips at $T = 100$ under strict budget accounting on the same filtered set and in the same baseline ordering (sorted by Compile+Semantic Repair `semantic_ok`) as Figure 4, showing $\Delta = \#\texttt{semantic\_ok}(\text{FORMALEVOLVE}) - \#\texttt{semantic\_ok}(\text{COMPILE+SEMANTIC REPAIR})$ per problem.

Table 9. Representative early semantic hits at a generator-call budget of $T = 100$. We include both directions (FormalEvolve earlier / baseline no-hit, and baseline-hit / FormalEvolve no-hit). "> 100" indicates no semantic hit within the call budget.

Benchmark	Problem id	Short description	Ours first hit	C+S Repair first hit
FormalEvolve hits earlier, or baseline has no hit within $T = 100$				
CombiBench	`0055_egmo_2022_p5`	Domino tilings: parity of $f(n, 2k)$ for all k	1	> 100
CombiBench	`0063_usamo_2000_p4`	3 colored squares form an axis-aligned right triangle	4	> 100
CombiBench	`0077_imo_2010_p5`	Box/coin operations reach $2010^{2010^{2010}}$?	4	> 100
CombiBench	`0035_brualdi_ch9_8`	Count SDRs for six 2-sets arranged in a cycle	21	> 100
CombiBench	`0011_brualdi_ch1_10`	No magic square of order 2	22	93
CombiBench	`0047_brualdi_ch13_10`	Tournament: a vertex reaches all others within 2 steps	27	95
CombiBench	`0057_imos1_2015_c6`	Infinitely many integers without a unique odd-sum representation	2	31
CombiBench	`0065_imo_2020_p3`	Split weighted colored pebbles into two equal piles	27	52
ProofNet	`0046_exercise_3_4_5b`	Quotients of solvable groups are solvable	19	> 100
ProofNet	`0124_exercise_26_12`	Perfect map: compact Y implies compact X	21	> 100
ProofNet	`0163_exercise_3_13`	Cauchy product of absolutely convergent series	25	> 100
ProofNet	`0119_exercise_23_3`	Connected union with shared intersection	18	86
ProofNet	`0095_exercise_5_6_14`	Distinct roots of $x^m - x$ in characteristic p	24	91
ProofNet	`0045_exercise_3_4_4`	Finite abelian groups have subgroups of each divisor order	12	62
ProofNet	`0005_exercise_6_4_12`	No simple group of order 224	6	45
Baseline hits, but FormalEvolve has no hit within $T = 100$				
CombiBench	`0037_brualdi_ch10_31`	Difference set $B = \{0, 3, 4, 9, 11\}$ in $\mathbb{Z}_{21}$	> 100	22
CombiBench	`0014_brualdi_ch2_36`	Counting combinations of multisets with bounded repetitions	> 100	39
CombiBench	`0068_imo_2000_p4`	Distribute cards into 3 boxes so sums are equal	> 100	50
CombiBench	`0021_brualdi_ch4_9`	Max inversions in a permutation of $\{1, \ldots, n\}$	> 100	85
ProofNet	`0185_exercise_5_1`	Blaschke condition for zeros of bounded holomorphic functions	> 100	3
ProofNet	`0009_exercise_11_2_13`	Divisibility in Gaussian integers implies divisibility in $\mathbb{Z}$	> 100	40
ProofNet	`0094_exercise_5_4_3`	A root of a polynomial with radicals is algebraic	> 100	49

Across both benchmarks, FormalEvolve improves coverage and reduces concentration (see Table 1 and Figures 4–9).

B.2.2. REFORM-AS-SEEDER: PLUG-IN HANDOVER FROM A STRONG SINGLE-TRAJECTORY SYSTEM

Setup and scope. We evaluate whether FormalEvolve remains beneficial when initialized from a strong single-trajectory autoformalization system, REFORM (Chen et al., 2025), treating it strictly as a *seeder*. This case study is run on CombiBench (N=100) and uses a fixed generator-call budget of 16 per problem, where both seeding and subsequent patch/repair calls are debited within the same budget. For our handover variants, k denotes the number of ReForm seeds per problem; the remaining calls are spent on patch/repair. Importantly, we do *not* claim token-parity with ReForm in this study: token statistics are reported only as a transparency audit, and our main conclusion is based on cross-problem concentration (Gini / top-10% share), not on maximizing SH@16.

Metrics. Beyond hit rates (CH@16 and SH@16), we focus on cross-problem *concentration* of semantic successes (Gini coefficient and top-10% share over per-problem `semantic_ok` counts). These concentration metrics directly quantify whether successes are spread across many problems or concentrated on a small easy subset under a tight call budget.

Table 10. ReForm-as-seeder handover case study on CombiBench ($N = 100$) with a 16-call budget. We treat REFORM as a seeder and evaluate whether plug-in handover reduces cross-problem concentration under a fixed call budget. Primary metrics are concentration of semantic-success counts (Gini and top-10% share over per-problem `semantic_ok`; lower is more uniform); we additionally report CH@16 and SH@16 for context. $\text{SemOK}_{\text{total}}$ denotes the total number of `semantic_ok` candidates aggregated across problems (the sum used to compute Gini / top-share).

Method	Budget	Gini↓	Top-10%↓	$\text{SemOK}_{\text{total}}$	CH@16	SH@16
ReForm (Chen et al., 2025)	sample (8 calls)	0.4196	0.2337	338	96	91
ReForm (Chen et al., 2025)	sample (16 calls)	0.4268	0.2247	672	97	93
Ours (ReForm seed)	16-call handover, k=4	0.3841	**0.1950**	436	96	86
Ours (ReForm seed)	16-call handover, k=6	**0.3813**	0.2032	497	97	91
Ours (ReForm seed)	16-call handover, k=8	0.3912	0.2179	514	97	91
Ours (ReForm seed)	16-call handover, k=9	0.4061	0.2220	545	96	91

Interpretation. Under the same 16-call budget, plug-in handover reduces concentration relative to the ReForm 16-call baseline (e.g., Gini 0.4268 $\rightarrow$ 0.3813 at k=6), indicating more uniform cross-problem distribution of semantic successes. At matched SH@16 (e.g., SH@16= 91), the handover variants achieve lower concentration than ReForm seeding alone (e.g., Gini 0.4196 $\rightarrow$ 0.3813 from the ReForm 8-call baseline to k=6). This improved uniformity comes with a small trade-off in SH@16 (e.g., 93/100 for the ReForm 16-call baseline versus 91/100 at k=6), consistent with reallocating part of the tight budget from seeding to downstream patch/repair.

Token audit (transparency only; response tokens). For ReForm sampling with 16 calls, the response-token statistics (cl100k_base) are mean ≈ 3055, p95 ≈ 8867, max ≈ 21275, and total ≈ 4.89M response tokens over the dataset. For our handover runs, seeding uses $k < 16$ ReForm trajectories per problem, so the response-token footprint is dominated by seeding and scales down accordingly (approximately $(k/16) \times 4.89$M total response tokens, plus patch/rewrite outputs). The additional patch/rewrite response tokens (counting only patch outputs) are orders of magnitude smaller: totals range from ≈ 12.5k to ≈ 45.6k across the dataset (roughly 125–456 response tokens per problem on average, depending on k). These token values are included only as an audit of system-level overhead and are not used to claim compute-matched superiority.

B.3. What drives gains: repair versus online diversity

Under strict budgets, bounded patch/repair calls are the dominant contributor to semantic hit rate: removing patch repair reduces SH@100 by 11/100 (CombiBench) and 13/186 (ProofNet) (Table 1). EvolAST-style fallback has a smaller and non-monotonic effect: on CombiBench it creates a semantic hit on 11 problems but suppresses it on 3 (net +8), whereas on ProofNet it helps on 1 but hurts on 5 (net -4). Because `semantic_ok` is defined by an imperfect semantic judge, EvolAST should be interpreted as a heuristic that can change the distribution of *judge-accepted* candidates; it can help on some instances and hurt on others.

Illustrative instances (EvolAST can help or hurt). The examples below show one instance where EvolAST enables a semantic hit and one where it suppresses a hit under the same strict T=100 budget.

Problem	Dataset	first hit call (w/ EvolAST / w/o)	semantic_ok (w/ EvolAST / w/o)	Δ semantic_ok
`0011_brualdi_ch1_10`	CombiBench	22 / `no hit`	12 / 0	+12
`0014_brualdi_ch2_36`	CombiBench	`no hit` / 48	0 / 3	-3

B.4. Failure modes and diagnostics

We log (i) compilation failures, (ii) semantic drift (judge rejects), and (iii) judge/prover disagreements for qualitative inspection. We keep the appendix discussion lightweight and rely on the audited examples below for concrete illustrations.

Example (judge/prover mismatch). In Appendix C.7.3 (ProofNet `0004_exercise_6_4_2`), both baselines obtain theorem-complete prover outputs under the same RR64 budget, while our best judge-accepted statement is only pass-level. This illustrates non-dominance between semantic judging and prover utility: semantic acceptance does not guarantee a prover-friendly theorem under a fixed budget, and theorem-complete proofs do not automatically imply faithful formalization.

C. Supplementary Material

C.1. Detailed Pseudocode and Budget Accounting

Accounting conventions. Under strict budget accounting, we debit only generator-side LLM calls (proposals and bounded repairs). EvolAST is a symbolic fallback triggered on (i) exact duplicates after canonicalization and (ii) compilation failures; it does not consume budget and is evaluated as the representative candidate of the triggering call. Each compilation-repair or semantic-repair attempt is itself a debited generator call whose representative candidate is the repaired output; semantic repair preserves the original compilable candidate and appends a repaired variant, and both are evaluated. All metrics normalize by the fixed budget T per problem; any early termination or failed call is treated as a consumed call with $C(c) = 0$ and $J(c) = 0$.

Parent selection. Parent selection uses the usage-penalized robust weighting described in Section 3; we do not repeat it here.

Algorithm C.1 FormalEvolve (per problem; detailed)

Input: informal statement x, generator-call budget T, seed model M_{seed}, patch model M_{patch}
Output: feasible archive $\mathcal{A}$ (partitioned into islands)
$t \leftarrow 0$; $\mathcal{A} \leftarrow \emptyset$
$\mathcal{A} \leftarrow \textsc{GenerateInitialCandidates}(x, M_{\text{seed}}, T, t)$ (compile-filtered; debits budget)
while $t < T$ **do**
  $I \leftarrow \textsc{SampleIsland}(\mathcal{A})$ (uniform over initialized islands)
  $p \leftarrow \textsc{SampleParent}(\mathcal{A}_I)$
  $(\mathcal{I}_{\text{arch}}, \mathcal{I}_{\text{top}}) \leftarrow \textsc{SampleContext}(\mathcal{A}_I, p)$
  $c \leftarrow \textsc{Propose}(p, \mathcal{I}_{\text{arch}}, \mathcal{I}_{\text{top}}, x; M_{\text{patch}})$
  $t \leftarrow t + 1$
  $c \leftarrow \textsc{EnforceProtocol}(c, p)$ (preamble inherit; theorem-only; placeholder proof)
  if $\textsc{IsExactDuplicate}(c, \mathcal{A}_I)$ **then**
    $c_{\text{rep}} \leftarrow \textsc{EvolAST}(p)$ (no-call representative)
    $\textsc{ProcessCandidate}(c_{\text{rep}}, x, p, \text{false})$ (no semantic repair)
  else if $\textsc{Compiles}(c)$ **then**
    $\textsc{ProcessCandidate}(c, x, p, \text{true})$
  else
    $c_{\text{rep}} \leftarrow \textsc{EvolAST}(p)$ (compile-fail representative; no call)
    $\textsc{ProcessCandidate}(c_{\text{rep}}, x, p, \text{false})$
    if $t < T$ **then**
      $c_{\text{fix}} \leftarrow \textsc{CompileRepair}(c, x; M_{\text{patch}})$; $t \leftarrow t + 1$ (bounded)
      $c_{\text{fix}} \leftarrow \textsc{EnforceProtocol}(c_{\text{fix}}, p)$
      $\textsc{ProcessCandidate}(c_{\text{fix}}, x, p, \text{true})$
    end if
  end if
  $\textsc{MaybeMigrate}(\mathcal{A})$ (periodic island migration; see Section C.3)
end while

Algorithm C.2 PROCESSCANDIDATE (compile gate, judge, and bounded semantic repair)

Input: candidate c, informal statement x, parent p, allow semantic repair flag $r \in \{\text{true}, \text{false}\}$
Effect: if c is feasible, evaluates $J(c)$ and inserts into the island archive; if $J(c) = 0$ and $r = \text{true}$, performs one bounded semantic-repair call and evaluates the repaired candidate
if COMPILES(c) **then**
  $m \leftarrow$ EVALUATE(c, x) (semantic judge J)
  $\mathcal{A}_I \leftarrow$ INSERTARCHIVE$(\mathcal{A}_I, (c, m))$ ($C(c) = 1$ only)
  if $r \wedge J(c) = 0 \wedge t < T$ **then**
    $c' \leftarrow$ SEMANTICREPAIR$(c, x; M_{\text{patch}})$; $t \leftarrow t + 1$ (bounded)
    $c' \leftarrow$ ENFORCEPROTOCOL(c', p)
    if COMPILES(c') **then**
      $m' \leftarrow$ EVALUATE(c', x)
      $\mathcal{A}_I \leftarrow$ INSERTARCHIVE$(\mathcal{A}_I, (c', m'))$
    end if
  end if
end if

C.2. Budget Audit and Evaluation Overhead

Generator-side (strict). We audit generator-call usage under the strict budget currency ($T = 100$ calls per problem). We decompose the fixed budget $N \times T$ into generation calls (Gen), compilation-repair calls (CRep), and semantic-repair calls (SRep). Table 7 provides a complementary per-problem distributional view (median/IQR) of CRep/SRep under the same strict accounting.

Evaluator-side (semantic judge). We report evaluator overhead in terms of semantic LLM-as-judge calls (CriticLean-Qwen3-14B). In this pipeline, semantic judging is invoked only after successful compilation, and EvolAST outputs are judged under the same gate; EvolAST-Judge is the subset of judge calls attributed to EvolAST-labeled candidates (no-call on the generator side).

Table 11. Strict generator-call budget audit (call currency). Budget is fixed to $N \times T$ with $T = 100$ per problem; Gen/CRep/SRep partition the same budget.

Dataset	Method	Budget	Gen	CRep	SRep
CombiBench	sample	10000	10000	0	0
CombiBench	strong_compile	10000	7031	2969	0
CombiBench	strong_semantic	10000	4257	1991	3752
CombiBench	ours	10000	4895	3042	2063
CombiBench	ours_no_evolast	10000	5139	3428	1433
CombiBench	ours_no_repair	10000	10000	0	0
ProofNet	sample	18600	18600	0	0
ProofNet	strong_compile	18600	13881	4719	0
ProofNet	strong_semantic	18600	10732	3987	3881
ProofNet	ours	18600	11872	4903	1825
ProofNet	ours_no_evolast	18600	11984	4887	1729
ProofNet	ours_no_repair	18600	18600	0	0

Table 12. Semantic judge overhead (CriticLean-Qwen3-14B). JudgeCalls counts how many compilable candidates reach semantic checking; EvolAST-Judge is the EvolAST-attributed subset and is included in JudgeCalls.

Dataset	Method	Problems	JudgeCalls	EvolAST-Judge
CombiBench	sample	100	6953	0
CombiBench	strong_compile	100	5513	0
CombiBench	strong_semantic	100	6109	0
CombiBench	ours	100	4970	863
CombiBench	ours_no_evolast	100	3800	0
CombiBench	ours_no_repair	100	9025	3797
ProofNet	sample	186	13062	0
ProofNet	strong_compile	186	11624	0
ProofNet	strong_semantic	186	11904	0
ProofNet	ours	186	10536	331
ProofNet	ours_no_evolast	186	10103	0
ProofNet	ours_no_repair	186	13465	1008

C.3. Reproducibility: Configuration and Hyperparameters

This section records the configuration used in our main experiments; Table 13 summarizes the key hyperparameters.

Lean 4 toolchain and library. All candidates are compiled under Lean 4.15.0 with Mathlib (`import Mathlib`). We use Mathlib4 `v4.15.0` (commit `9837ca9`) as bundled by our Kimina Lean Server installation. Compilation is served by Kimina Lean Server (Dos Santos et al., 2025), which provides a pinned Lean 4/Mathlib environment suitable for large-scale batch checking.

Table 13. Key hyperparameters used in FormalEvolve.

Component	Setting
Models	Seed: Kimina-Autoformalizer-7B; patch/repair: Qwen3-30B-A3B.
Compilation	Lean 4.15.0 + Mathlib via Kimina Lean Server (Dos Santos et al., 2025).
Semantic judge	CriticLean-Qwen3-14B (LLM-based judge; proof ignored).
Islands	2
Island separation	enabled; parent selection and inspirations restricted to the sampled island
Archive invariant	compilation-feasible (compile_ok$=1$)
Archive size	global capacity 40 (compilation-feasible)
Migration policy	interval 10 gens, rate 0.1; moves feasible gen>0 candidates; elitism (top-1 protected)
Parent selection	weighted ($\lambda = 10$) with usage penalty $\beta = 0.05$
Operator mix (default)	`full`/`diff`/`cross` with probs 0.5/0.3/0.2
Patch attempts	max patch attempts 1 (per step)
Inspiration pool	per step: 4 archive inspirations + 2 top-k inspirations (when available)
Cross prompt inspirations	samples $k = 1$ inspirations uniformly from the pool
Repair budget	max attempts 2, temperature 0.7 (applies to compile + semantic repair)
EvolAST fallback	enabled as a fallback for (i) exact-duplicate edits, and (ii) compilation failures (evaluated as a no-call fallback alongside bounded compile repair)

Implementation notes: archive and island mechanics.

- **Global feasible archive.** We maintain a global compilation-feasible archive (capacity 40); selection is applied only within this feasible set.
- **Island-local behavior.** Parent selection and cross-patch inspirations are restricted to the sampled island (via `island_idx`); children inherit the parent island by default.
- **Migration.** Every 10 generations, we migrate a small fraction of feasible candidates between islands (rate 0.1), while protecting the top-1 elite in each island.

C.4. Additional Uniformity Diagnostics (Gini vs budget)

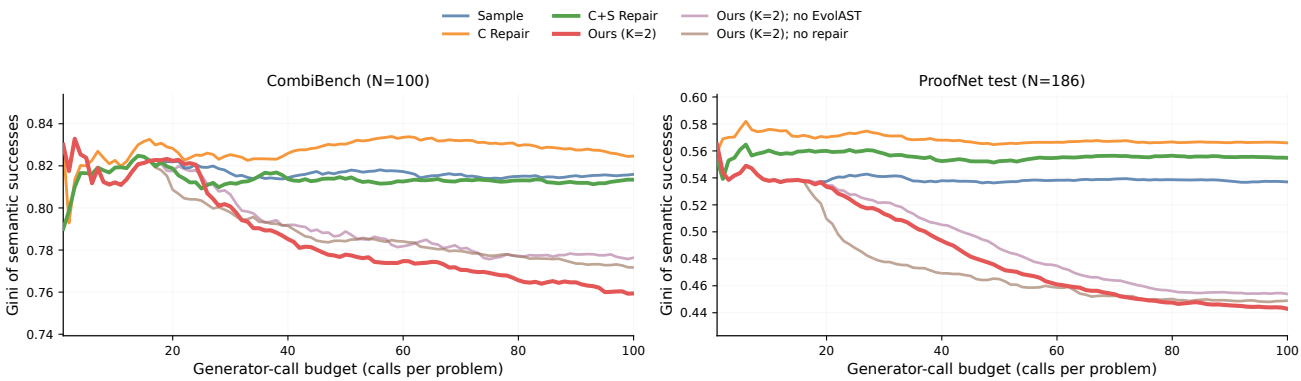


Figure 9. Cross-problem concentration of semantic successes vs generator-call budget, summarized by the Gini coefficient of per-problem semantic-success counts under strict budget accounting. Lower is more uniform (less concentrated on a small subset of problems). Table 1 reports the corresponding summary at $T = 100$.

C.5. Prompting Protocol (Summary)

This section records the prompt templates used in our implementation, following the spirit of reflective autoformalization work (e.g., REFORM) but tailored to our population-based setting. We present the generator prompts (initial statement generation and evolution operators), bounded repair prompts, and the semantic judge prompt used by our judge configuration (CriticLean-Qwen3-14B).

Notation. Placeholders include {informal} (natural-language statement), {code_content} (current Lean file), {performance_metrics} (compile/semantic metrics), {text_feedback_section} (optional judge feedback), {original_code} (candidate file for repair), {compile_error_type}/{compile_error_msg} (compiler feedback), {critic_feedback} (semantic-judge feedback), {inspiration_code_1} (cross inspiration), and {lean_statement} (theorem statement for judging). For pdfLaTeX compatibility, we render common Lean unicode symbols in ASCII (e.g., Complex, Omega, and right-arrow tokens).

Table 14. Prompt components used in our pipeline (summary).

Component	Model	Purpose / output constraints
Initial generation	M_{seed}	Produce one Lean 4 file with a single theorem statement (`:= by sorry`).
Operators (full/diff/cross)	M_{patch}	Edit-conditioned rewrite of a parent; cross additionally conditions on one inspiration from the archive.
Compile repair (bounded)	M_{patch}	Minimal patch conditioned on compiler feedback; debited within the same call budget.
Semantic repair (bounded)	M_{patch}	Revision conditioned on the informal statement and judge feedback; debited within the budget.
Semantic judge	`CriticLean-Qwen3-14B`	Binary verdict on statement faithfulness (proof ignored), returned as JSON (`Correct`/`Incorrect` plus a short rationale).

C.5.1. STATEMENT GENERATION (INITIAL)

Prompt template.

```
You are an expert in Lean 4 theorem proving and Mathlib.

Given a problem statement in natural language, write a COMPLETE Lean 4 file (imports +
theorem)
that formalizes the mathematical content.

MANDATORY OUTPUT REQUIREMENTS:
- Output EXACTLY ONE Lean 4 code block: ```lean ... ```.
- The code MUST start with the following two lines (in this order):
  import Mathlib
  import Aesop
- You MAY add additional `import ...` / `open ...` / `set_option ...` lines after that
if needed (unless a mode-specific prefix locks the header, e.g., diff mode).
- Do NOT include any comments.
- Include EXACTLY ONE `theorem` declaration.
- The theorem MUST end with `:= by sorry` (do not provide a proof).
- Use Lean 4 v4.15-compatible syntax and Mathlib definitions.

Natural language statement:
{informal}

Return ONLY the Lean 4 code block.
```

C.5.2. EVOLUTION OPERATORS (FULL / DIFF / CROSS)

Shared prompt skeleton.

```
You are an expert in Lean 4 theorem proving and the Mathlib library.
```

```
Your task is to improve a Lean 4 formalization for a given natural language mathematical
claim.

Hard requirements (must satisfy all):
- Output a COMPLETE Lean 4 file (imports + exactly one theorem) in a single ```lean```
code block.
- The code MUST start with these two lines (in this order):
  import Mathlib
  import Aesop
- You MAY add additional `import ...` / `open ...` / `set_option ...` lines after that
if needed.
- Do NOT include any comments.
- Include EXACTLY ONE `theorem` declaration.
- The theorem MUST end with `:= by sorry` (do not provide a proof).

Quality goals:
- compile_ok: The file compiles without errors.
- semantic_ok: The theorem statement matches the informal mathematical meaning.

Natural language statement:
{informal}

Current Lean 4 program:
```lean
{code_content}
```

Evaluation results:
{performance_metrics}{text_feedback_section}

Return ONLY the Lean 4 code block.
```

Full-mode addenda (sampled variants; one per call).

```
(i) Default rewrite: revise types/hypotheses/structure as needed.
(ii) Different interpretation: try a different reasonable formalization of the same
claim.
(iii) Type optimization: improve binder structure and type-class constraints.
(iv) Mathlib alignment: prefer standard Mathlib names and definitions.
(v) Simplification: if possible, state a simpler claim without changing meaning.
```

Diff mode prepends a short local-edit instruction (still returning a full file), while cross mode additionally provides one sampled inspiration candidate from the archive/top-k pool as extra context.

Diff-mode prefix (local edit).

```
You are doing a LOCAL EDIT of a Lean 4 formalization file.
Header lock (diff mode only):
- Do NOT add/remove/reorder/modify any preamble lines (imports/opens/options).
- Keep everything before the `theorem` declaration EXACTLY unchanged.
Goal:
- Make the smallest possible change to improve compilation and formalization quality.
- Keep changes minimal; do NOT change the intended mathematical meaning.
```

Cross-mode prefix (inspiration-guided).

```
You are doing CROSS patching.
You are given one inspiration candidate from the archive/top-k pool as extra context.
```

````
Your task is to improve the current formalization by optionally borrowing useful
structure from the inspiration.
````

Cross-mode extra context (one per call).

````
[Inspiration candidate]
```lean
{inspiration_code_1}
```
````

C.5.3. BOUNDED REPAIR PROMPTS

Repairs count toward the generator-call budget. Compilation repair triggers only when `compile_ok`=0; semantic repair triggers only when `compile_ok`=1 and `semantic_ok`=0.

Compilation repair prompt.

````
You are an expert in Lean 4 theorem proving and Mathlib.
You are doing SYNTAX / COMPILATION REPAIR.

You will receive:
- A natural language statement (the semantic target)
- A Lean 4 file that fails to compile
- Compiler error feedback

Your job:
- Produce a corrected Lean 4 file that compiles.

IMPORTANT:
- Fix compilation only; do NOT change the intended mathematical meaning.
- Keep changes minimal (types, identifiers, imports, binder annotations, etc.).

Natural language statement:
{informal}

Current Lean 4 code (does NOT compile):
<CURRENT_CODE>
{original_code}
</CURRENT_CODE>

Compiler feedback:
- Error type: {compile_error_type}
- Error message:
```
{compile_error_msg}
```

Output requirements:
1) Output EXACTLY ONE ```lean``` code block (Lean 4; no extra text).
2) The code MUST start with:
   import Mathlib
   import Aesop
3) Do NOT include any comments.
4) Include EXACTLY ONE theorem, and end it with `:= by sorry`.
````

Semantic repair prompt.

````
You are an expert in mathematics and Lean 4 (Mathlib).
````

```
You are doing SEMANTIC REPAIR.

You will receive:
- A natural language statement (the semantic target)
- A Lean 4 file that is intended to formalize it
- Critic feedback (Accuracy Confirmation) describing mismatches

Your job:
- Modify the Lean 4 code so that the theorem statement matches the natural language
statement.

Rules:
- You MAY change hypotheses and the conclusion if needed to match the semantics.
- You MAY adjust/add imports/opens/options as needed to keep the file compiling.
- Do NOT "solve" the task by weakening it to `True` or a tautology.
- Do NOT include any comments.
- Output a complete Lean 4 file starting with:
  import Mathlib
  import Aesop
- Include exactly one theorem, ending with `:= by sorry`.

Natural language statement:
{informal}

Current Lean 4 code:
<CURRENT_CODE>
{original_code}
</CURRENT_CODE>

Critic feedback (Accuracy Confirmation):
{critic_feedback}

Goal:
- Modify the Lean 4 theorem so that the semantic judge would accept it as an exact
formalization of the natural-language statement.
- Address every mismatch mentioned in the Critic feedback.

Output requirements:
1) Output EXACTLY ONE ```lean``` code block (Lean 4; no extra text).
2) The code MUST start with:
   import Mathlib
   import Aesop
3) Do NOT include any comments.
4) Include EXACTLY ONE theorem, and end it with `:= by sorry`.
```

C.5.4. SEMANTIC JUDGE PROMPT (CRITICLEAN-QWEN3-14B)

We use an LLM-based semantic judge instantiated as CriticLean-Qwen3-14B to judge semantic consistency between the natural-language statement and the Lean 4 theorem statement (proof ignored). The prompt template below is used for each semantic check.

```
Role: Lean 4 & Formal Verification Expert
Input:
- Mathematical_Text: A math problem and its answer (no proof).
- Lean4Code: A Lean 4 theorem statement formalizing the problem (proof intentionally
omitted).
Goal:
Determine if the Lean 4 theorem statement is an exact and faithful formalization of the
mathematical problem.
Do not evaluate or consider the answer or the proof. Your sole task is to verify the
correctness of the formalization.

Evaluation Stages (All required):
1. Math Assertion Analysis
```

```
Identify all structurally and semantically relevant components of the mathematical
problem, including
variables, types, quantifiers, constraints, logic structure, conclusion, and so on. The
analysis should be
based on the actual content of the text.

2. Lean 4 Statement Analysis (ignore proof part)
Extract all structurally and semantically relevant components from the Lean 4 statement,
including
variables, types, conditions, quantifiers, constraints, the final claim, and so on. The
analysis should
reflect the actual content present in the Lean 4 code.

3. Comparative Verification
Check for exact correspondence between the math and Lean 4 statements; you may refer to
aspects like:
- Semantic alignment, logic structure, and quantifier correctness.
- Preservation of constraints and boundary assumptions.
- Accurate typing and use of variables.
- Syntactic validity and proper Lean 4 usage (free from errors).
- Use of symbols and constructs without semantic drift.
- No missing elements, no unjustified additions, and no automatic corrections or
completions.

4. Final Judgement
Based solely on the above analysis, judge whether the Lean 4 statement is a correct and
exact formalization
of the mathematical problem.

5. Accuracy Confirmation
If correct: clearly confirm why all elements match.
If incorrect: list all mismatches and explain how each one affects correctness.

Note: The final judgment must be based only on what is explicitly and formally expressed
in the Lean 4 statement.
Do not consider or assess any part of the proof. Your judgment should be entirely about
the accuracy of the statement formalization.

Output Format:
Return exactly one JSON object:
{
  "reasons": "<your detailed analysis as a single string>",
  "is_assistant_correct": "Correct or Incorrect"
}

Input Data:
-- Start of Mathematical_Text --
{informal}
-- End of Mathematical_Text --
-- Start of Lean4Code --
{lean_statement}
-- End of Lean4Code --
```

C.5.5. PROVER PROMPT (GOEDEL-PROVER-V2-32B)

We evaluate downstream proof utility using a fixed prover configuration (Goedel-Prover-V2-32B) and a fixed prompt template. The prompt asks the prover to complete a Lean file by replacing the placeholder proof.

````
Complete the following Lean 4 code.
Return ONLY a complete Lean 4 file inside a ```lean4``` code fence (no explanations).
```lean4
{lean_file}
```
````

C.5.6. PROVER PROMPT (GOEDEL-PROVER-V2-32B)

We evaluate downstream proof utility using a fixed prover configuration (Goedel-Prover-V2-32B) and a fixed prompt template. The prompt asks the prover to complete a Lean 4 file by replacing the placeholder proof.

```
Complete the following Lean 4 code.
Return ONLY a complete Lean 4 file inside a ```lean4 code fence (no explanations).

```lean4
{lean_file}
```
```

C.5.7. JUDGE RELIABILITY ON CONSISTENCYCHECK

We evaluate the deployed semantic judge configuration (CriticLean-Qwen3-14B; served as `criticlean-qwen3-14b`) on two public benchmarks. Following our end-to-end pipeline semantics, compilation failures are rejected by the compilation gate and counted as negative predictions.

Table 15. Semantic judge evaluation under our deployed configuration (end-to-end compile gate).

Dataset	N	ACC	Precision	Recall	F1
CONSISTENCYCHECK (Chen et al., 2025)	859	0.790	0.823	0.872	0.847
CRITICLEANBENCH (Peng et al., 2025)	500	0.810	0.770	0.884	0.823

The CONSISTENCYCHECK accuracy closely matches REFORM's report that CriticLean-14B achieves 79.1% accuracy on CONSISTENCYCHECK, and provides additional evidence that semantic judging remains a noisy proxy under practical prompting and serving conditions (Chen et al., 2025). They further report that a non-trivial fraction of human-written formalizations in existing benchmarks can contain semantic errors (16.4% in MiniF2F and 38.5% in ProofNet), underscoring the intrinsic difficulty of semantic faithfulness (Chen et al., 2025).

C.6. Prover-Stage Audit (Concise): What Proof Success Certifies

Scope and claim boundary. We audit the prover stage on the subset of semantically consistent statements (judge-accepted) produced within the generator-call budget (Section 4). A prover-complete Lean file certifies that the produced formal artifact is accepted by Lean and contains no remaining `sorry`. It does not, by itself, certify faithfulness to the original informal statement, since semantic consistency is a noisy proxy (Appendix C.5.7).

Prompt reference. The prover prompt template is fixed (Appendix C.5.6); we omit repeating it in each case study and report only the proved formal statement and the prover's response.

ASCII normalization. For compatibility with pdfLaTeX, raw transcripts in `PromptBox` are displayed after a lightweight ASCII-only normalization of common Unicode math symbols (e.g., rendering quantifiers and connectives in ASCII form). This affects presentation only.

Theorem-complete reporting. In addition to standard pass@64 and complete@64, we report *theorem-complete@64*: a problem counts as theorem-complete@64 if at least one prover attempt yields a Lean-accepted, `sorry`-free file that contains an explicit `theorem` or `lemma`. This guards against Lean-complete outputs that only introduce auxiliary `def`/`abbrev`/`example` content without proving a named proposition.

Prompt categories. Table 16 also reports a coarse prompt-type audit *over prover attempts* (not over problems): the denominator is the number of RR64 prover prompts actually issued under a fixed per-problem attempt budget (so it scales with the number of problems that have a non-empty statement repertoire for proving). We label an attempt as "Prompt has theorem" if the candidate statement sent to the prover contains an explicit `theorem`/`lemma` declaration. We label an attempt as "Prompt abbrev-only" if the candidate contains an `abbrev` declaration and no `theorem`/`lemma`. (Remaining attempts typically correspond to `def` or `example` templates and are omitted from this summary.)

Table 16. Prover-stage audit statistics under $B{=}64$ prover attempts per problem. Prompt-category counts are reported *per prover attempt* (denominator: number of RR64 prompts issued), not per problem. We report (i) whether the prover prompt contains a `theorem`/`lemma` declaration vs. abbrev-only prompts, and (ii) pass@64/complete@64 together with theorem-complete@64.

Benchmark	Method	Prompt has theorem	Prompt abbrev-only	pass@64	complete@64	theorem-complete@64
CombiBench	Ours	2299/3648 (0.630)	384/3648 (0.105)	44/100	27/100	13/100
CombiBench	Strong	1813/2944 (0.616)	384/2944 (0.130)	40/100	23/100	8/100
CombiBench	Sample	1792/2816 (0.636)	256/2816 (0.091)	41/100	23/100	8/100
ProofNet	Ours	9136/9536 (0.958)	192/9536 (0.020)	127/186	52/186	45/186
ProofNet	Strong	8938/9280 (0.963)	128/9280 (0.014)	119/186	50/186	46/186
ProofNet	Sample	8164/8512 (0.959)	128/8512 (0.015)	106/186	46/186	41/186

Takeaway. On ProofNet, most prover prompts contain a `theorem`/`lemma` (around 0.96), so theorem-complete filtering is close to the standard metrics. On CombiBench, 9–13% of prompts are abbrev-only; theorem-complete@64 is therefore a stricter and more robust success signal.

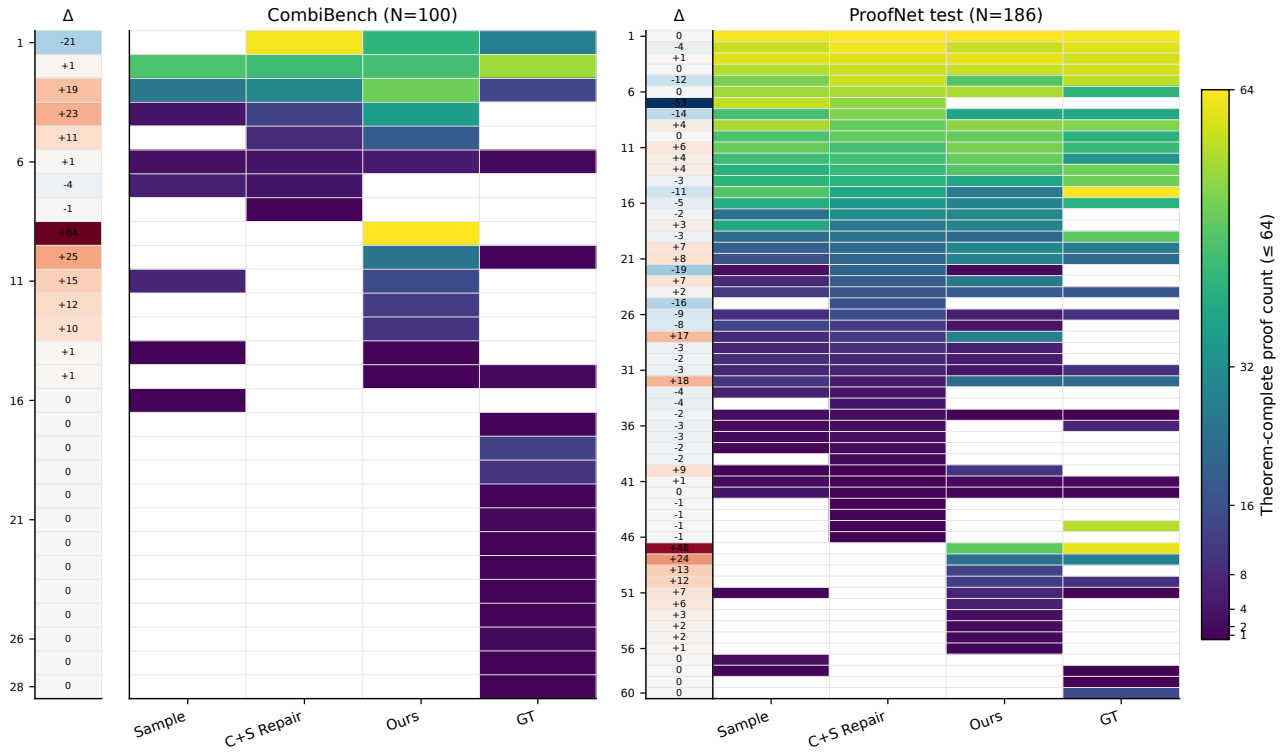


Figure 10. Filtered per-problem theorem-complete attempt counts under RR64 (cell = number of attempts, out of $B{=}64$, that yield a `theorem`/`lemma` without `sorry`). Rows show only problems where at least one method attains at least one theorem-complete attempt (so the number of displayed problems is smaller than N). Generator-based methods only attempt statements that pass the semantic judge (`semantic_ok`$= 1$), whereas the Ground truth column runs the prover on the dataset-provided ground-truth formal statement (one per problem), bypassing the judge.

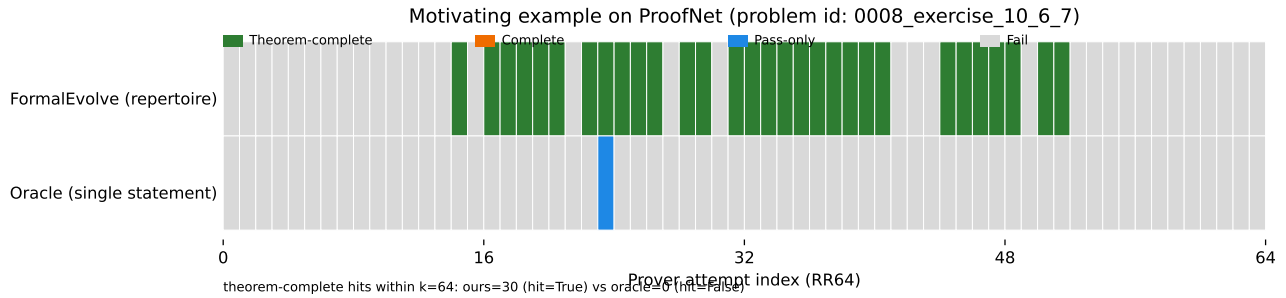


Figure 11. Data-driven motivating example under RR64 on ProofNet: even among semantically consistent candidates, prover outcomes can vary substantially under a fixed prover and attempt budget. The oracle uses a single canonical statement (retried under RR64), while FormalEvolve provides a diverse repertoire of semantically consistent statements, increasing the chance that at least one provable formulation is found within budget.

Navigation. We consolidate qualitative examples into a short, easy-to-find appendix block (2 wins + 1 failure) in Appendix C.7.

C.7. Case Studies (2 successes + 1 non-dominance example)

Scope. We keep qualitative evidence minimal: two short success cases that illustrate prover-stage outcomes under the same RR64 budget (one where baselines fail to produce any Lean 4-accepted proof output, and one where a baseline reaches only pass-level artifacts), plus one counterexample that highlights non-dominance and judge/prover mismatch. To contextualize these examples, Table 8 quantifies prover-stage non-dominance frequency at theorem-complete@64 (FormalEvolve versus Compile+Semantic Repair) under the same RR64 protocol.

C.7.1. Theorem-complete proof: ours completes; baselines have no pass

Benchmark: `proofnet`.
Problem id: `0088_exercise_4_3_1`.
Informal statement:

```
If $R$ is a commutative ring and $a \in R$, let $L(a) = \{x \in R \mid x a = 0\}$. Prove
that $L(a)$ is an ideal of $R$.
```

Outcome under the same RR64 prover budget. Our run attains a theorem-complete proof, while both sampling baselines fail to produce any Lean 4-accepted proof output within $B = 64$ attempts (no `pass`).

Takeaway. This instance illustrates a practical benefit of a repertoire: under a fixed prover attempt budget, the ability to try multiple semantically consistent statements can be the difference between reaching a complete proof and having no Lean-accepted proof output at all.

Notation. Lean 4 admits both Unicode and ASCII forms. For pdfLaTeX compatibility, we display prover transcripts in ASCII, where quantifiers/connectives may appear as `forall`/`->` and the constructor names `And`/`Or`, corresponding to $\forall/\to$ and $\wedge/\vee$.

Prover output excerpt (ours; theorem-complete).

```
import Mathlib

open Fintype Set Real Ideal Polynomial
open scoped BigOperators
noncomputable section

theorem my_theorem_123 {R : Type*} [CommRing R] (a : R) :
  forall x y : R, x * a = 0 -> y * a = 0 ->
```

```
    And ((x + y) * a = 0)
      (And ((-x) * a = 0)
        (forall r : R, And ((r * x) * a = 0) ((x * r) * a = 0))) := by
intro x y hx hy
have h1 : (x + y) * a = 0 := by
  calc
    (x + y) * a = x * a + y * a := by ring
    _ = 0 + 0 := by rw [hx, hy]
    _ = 0 := by simp
have h2 : (-x) * a = 0 := by
  calc
    (-x) * a = - (x * a) := by ring
    _ = -0 := by rw [hx]
    _ = 0 := by simp
have h3 : forall r : R, And ((r * x) * a = 0) ((x * r) * a = 0) := by
  intro r
  have h4 : (r * x) * a = 0 := by
    calc
      (r * x) * a = r * (x * a) := by ring
      _ = r * 0 := by rw [hx]
      _ = 0 := by simp
  have h5 : (x * r) * a = 0 := by
    calc
      (x * r) * a = x * (r * a) := by ring
      _ = x * (r * a) := by rfl
      _ = (r * x) * a := by ring
      _ = 0 := by
        calc
          (r * x) * a = r * (x * a) := by ring
          _ = r * 0 := by rw [hx]
          _ = 0 := by simp
  exact And.intro h4 h5
exact And.intro h1 (And.intro h2 h3)
```

```
Ours:    pass=True,  complete=True,  sorries=0
Strong:  pass=False
Sample:  pass=False
```

C.7.2. COMPLETE PROOF: OURS COMPLETES; BASELINE IS PASS-ONLY

Benchmark: `proofnet`.
Problem id: `0000_exercise_2_3_2`.
Informal statement:

```
Prove that the products $a b$ and $b a$ are conjugate elements in a group.
```

Outcome under the same RR64 prover budget. Our run reaches a theorem-complete proof (`complete=True, sorries=0`), while Compile+Semantic Repair only reaches pass-level artifacts with remaining `sorry`.

Prover output excerpt (ours; theorem-complete).

```
import Mathlib

theorem my_theorem_xxx {G : Type*} [Group G] (a b : G) :
  exists g : G, a * b = g * (b * a) * g^(-1) := by
  refine Exists.intro a ?_
  simp [mul_assoc, mul_left_inv, mul_right_inv]
```

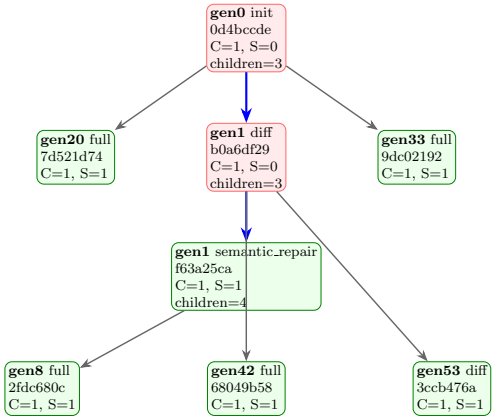


Figure 12. ProofNet `0000_exercise_2_3_2` statement evolution trace (C=compile-ok, S=semantic-ok). The repair step removes an extra conjunction and introduces the conjugation structure (the g^{-1} term).

Statement evolution trace (mechanism note). This instance illustrates why we separate compilation from semantic checking: an early candidate may compile yet express a non-standard (and generally stronger) variant of the target claim, which can be judged inconsistent and/or harder for the prover to use under a fixed attempt budget. Here, the parent uses a conjunction of two equations, whereas the informal goal is a single conjugacy statement of the form $a * b = g * (b * a) * g^{-1}$. Bounded semantic repair removes the extra conjunct and rewrites the statement into this standard form while preserving compilation feasibility (Figure 12).

```
init (C=1,S=0): exists g, a*b = g*b*a /\ b*a = g*a*b
repair (C=1,S=1): exists g, a*b = g*(b*a)*g^(-1)
```

C.7.3. Non-dominance case: baseline theorem-complete; ours pass-only

Benchmark: `proofnet`.
Problem id: `0004_exercise_6_4_2`.
Informal statement:

```
Prove that no group of order p q, where p and q are prime, is simple.
```

Outcomes under the same RR64 prover budget. In this instance, our best judge-accepted statement yields `pass=True` but not a complete proof, while both sampling baselines obtain a theorem-complete proof within the same prover attempt budget.

Takeaway. This example highlights non-dominance: even under the same prover budget, different semantically accepted statements can differ substantially in proof-search friendliness, and prover success does not automatically imply semantic faithfulness.

Prover output excerpt (ours; pass-only). The excerpt below contains remaining `sorry`; baseline theorem-complete outputs are omitted for space. Ours (pass-only):

```
import Mathlib

theorem my_theorem_12345 {G : Type*} [Group G] [Fintype G] (p q : Nat)
```

```
  (hp : p.Prime) (hq : q.Prime) (hcard : Fintype.card G = p * q) :
    Not (IsSimpleGroup G) := by
  have h_main : Not (IsSimpleGroup G) := by sorry
  sorry
```

```
Ours:    pass=True,  complete=False, sorries=1
Strong:  pass=True,  complete=True,  sorries=0
Sample:  pass=True,  complete=True,  sorries=0
```

C.8. Early proof successes under fixed prover budgets

Table 17 lists representative instances where FormalEvolve reaches a successful proof substantially earlier than Compile+Semantic Repair under the same prover attempt budget, or succeeds when the baseline fails within $B = 64$ attempts. We report the first prover attempt index k (under the round-robin schedule described in Section A.1) where the prover returns (i) any proof script accepted by Lean 4 (pass) or (ii) a complete proof without `sorry` (complete).

Table 17. Representative early proof successes under a fixed per-problem prover attempt budget $B = 64$, computed on the semantically consistent repertoire produced within $T = 100$ generator calls. We report the first prover attempt index k where the prover succeeds; "> 64" indicates no success within the attempt budget.

Benchmark	Criterion	Problem (id + short description)	Ours first k	C+S Repair first k
Both succeed, but FormalEvolve is much earlier				
ProofNet	pass	`0113_exercise_18_8a` – order topology: $\{x \mid f(x) \leq g(x)\}$ is closed	1	58
ProofNet	complete	`0113_exercise_18_8a` – order topology: $\{x \mid f(x) \leq g(x)\}$ is closed	12	58
ProofNet	pass	`0174_exercise_5_3` – $f(x) = x + \varepsilon g(x)$ is injective for small ε	2	64
ProofNet	complete	`0174_exercise_5_3` – $f(x) = x + \varepsilon g(x)$ is injective for small ε	13	64
CombiBench	pass	`0047_brualdi_ch13_10` – tournament radius-2 vertex exists	9	27
CombiBench	complete	`0047_brualdi_ch13_10` – tournament radius-2 vertex exists	21	27
CombiBench	pass	`0057_imosl_2015_c6` – infinitely many non-clean integers	1	5
CombiBench	complete	`0057_imosl_2015_c6` – infinitely many non-clean integers	3	15
FormalEvolve succeeds, baseline fails within $B = 64$				
ProofNet	complete	`0000_exercise_2_3_2` – ab and ba are conjugate in a group	2	> 64
ProofNet	complete	`0046_exercise_3_4_5b` – solvable quotients are solvable	1	> 64
CombiBench	complete	`0007_hackmath_8` – ferry crossing: women in first group	1	> 64
CombiBench	complete	`0063_usamo_2000_p4` – 3 colored squares make a right triangle	1	> 64

C.9. Ablation Note: When EvolAST changes outcomes on ProofNet (small effect)

On ProofNet, enabling EvolAST slightly decreases semantic hit@100 under strict budget accounting (158/186 vs 162/186). The gap is small (4 problems), but it illustrates an important point for budgeted search: a diversity operator can change the search trajectory and archive composition, and may not be universally beneficial across datasets.

Where the difference comes from. Under strict budget accounting, there are 5 problems that achieve at least one semantically consistent statement under the no-EvolAST configuration but not under the main configuration, and 1 problem where the reverse holds. Because the generator and patching process are stochastic, these disagreements reflect diverging search trajectories under nearly identical protocols; attributing causality to any single operator requires careful per-instance inspection. Empirically, these disagreements are consistent with a mixture of direct trigger effects (duplicates/compile-fail fallbacks) and indirect trajectory effects, rather than a deterministic monotonic gain from the operator.

Table 18. ProofNet instances where the main run and the no-EvolAST ablation disagree on semantic hit@100 under strict budget accounting. We report the first call index where a semantic success appears (when it does), computed from the strict budget accounting logs.

Problem id	Short description	Main (repair + EvolAST)	No EvolAST (repair only)
`0066_exercise_8_3_6b`	$\mathbb{Z}[i]/(q)$ is a field with q^2 elements (for $q \equiv 3 \bmod 4$)	no hit	hit at call 42
`0070_exercise_9_4_9`	$x^2 - \sqrt{2}$ irreducible over $\mathbb{Z}[\sqrt{2}]$	no hit	hit at call 65
`0142_exercise_4_15a`	Uniform continuity iff having a modulus of continuity	no hit	hit at call 49
`0151_exercise_1_8`	No ordered-field structure on $\mathbb{C}$	no hit	hit at call 39
`0185_exercise_5_1`	Blaschke condition: $\sum_n (1 - \lvert z_n \rvert) < \infty$ for bounded holomorphic f	no hit	hit at call 66
`0123_exercise_25_9`	Identity component is a normal subgroup	hit at call 87	no hit

Representative example (problem `0142_exercise_4_15a`). In this instance, the main configuration produced many compilable candidates but no semantic success, whereas the no-EvolAST configuration eventually reaches a judge-accepted statement after semantic repair. One characteristic we observe in the main run is that EvolAST fallback triggers repeatedly on duplicates and compile failures, producing compilable but judge-rejected variants; in this problem, many candidates are explicitly tagged as EvolAST fallbacks, and all have $C(c) = 1$ but $J(c) = 0$. This can bias the archive toward structurally perturbed but semantically misaligned candidates under a strict budget, which in turn alters parent selection and patch contexts.

Concretely, the semantic judge flags a common failure mode here: *domain mismatch* (e.g., proving uniform continuity on $\mathbb{R}$ rather than restricting to $[a, b]$). The best candidate found by the main run (compilable but judge-rejected) has the unconstrained target:

$$(\exists \mu,\ \cdots \wedge\ \forall s, t \in \mathbb{R},\ |f(s) - f(t)| \le \mu(|s - t|)) \ \leftrightarrow\ \texttt{UniformContinuous}\ f$$

whereas the successful no-EvolAST semantic-repair output explicitly restricts the bound to $s, t \in [a, b]$ and targets uniform continuity on the interval:

$$(\exists \mu,\ \cdots \wedge\ \forall s, t,\ s \in \texttt{Icc}\ a\ b \to t \in \texttt{Icc}\ a\ b \to |f(s) - f(t)| \le \mu(|s - t|))$$
$$\leftrightarrow\ \texttt{UniformContinuousOn}\ f\ (\texttt{Icc}\ a\ b)$$

This illustrates a fundamental limitation of a conservative type-rewrite operator: our EvolAST rule set does not introduce new problem-specific binders (e.g., adding explicit a, b interval parameters) or add new domain-restriction hypotheses; it only rewrites within the existing binder and goal AST. Therefore, when the current parent trajectory has not yet reached a semantically correct "skeleton", applying EvolAST tends to preserve the same semantic mistake, producing many feasible variants that remain judge-rejected. This effect is amplified by our implementation choice that EvolAST candidates are not sent into LLM repair (to avoid unbudgeted chains), so they cannot be subsequently corrected by semantic repair.

Takeaway. These examples support a conservative stance: online diversity operators are not guaranteed to monotonically improve semantic hit rates under a fixed budget, and their benefit can be dataset-dependent. In practice, this motivates making the EvolAST rule set and trigger conditions tunable, and reporting ablations transparently rather than assuming a universal gain.

Cross patching (generation 6). Cross patching instantiates the cross operator prompt (above) with (i) the current parent program, (ii) the informal statement, (iii) evaluation feedback (showing semantic_ok$= 0$), and (iv) one sampled inspiration candidate from the archive. In this example, the cross patch yields a compilable and semantically consistent candidate (compile_ok$= 1$, semantic_ok$= 1$).

EvolAST (aggressive rule-based mode). In our experiments, EvolAST is used in its rule-based mode: it applies a bounded sequence of conservative, semantics-intended AST rewrites to the theorem type (binder types and goal type), keeping the proof body unchanged. The rewrite surface is intentionally controlled and includes: conservative hypothesis reordering; commutativity/associativity/distributivity for conjunction/disjunction (Lean: $\wedge/\vee$); symmetry swaps (e.g., $a = b$ to $b = a$, $P \leftrightarrow Q$ to $Q \leftrightarrow P$); and dual relations (e.g., $a < b$ to $b > a$, $a \le b$ to $b \ge a$).

To illustrate both a small and a non-trivial rewrite under the same operator, we show the corresponding Lean-style logical forms:

$$P \wedge Q \ \Rightarrow\ Q \wedge P \qquad \text{(commutativity)}$$

$$(b > a) \wedge \big((P \wedge Q) \vee R\big) \ \Rightarrow\ \big(((b > a) \wedge P) \wedge Q\big) \vee \big((b > a) \wedge R\big) \qquad \text{(distributivity + associativity)}$$

All EvolAST outputs are filtered by the compilation gate before entering the archive, and by the semantic judge before contributing to the semantically consistent repertoire $\tilde{G}_t$ used for SH@T and downstream proving.