SAIR-Comb : A Structure-Aware Iterative Refinement Framework for Combinatorics Autoformalization

Weijie Jiang[1], Gaolei He[1], Beibei Xiong[1], Jianlin Wang[2], Zhengfeng Yang[1]*
[1]East China Normal University
[2]Henan University
{jasinjiang, glhe, bbxiong}@stu.ecnu.edu.cn
jlwang@henu.edu.cn
zfyang@sei.ecnu.edu.cn

Abstract

Autoformalization aims to bridge the gap between human mathematical intuition and formal proof by automating the translation of informal reasoning into machine-verifiable languages. Despite significant breakthroughs catalyzed by Large Language Models (LLMs), autoformalizing Combinatorics remains a formidable challenge due to its intricate structural dependencies and the severe scarcity of high-quality formal datasets. To address these challenges, we propose *SAIR-Comb*, a **S**tructure-**A**ware **I**terative **R**efinement framework for **Comb**inatorics powered by Lean 4 and LLMs. *SAIR-Comb* employs a multi-stage pipeline: first, it performs data augmentation and refinement by rectifying syntactic, semantic, and structural errors, guided by a curated manual combinatorics dataset. The model then undergoes a two-stage training regime: expert iteration with syntactic grounding, followed by reinforcement learning (RL) to align formal reasoning trajectories. Furthermore, we introduce Structural Consistency—a rigorous new metric designed to expose formalizing failures that elude traditional semantic-only evaluations. Experiments demonstrate that *SAIR-Comb* achieves strong performance on the specialized CombiBench while remaining highly competitive on general-domain benchmarks, including PutnamBench and ProverBench.

1 Introduction

Large language models (LLMs) have catalyzed significant advances in mathematical reasoning, substantially propelling the development of automated theorem proving (ATP) (Wu et al., 2022; Yang et al., 2024; Xin et al., 2025; Lin et al., 2025b,a; Ren et al., 2025; Wang et al., 2025). ATP requires models to derive rigorous logical proofs from formal theorem statements, which are subsequently verified within formal systems such as Lean 4 (Moura and Ullrich, 2021). However, the severe scarcity of high-quality formal training data remains a critical bottleneck (Gao et al., 2025). Autoformalization (Weng et al., 2025)—the automated translation of natural-language mathematical discourse into machine-verifiable representations—has emerged as a pivotal technique for synthesizing large-scale, high-fidelity datasets. Recent works in autoformalization have significantly improved syntactic and semantic accuracy through techniques such as Retrieval-Based Translation (Gao et al., 2025), expert iteration (Wang et al., 2025), knowledge-reasoning fusion (Wu et al., 2026), CoT Reasoning (Lin et al., 2025b), and tool feedback (Guo et al., 2025), achieving strong performance on competition-level benchmarks (Tsoukalas et al., 2024; Yu et al., 2025). However, these methods largely target algebra and number theory, often neglecting structural alignment between informal and formal representations in domains with high modeling demands. Combinatorics, as a foundational pillar of discrete mathematics, presents unique challenges: its propositions are intrinsically structured, requiring the precise modeling of discrete objects, complex relations, and recursive constructions (Merris, 2003). This structural density makes combinatorial formalization exceptionally prone to structural errors—logical failures where the formal output may appear semantically plausible but collapses under rigorous structural scrutiny. To date, specialized research remains limited, and there is a conspicuous lack of targeted data augmentation frameworks or domain-specialized models capable of preserving structural fidelity. Furthermore, current evaluation metrics exhibit significant limitations. Bidirectional equivalence (BEq) (Liu et al., 2025b; Poiroux et al., 2025) checks often fail on complex combinatorial structures due to restricted tactic sets, while semantically-focused LLM-as-Judge (Zhang et al., 2025) approaches tend to be overly permissive and often fail to detect structural

*Corresponding author. Zhengfeng Yang

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 23452–23473

inconsistencies in formal representations. To address these gaps, we first introduce Structural Consistency, a rigorous new metric designed to assess the structural integrity of formal expressions and expose formalizing failures that elude traditional semantic-only evaluations. Finding that current methods often struggle with such structural fidelity, we propose *SAIR-Comb*, a structure-aware iterative refinement framework that leverages a manually curated combinatorics dataset to drive a specialized data augmentation pipeline for rectifying syntactic, semantic, and structural discrepancies, integrated with a two-stage training strategy encompassing expert-iteration with syntactic grounding and reinforcement learning-based trajectory alignment. Our contributions are summarized as follows:

1. We propose Structural Consistency, which measures the structural alignment between natural language and formal representations, revealing the shortcomings of existing models.

2. We develop a framework for combinatorics-specialized autoformalization, integrating manual data curation with a multi-stage refinement pipeline and a two-stage training strategy (expert iteration and RL alignment).

3. Experimental results show that *SAIR-Comb* achieves strong performance on combinatorial tasks while remaining highly competitive on general-domain benchmarks.

2 Related Work

Autoformalization. Automated formalization converts natural language descriptions into formal specifications (Weng et al., 2025) that can be verified by proof assistants such as Lean 4 (Moura and Ullrich, 2021). The limited scalability and generalization of rule-based systems necessitate a transition toward LLM-based approaches. Pioneering works like Herald (Gao et al., 2025) mitigate hallucination via retrieval augmentation from Mathlib. Kimina-Autoformalizer (Wang et al., 2025) and Goedel-Formalizer (Lin et al., 2025b) improve robustness through expert iteration and knowledge distillation. Parallel efforts such as ATF (Guo et al., 2025) enhance syntactic and semantic fidelity by incorporating tool feedback and consistency checks, while StepFun (Wu et al., 2026) aligns informal reasoning with formal logic. Within the realm of retrieval-based methods, recent advancements include RAutoformalizer (Liu et al., 2025b), which prioritizes dependency-aware augmentation, and ATLAS (Liu et al., 2025c), which exploits the structural properties of formal languages to scale data synthesis. Despite these advances, existing approaches often overlook the structural mapping between informal and formal domains, yielding valid generations that diverge from the original intent.

Evaluation Metrics for Autoformalization. The challenges of automatically evaluating natural language generation intensify as task complexity escalates. Symbolic approaches such as BEq (Liu et al., 2025b) attempt to verify semantic equivalence via a restricted set of Lean 4 tactics, their constrained reasoning capabilities often prove inadequate for complex, highly structured formalizations. There are also studies that explore evaluation via symbolic equivalence and semantic alignment (Li et al., 2024b). However, current evaluation pipelines predominantly adopt LLM-based paradigms. The mainstream workflow adopts a two-stage procedure: ensuring syntactic validity via the Lean compiler, followed by semantic check using the LLM-as-Judge (Gao et al., 2025; Lin et al., 2025b). Recent advancements have further refined this approach by employing multi-model voting mechanisms to mitigate evaluation bias (Guo et al., 2025) and validate the reliability of LLM-as-Judge (Chen et al., 2025). Despite these methodological improvements, existing pipelines primarily assess isolated correctness or broad semantic similarity, lacking a systematic verification of the fine-grained structural alignment between natural and formal specifications.

3 Structural Consistency

We conduct semantic consistency checks on the outputs of mainstream autoformalization models and perform an in-depth analysis of the generated Lean 4. Our analysis reveals that even semantically valid formalizations can exhibit significant structural misalignment with the original natural language descriptions, as detailed in Appendix E.

Figure 1 categorizes these structural inconsistencies through representative examples. Case 1 exhibits **structure collapse**, where structured operations are oversimplified into numerical or trivial expressions, leading to a loss of core information. Case 2 establishes a superficial framework but omits critical definitions via `sorry` placeholders.

Figure 1: An example of Structural Inconsistency.

Case 3 correctly defines the main objects but introduces redundant definitions (e.g., `countOneEven`), leading to definition redundancy. In contrast, the final case faithfully preserves both the semantic and structural integrity of the original proposition, defining all necessary objects and relations without redundancy or omission.

Based on the above inconsistencies, we propose **Structural Consistency** as a stricter evaluation metric. Beyond semantic correctness, a high-quality Lean 4 formalization must satisfy the following alignment dimensions:

1. **Structural Fidelity:** Faithfully model the problem's objects, actions, types, and other key elements using appropriate structures.

2. **Definition Completeness:** Provide complete and interrelated definitions without omissions, redundancies, or isolated components.

3. **Complexity Preservation:** Preserve inherent complexity by avoiding oversimplification of structurally rich problems into mere arithmetic or procedural expressions.

In Section 5.3, we adopt a majority-voting LLM-as-a-Judge mechanism, using rigorously designed prompts specifically for structural consistency to evaluate the alignment of model outputs.

4 The SAIR-Comb Framework

We introduce SAIR-Comb, a comprehensive framework designed for combinatorial autoformalization that systematically integrates dataset construction, iterative refinement, and specialized training to bridge the gap between informal and formal structures. As illustrated in Figure 2, it begins with Structured Combinatorics Dataset Construction to establish a high-quality foundation for initialization, followed by Structure-Aware Data Augmentation and Refinement—an iterative process designed to rectify syntactic, semantic, and structural discrepancies. Finally, we employ Two-Stage Reasoning Alignment Training, which integrates expert iteration with reinforcement learning to obtain a domain-specialized model capable of preserving deep structural fidelity. By tightly coupling data-driven refinement with reasoning alignment, SAIR-Comb transforms informal mathematical discourse into rigorous, structural-consistent formalizations.

4.1 Structured Combinatorics Dataset Construction

We select exercises from the classic textbook *Applied Combinatorics* (Tucker, 1994) as the initial source and choose 363 representative combinatorial problems covering several core topics, including counting methods, permutations and combinations, recurrence relations, inclusion-exclusion, generating functions, and graph-related problems. These problems are first manually written by human experts to ensure that each instance strictly satisfies both semantic and structural consistency requirements. We then adopt a human-LLM collaborative approach. Problems are first categorized by domain and principle according to chapters of the textbook, and then a complete reasoning sample is constructed for each problem, including both the mathematical reasoning and formalizing process. The mathematical reasoning clearly presents the logic and solution strategy of the problem while providing the precise answer and the formalizing process builds on the preceding components in a step-by-step chain-of-thought format.

This dataset directly serves as the data augmentation template for the SAIR-Comb framework, providing structured guidance for models to learn formalizing reasoning patterns in Combinatorics. The dataset distribution and specific examples are provided in Appendix B.

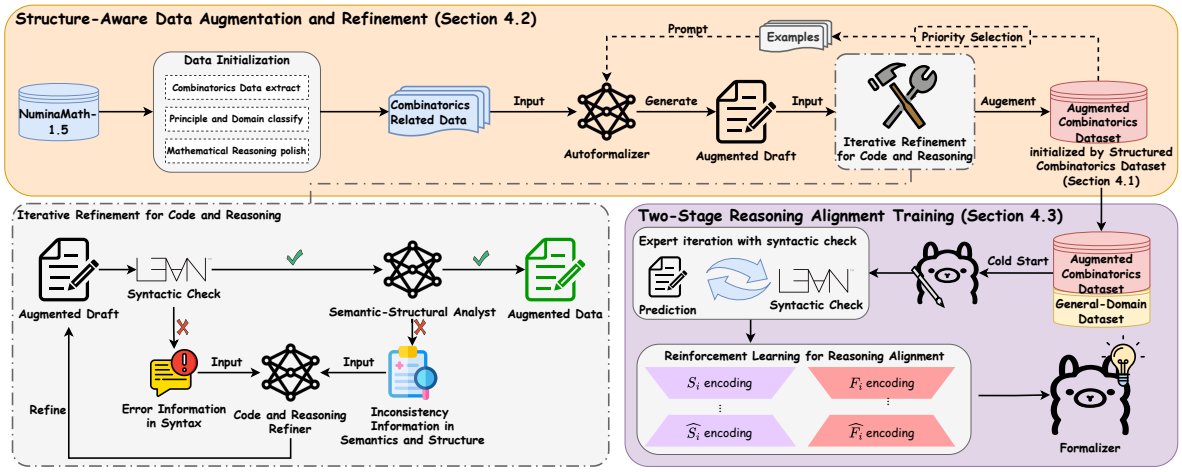

Figure 2: The SAIR-Comb Framework.

4.2 Structure-Aware Data Augmentation and Refinement

This component is driven by Lean 4 and LLMs, which is designed for syntactic, semantic and structural iterative refinement for data augmentation in Combinatorics. All LLMs we employ in this part are DeepSeek-V3.2 (DeepSeek-AI et al., 2025).

Data Initialization. Data for augmentation and refinement are sourced from NuminaMath-1.5 (Li et al., 2024a), which provides not only problem–answer pairs but also detailed problem categorizations and high-quality mathematical reasoning. We first extract combinatorics-related problems from NuminaMath-1.5 and filter out samples with incomplete reasoning or missing correct answers. Next, We use an LLM to assign domain and principle labels to the combinatorial problems, and systematically polish the corresponding mathematical reasoning processes to ensure that the Combinatorics related data closely aligns with the reasoning style of our handwritten dataset.

Draft Augmentation with Priority Selection. We employ an LLM as the autoformalizer. Its input includes the natural language statement, the corresponding domain, principle, and mathematical reasoning, along with reference examples from the Augmented Combinatorics Dataset. We prioritize enhanced examples matching the current problem's domain and principle and resort to random sampling from general domains when insufficient. We design a strict prompting template directing step-by-step formalizations aligned with the augmented data's structure, and apply a regular-expression-based filter to discard initial drafts that violate the required format.

Iterative Refinement for Code and Reasoning. This process consists of two stages. The first stage focuses on repairing potential syntactic errors, and the second stage addresses semantic-structural inconsistencies. SAIR-Comb not only corrects errors or inconsistencies in formalizations but also improves the associated thought process, maintaining alignment between logical reasoning and formal expressions.

We first extract the complete Lean 4 code from the initial draft and perform an iterative syntactic check. If errors are detected, we record the specific error messages along with their locations and perform targeted repair using a refiner LLM. Additionally, during the iterative process, we maintain a history of past errors and provide it as a reference. This mechanism prevents the model from repeatedly introducing the same errors during multiple repair attempts, and incorporating multiple error references into the repair reasoning helps improve the overall quality of the augmented data.

Drafts that pass the syntactic check are given to a Semantic-Structural Analyst LLM. Guided by the concept of semantic and structural consistency, we design crafted prompts to direct the model to perform consistency checks. If inconsistencies are detected, the model outputs specific descriptions. The refiner LLM then performs targeted refinements on Lean 4 code segments and the corresponding reasoning process based on this information. Figure 3 illustrates a case study of structure collapse.

Iteration continues until no issues remain, or the draft is discarded after exceeding the maximum iterations. This ensures that only structurally sound formalizations are retained. Successfully verified

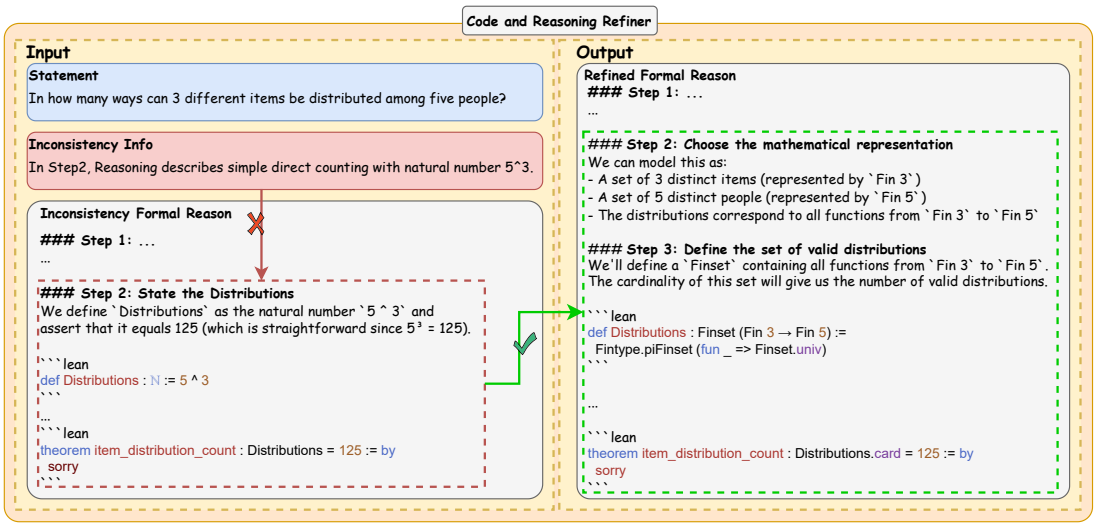

Figure 3: **Case study of structure collapse.** Code that should be structurally defined in `Distributions` is oversimplified as arithmetic expressions. The refiner LLM takes the natural language statement, the inconsistency information, and the corresponding formalizing process as input, then performs targeted refinement.

drafts are added to the Augmented Combinatorics Dataset to guide further augmentation.

4.3 Two-Stage Reasoning Alignment Training

To ensure the model performs well not only in Combinatorics but also remains competitive in general domains, we select 20k manually written samples from NuminaMath-Lean (Wang et al., 2025) and approximately 5.6k from FormalMath-Lean (Yu et al., 2025). We use Qwen3-32B (Yang et al., 2025) to generate the corresponding mathematical reasoning and formalizing process for these data and align their format with our Combinatorics dataset. Training hyperparameters and detailed dataset scales are provided in Appendix C.

Expert iteration with syntactic check. We perform a cold-start fine-tuning by combining our iteratively augmented Combinatorics dataset (about 3k samples) with the above 25.6k general-domain samples. During iteration, NuminaMath-1.5 serves as input, and model-generated drafts undergo a syntactic check. Drafts pass the check are added to the iteration training set. Since generated data contain rich reasoning processes, minor deviations can still provide effective learning signals, so we do not strictly enforce semantic or structural consistency checks. Over two iterations, the model generates around 35K entries in each round, bringing the total training data to approximately 100K. For specific training data sizes, see Appendix C.

Reinforcement Learning for Reasoning Alignment. We introduce reinforcement learning on top of expert iteration, using the augmented Combinatorics and FormalMath-Lean as the reinforcement corpus. Consider a generated output $(\hat{S}_i, \hat{F}_i, \hat{L}_i)$ and its corresponding ground truth (S_i, F_i, L_i), where S_i denotes mathematical reasoning, F_i the formalizing process and L_i the full Lean 4 code. Our reward function consists of three components:

- **Output-format reward** R_{format}: 1 if $\hat{S}_i$ and $\hat{F}_i$ are enclosed in <*think*>...</*think*> and followed by $\hat{L}_i$, 0 otherwise.
- **Syntactic reward** $R_{syntactic}$: 1 if $\hat{L}_i$ passes the syntactic check, 0 otherwise.
- **Reason-alignment reward** R_{reason}: the similarity between $(\hat{S}_i, \hat{F}_i)$ and (S_i, F_i), with values ranging from 0 to 1.

We use the state-of-the-art lightweight embedding model all-MiniLM-L6-v2 (Reimers and Gurevych, 2020) to embed $\hat{S}_i$, $\hat{F}_i$, S_i, and F_i into 384-dimensional vectors and compute their cosine similarities. The R_{reason} is defined as:

$$R_{reason} = \alpha \,\mathrm{sim}(f_{emb}(S_i), f_{emb}(\hat{S}_i)) + (1-\alpha)\,\mathrm{sim}(f_{emb}(F_i), f_{emb}(\hat{F}_i)), \quad (1)$$

where α is a hyperparameter in $[0, 1]$, $sim(x, y)$ denotes cosine similarity between x and y, and $f_{emb}(x)$ represents the embedding operation.

The total reward is taken as the minimum of the three individual rewards. We use CISPO (MiniMax et al., 2025) algorithm as it has been shown to perform well on long-text chain-of-thought tasks.

Table 1: **Model performance across benchmarks. Semantic:** percentage of syntactically correct code satisfying semantic consistency. **Structural:** percentage of semantically consistent code satisfying structural consistency.

Model	CombiBench		ProverBench		PutnamBench	
	Semantic	Structural	Semantic	Structural	Semantic	Structural
Pass@1						
Kimina-Autoformalizer-7B	4.88%	3.06%	42.67%	35.09%	17.15%	14.94%
Goedel-Formalizer-8B	**16.25%**	9.12%	**64.94%**	**54.99%**	**28.19%**	**21.46%**
StepFun-Formalizer-7B	10.00%	5.81%	38.97%	34.20%	14.34%	11.90%
ATF-8B	14.25%	8.19%	53.09%	43.75%	21.34%	17.07%
SAIR-Comb-8B (Ours)	14.19%	**9.81%**	48.85%	41.92%	16.81%	14.23%
Pass@8						
Kimina-Autoformalizer-7B	9.00%	7.00%	65.06%	57.42%	33.77%	29.17%
Goedel-Formalizer-8B	34.73%	25.61%	**86.24%**	**78.74%**	**54.31%**	**44.12%**
StepFun-Formalizer-7B	25.94%	20.52%	58.87%	53.77%	31.11%	25.92%
ATF-8B	32.08%	22.02%	75.23%	63.47%	44.42%	36.21%
SAIR-Comb-8B (Ours)	**40.33%**	**33.71%**	79.79%	72.51%	43.98%	38.01%
Pass@16						
Kimina-Autoformalizer-7B	11.00%	9.00%	68.39%	62.07%	38.18%	33.18%
Goedel-Formalizer-8B	41.00%	32.00%	**89.66%**	**82.76%**	**62.73%**	**51.52%**
StepFun-Formalizer-7B	32.00%	27.00%	64.37%	58.62%	37.58%	31.52%
ATF-8B	39.00%	28.00%	82.18%	68.39%	51.36%	42.58%
SAIR-Comb-8B (Ours)	**49.00%**	**42.00%**	86.21%	79.89%	53.79%	46.82%

5 Experiment

In this section, we present the experimental setup and validate the effectiveness of SAIR-Comb through the results of its trained models. The main experiments show that under high-sample settings, our model exhibits strong specialization and generalization, while ablation studies confirm the contributions of each training stage. Detailed training procedures are provided in Appendix C.

5.1 Benchmarks and Baselines

Benchmarks. We use CombiBench—the authoritative Combinatorics benchmark (Liu et al., 2025a)—to evaluate the formalization performance of our model in combinatorial problems. We then use ProverBench (Ren et al., 2025) and PutnamBench (Tsoukalas et al., 2024) to evaluate model performance on general-domain mathematical problems, where ProverBench mainly consists of algebraic tasks with lower semantic-structural requirements, while PutnamBench spans multiple domains and demands higher structural complexity. Notably, since the original ProverBench contains duplicated problems, which can bias the evaluation, we use the 13-gram decontaminated version introduced by StepFun (Wu et al., 2026).

Baselines. We select representative prior works as baselines, including Kimina-Autoformalizer-7B (Wang et al., 2025), Goedel-Formalizer-8B (Lin et al., 2025b), StepFun-Formalizer-7B (Wu et al., 2026), and ATF-8B (Guo et al., 2025), with their parameters configured to match the official implementations.

5.2 Metrics

We adopt the majority-voting LLM-as-Judge approach for evaluation, selecting the high-performing DeepSeek-V3.2 (DeepSeek-AI et al., 2025) and Qwen3-Max (Yang et al., 2025) (temperature 0.6) as parallel judges, and consider a case as passed only if both models agree. A generated code first undergoes a syntactic check. Code passing syntax is first evaluated for semantic consistency and further assessed for structural consistency if successful. We employ carefully designed, multi-dimensional prompts that reference both the original natural language description and the ground-truth formalization. Incorporating the ground-truth ensures stricter and more reliable judgments. Detailed prompt configurations are provided in Appendix G. Overall performance is reported using the unbiased *Pass@k* metric.

5.3 Main Results

Based on the main experimental results in Table 1, we draw several conclusions:

Strong performance on CombiBench. Although

Table 2: **Ablation results.** Pass@16 performance for Cold Start, Expert Iteration, and Expert Iteration + CISPO.

Model	CombiBench		ProverBench		PutnamBench	
	Semantic	Structural	Semantic	Structural	Semantic	Structural
Cold Start	39.00%	32.00%	82.76%	78.74%	46.67%	41.52%
+ Expert Iteration	47.00% (+8.00%)	39.00% (+7.00%)	86.78% (+4.02%)	79.31% (+0.57%)	51.97% (+5.30%)	44.09% (+2.57%)
+ CISPO	49.00% (+2.00%)	42.00% (+3.00%)	86.21% (-0.57%)	79.89% (+0.58%)	53.79% (+1.82%)	46.82% (+2.73%)

semantic consistency at Pass@1 is slightly lower than Goedel (-2.06%) and ATF (-0.06%), structural consistency reaches 9.81%, surpassing all baselines. As the number of samples increases, SAIR-Comb shows significant improvements, especially at Pass@16, where semantic and structural consistency reach 49.00% and 42.00%, exceeding the best baseline Goedel by 8.00% and 10.00%, respectively. The advantage over ATF is even more pronounced, demonstrating its strong capability in handling highly structured combinatorics tasks.

Competitive generalization to general domains. On general benchmarks, SAIR-Comb exhibits robust cross-domain performance. At Pass@1, SAIR-Comb achieves semantic and structural consistency of 48.85% and 41.92% on ProverBench and 16.81% and 14.23% on PutnamBench, placing it at a moderate level among baseline models. At Pass@8, SAIR-Comb reaches 79.79% and 72.51% on ProverBench, approaching Goedel, and 43.98% and 38.01% on PutnamBench, where structural consistency slightly exceeds ATF while semantic consistency remains slightly lower. At Pass@16, SAIR-Comb further improves to 86.21% and 79.89% on ProverBench and 53.79% and 46.82% on PutnamBench, remaining only marginally below Goedel while clearly outperforming other baselines. These results indicate that, although SAIR-Comb is primarily designed for structurally intensive combinatorial formalization, it maintains stable and competitive performance on general benchmarks under higher sampling budgets.

Structural Consistency reveals substantial performance drops across models. After applying structural consistency check, all models exhibit noticeable performance drops, especially models with stronger semantic performance, generally showing larger drops. At Pass@16, for instance, ATF decreases by 11.00% on CombiBench, 13.79% on ProverBench and 8.78% on PutnamBench. Goedel also shows clear declines of 9.00%, 6.90% and 11.21% on the same benchmarks. In contrast, our model exhibits smaller decreases of 7.00%, 6.32% and 6.97%, indicating stronger robustness in preserving structural consistency. These results indicate that evaluating structural consistency is essential and also underscore the inherent structural complexity of autoformalization tasks.

5.4 Ablation Studies

We conduct an ablation study on the two-stage training strategy. The results are shown in Table 2, from which we draw several conclusions:

Expert Iteration provides substantial improvements. Semantic and structural consistency increase by 8.00% and 7.00% on CombiBench. Both consistencies rise by 4.02% and 0.57% on ProverBench, and improve by 5.30% and 2.57% on PutnamBench. Averaged across the three benchmarks, semantic consistency grows by approximately 5.77% and structural consistency by approximately 3.38%. The relatively smaller gains on ProverBench are due to most problems having lower requirements for semantic and structural correctness. PutnamBench and CombiBench contain more combinatorial and general-domain highly-structured problems, resulting in more noticeable improvements. Analyzing the underlying factors, we find that the model already demonstrates strong initial formalization ability at cold start, and the iterative process of generating high-quality reasoning samples filtered through Lean 4 further enhances both data quality and model performance.

Reinforcement learning further improves performance. On CombiBench, the two consistency increase by 2.00% and 3.00%, showing clear improvements on highly-structured combinatorics tasks. On ProverBench, semantic slightly decreases by 0.57% while structural increases by 0.58%, indicating that CISPO has limited effect on benchmarks with low structural requirements. On PutnamBench, the consistencies rise by 1.82% and 2.73%, This indicates that CISPO remains effective on general-domain benchmarks where structural requirements exist. two metrics grow by 1.08% and 2.10% on average, demonstrating CISPO can further enhance the model's overall performance.

6 Further Analysis

Based on our experimental results, SAIR-Comb shows a clear improvement trend as Pass increases. At low Pass values, its performance is relatively weak, but as Pass rises, it catches up with other baseline models, surpassing all baselines on CombiBench.

Table 3: Syntactic pass rates under different Pass@k.

	Pass@1	Pass@8	Pass@16
CombiBench			
ATF-8B	**58.50%**	**84.82%**	88.00%
SAIR-Comb-8B (Ours)	38.12%	83.63%	**94.00%**
ProverBench			
ATF-8B	**86.82%**	94.94%	95.40%
SAIR-Comb-8B (Ours)	80.17%	**97.21%**	**98.28%**
PutnamBench			
ATF-8B	**61.39%**	84.60%	88.94%
SAIR-Comb-8B (Ours)	51.43%	**87.33%**	**93.03%**

We hypothesize that the performance gap at low Pass values is mainly due to SAIR-Comb's lower syntax success rate. For a fair comparison, we select ATF as the baseline, as its overall performance is comparable to that of SAIR-Comb. As shown in Table 3, at Pass@1, the syntax success rate of SAIR-Comb is noticeably lower than that of ATF; on ProverBench, which has low structural requirements, the difference is only 6.65%, but as the benchmarks' structural demands increase, the gap widens to 9.96% on PutnamBench and even reaches 20.38% on CombiBench, which has the highest structural requirements. This trend suggests that the model's tendency to generate highly-structured formalization increases the syntax difficulty in a single sampling.

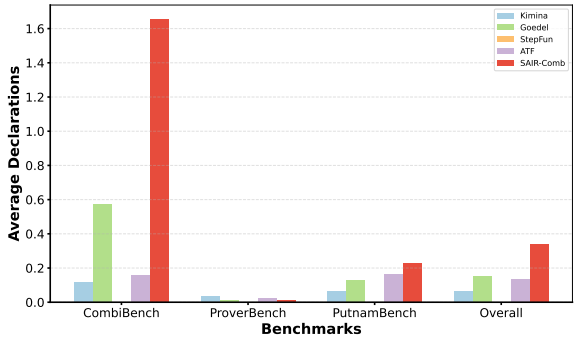

Figure 4: **Comparison of Structural Complexity.** We count the number of structures generated by all models across all samples and take the average, excluding `theorems` since each formalization necessarily contains a main theorem.

For further verification, we analyze the structural complexity of formalizations generated by all models. Specifically, we count all definitions except the main theorem (*def, structure, instance, abbrev, inductive, lemma*) and compute the average per output (as shown Figure 4). The results show that on CombiBench, SAIR-Comb generates an average of over 1.6 structural definitions per output, clearly ahead of other models; it also maintains a substantial lead on PutnamBench. On ProverBench, where structural requirements are low, the additional definitions of all models are nearly zero. This phenomenon is negatively correlated with syntactic success, supporting our hypothesis and also pointing to a direction for future formalization research: how to preserve the inherent syntactic performance of the model while ensuring semantic consistency and structural fidelity.

Notably, under high Pass settings, SAIR-Comb achieves superior syntactic success, consistent with the overall trend of improving semantic and structural consistency observed in the main experiments. This demonstrates that with increased sampling, the model not only maintains stable performance but also better leverages structural information, exhibiting stronger expressiveness even on structurally complex benchmarks.

7 Conclusions

We tackle the challenges of proposing SAIR-Comb, a structure-aware iterative refinement framework. The framework leverages a manually curated dataset of 363 high-quality formalizations to guide data augmentation and employs a two-stage training strategy—combining expert iteration with reinforcement learning-based reasoning alignment—to systematically train a specialized model in Combinatorics. We further introduce Structural Consistency as a stringent evaluation metric to assess the fidelity of formal expressions and capture discrepancies that semantic-only evaluations fail to detect. Experimental results on CombiBench show that our model achieves strong performance in both semantic and structural aspects, while on ProverBench and PutnamBench demonstrate that it remains highly competitive on general-domain formalization tasks. The observed performance drop under structural consistency evaluation highlights the necessity of this metric and exposes the limitations of existing methods in handling complex structures.

Limitations

Although our work substantially improves autoformalization in Combinatorics, several limitations remain. Due to the inherent randomness of model generation, SAIR-Comb may still produce semantic or structural inconsistencies in a small number of cases. In addition, despite employing multi-model evaluation, stricter structural consistency criteria, and ground-truth assistance, LLM-as-Judge evaluation may still suffer from certain biases. In future work, we plan to explore more rigorous data augmentation methods to construct larger-scale datasets and investigate more reliable and robust evaluation standards.

Ethical Considerations

In this work, we employ large language models for data augmentation, code generation, and reasoning verification. Models used include Qwen3-8B, Goedel-Formalizer-8B, Kimina-Autoformalizer-7B, StepFun-Formalizer-8B, ATF-8B, DeepSeek-V3.2, and Qwen3-Max. Similar to other language models, SAIR-Comb-8B may occasionally produce incorrect or hallucinated outputs. Our experiments are based on the open-source datasets NuminaMath-Lean, FormalMath, and NuminaMath, and evaluated using the benchmarks CombiBench, ProverBench, and PutnamBench. We strictly adhere to the licenses and usage policies associated with these resources and fully acknowledge their foundational contributions. Additionally, the manually constructed dataset introduced in this study has undergone thorough human review and de-identification to ensure the absence of personally identifiable information, sensitive content, and potential ethical risks. We also explicitly recognize the valuable contributions made by the domain experts involved in its curation.

Acknowledgement

This work was supported in part by the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDA0480501, in part by the National Key Research and Development Program of China under Grant 2023YFA1009402, in part by ECNU Multifunctional Platform for Innovation (001).

References

Guoxin Chen, Jing Wu, Xinjie Chen, Wayne Xin Zhao, Ruihua Song, Chengxi Li, Kai Fan, Dayiheng Liu, and Minpeng Liao. 2025. Reform: Reflective autoformalization with prospective bounded sequence optimization. *arXiv preprint arXiv:2510.24592*.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Ban Lin, Bing Xue, Bing-Li Wang, Bin Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, and 244 others. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Guoxiong Gao, Yutong Wang, Jiedong Jiang, Qi Gao, Zihan Qin, Tianyi Xu, and Bin Dong. 2025. Herald: A natural language annotated lean 4 dataset. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Qi Guo, Jianing Wang, Jianfei Zhang, Deyang Kong, Xiangzhou Huang, Xiangyu Xi, Wei Wang, Jingang Wang, Xunliang Cai, Shikun Zhang, and Wei Ye. 2025. Autoformalizer with tool feedback. *arXiv preprint arXiv:2510.06857*.

Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. 2024a. Numinamath. `https://huggingface.co/datasets/AI-MO/NuminaMath-1.5`.

Zenan Li, Yifan Wu, Zhaoyu Li, Xinming Wei, Fan Yang, Xian Zhang, and Xiaoxing Ma. 2024b. Autoformalize mathematical statements by symbolic equivalence and semantic consistency. *Advances in Neural Information Processing Systems*, 37:53598–53625.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. 2025a. Goedel-prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. 2025b. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*.

Junqi Liu, Xiaohan Lin, Jonas Bayer, Yael Dillies, Weijie Jiang, Xiaodan Liang, Roman Soletskyi, Haiming Wang, Yunzhou Xie, Beibei Xiong, Zhengfeng Yang, Jujian Zhang, Lihong Zhi, Jia Li, and Zhengying Liu.

2025a. Combibench: Benchmarking LLM capability for combinatorial mathematics. *arXiv preprint arXiv:2505.03171*.

Qi Liu, Xinhao Zheng, Xudong Lu, Qinxiang Cao, and Junchi Yan. 2025b. Rethinking and improving autoformalization: towards a faithful metric and a dependency retrieval-based approach. In *The Thirteenth International Conference on Learning Representations*.

Xiaoyang Liu, Kangjie Bao, Jiashuo Zhang, Yunqi Liu, Yu Chen, Yuntian Liu, Yang Jiao, and Tao Luo. 2025c. Atlas: Autoformalizing theorems through lifting, augmentation, and synthesis of data. *arXiv preprint arXiv:2502.05567*.

Wangyue Lu, Lun Du, Sirui Li, Ke Weng, Haozhe Sun, Hengyu Liu, Minghe Yu, Tiancheng Zhang, and Ge Yu. 2025. Automated formalization via conceptual retrieval-augmented llms. *arXiv preprint arXiv:2508.06931*.

Russell Merris. 2003. *Combinatorics*. John Wiley & Sons.

MiniMax, Aili Chen, Aonian Li, Bangwei Gong, Binyan Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, Chengjun Xiao, Chengyu Du, Chi Zhang, Chu Qiao, Chunhao Zhang, Chunhui Du, Congchao Guo, Da Chen, and 108 others. 2025. Minimax-m1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*.

Hayden Moore and Asfahan Shah. 2025. Evaluating autoformalization robustness via semantically similar paraphrasing. *arXiv preprint arXiv:2511.12784*.

Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pages 625–635, Cham. Springer International Publishing.

Auguste Poiroux, Gail Weiss, Viktor Kunčak, and Antoine Bosselut. 2025. Reliable evaluation and benchmarks for statement autoformalization. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 17958–17980.

Nils Reimers and Iryna Gurevych. 2020. Making monolingual sentence embeddings multilingual using knowledge distillation. *arXiv preprint arXiv:2004.09813*.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. 2025. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. 2024. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. *Advances in Neural Information Processing Systems*, 37:11545–11569.

Alan Tucker. 1994. *Applied combinatorics*. John Wiley & Sons, Inc.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, and 21 others. 2025. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*.

Ke Weng, Lun Du, Sirui Li, Wangyue Lu, Haozhe Sun, Hengyu Liu, and Tiancheng Zhang. 2025. Autoformalization in the era of large language models: A survey. *arXiv preprint arXiv:2505.23486*.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. *Advances in neural information processing systems*, 35:32353–32368.

Yutong Wu, Di Huang, Ruosi Wan, Yue Peng, Shijie Shang, Chenrui Cao, Lei Qi, Rui Zhang, Xishan Zhang, Zidong Du, Jie Yang, and Xing Hu. 2026. Stepfun-formalizer: Unlocking the autoformalization potential of llms through knowledge-reasoning fusion. In *Fortieth AAAI Conference on Artificial Intelligence, Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence, Sixteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2026, Singapore, January 20-27, 2026*, pages 33980–33988. AAAI Press.

Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Ming Ding. 2025. Bfs-prover: Scalable best-first tree search for llm-based automatic theorem proving. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32588–32599.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. 2024. Formal mathematical reasoning: A new frontier in ai. *arXiv preprint arXiv:2412.16075*.

Zhouliang Yu, Ruotian Peng, Keyi Ding, Yizhe Li, Zhongyuan Peng, Minghao Liu, Yifan Zhang, Zheng Yuan, Huajian Xin, Wenhao Huang, Yandong Wen, Ge Zhang, and Weiyang Liu. 2025. Formalmath: Benchmarking formal mathematical reasoning of large language models. *arXiv preprint arXiv:2505.02735*.

Lan Zhang, Marco Valentino, and Andre Freitas. 2025. Beyond gold standards: Epistemic ensemble of llm judges for formal mathematical reasoning. *arXiv preprint arXiv:2506.10903*.

Yaowei Zheng, Junting Lu, Shenzhi Wang, Zhangchi Feng, Dongdong Kuang, Yuwen Xiong, and Richong Zhang. 2025. Easyr1: An efficient, scalable, multimodality rl training framework. https://github.com/hiyouga/EasyR1.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 400–410. Association for Computational Linguistics.

A Reliability Analysis of the LLM-as-Judge Approach

To evaluate the reliability of the LLM-as-Judge paradigm, we compare its outputs against human annotations using standard metrics, including precision, recall, specificity, accuracy, and F1-score. To ensure fairness and representativeness while mitigating potential biases, we perform stratified sampling by selecting 10 samples that pass the consistency checks and 10 that fail the same criteria for each model on each benchmark, resulting in a total of 300 samples. To avoid potential ordering bias affecting the evaluation results, the samples are randomly shuffled and then evaluated by human experts. We treat human expert evaluations as the ground truth and define the evaluation metrics accordingly: a sample is considered a true positive when both the human expert and the LLM judge it as correct, and a true negative when both judge it as incorrect; the remaining cases are defined as false positives and false negatives, respectively.

As shown in Table 4, the LLM-as-Judge paradigm demonstrates strong agreement with human evaluation, achieving an accuracy of 94.33% and an F1-score of 94.24%. These results indicate that LLM can effectively approximate human judgment and serve as a reliable tool for large-scale automatic evaluation. Notably, the recall (95.86%) is slightly higher than the precision (92.67%), indicating a mild tendency toward permissive judgments. Overall, the method maintains a favorable balance between coverage and precision, validating the effectiveness and reliability of the LLM-as-Judge paradigm.

Table 4: LLM–Human agreement.

Metric	Value
Precision	92.67%
Recall	95.86%
Specificity	92.90%
Accuracy	94.33%
F_1-score	94.24%

B Strctured Combinatorics Dataset

This appendix details the composition of our experts-handwritten Strctured Combinatorics Dataset, which serves as high-quality initialization data for iterative augmentation and refinement.

Figure 5 shows the distribution of the 363 selected problems across major sections of the textbook Applied Combinatorics. The problems span core subfields, with notable concentrations appearing in **Graph Theory** (22 problems), **Covering Circuits and Graph Coloring** (16 problems), **Trees and Searching** (15 problems), **General Counting Methods for Arrangements and Selections** (160 problems), **Generating Functions** (62 problems), **Recurrence Relations** (45 problems) and **Inclusion-Exclusion** (43 problems). This coverage ensures the dataset reflects the structural diversity of combinatorics.

Table 5 reports the average and maximum numbers of structural declarations and lines of Lean 4 code for each chapter in our expert-curated dataset, highlighting the highly-structured and implementation complexity of the dataset.

Figure 6 provides a concrete example from the "General counting methods for arrangements and selections" domain (principle: Arrangements and Selections with Repetitions). The figure illustrates the original natural language statement with corresponding domain and principle, the mathematical reasoning and the complete step-by-step formalizing process in Lean 4. The key rea-

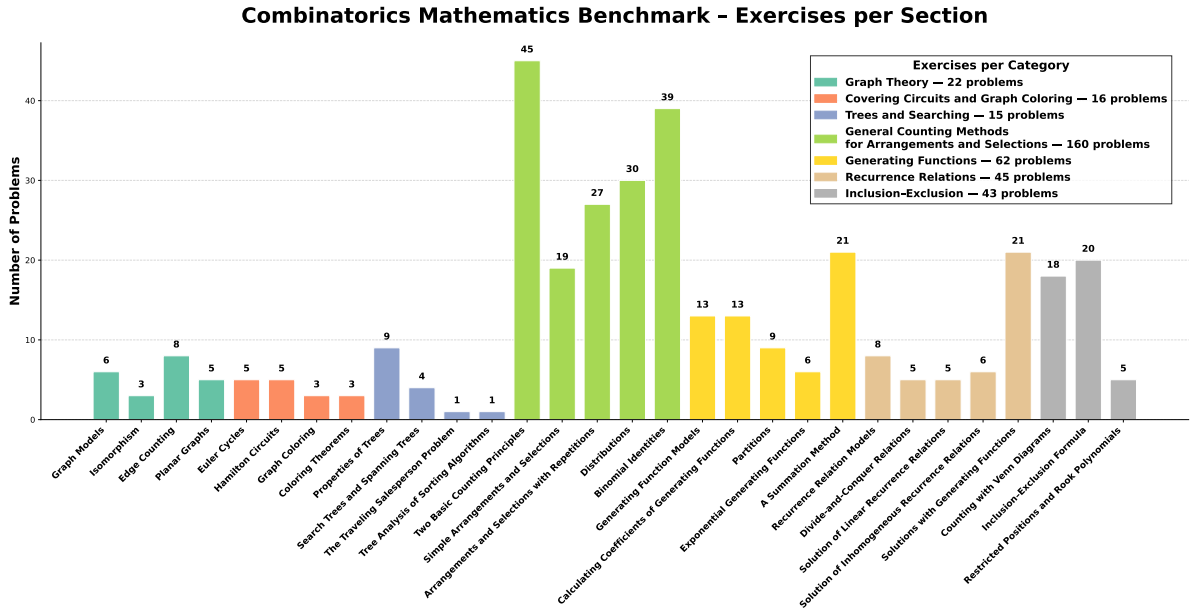

Figure 5: Distribution of problems in the Structured Combinatorics Dataset.

Statement

How many ways are there to pick a collection of nine coins from piles of pennies, nickels, dimes, and quarters?

Domain

General counting methods for arrangements and selections

Principle

Arrangements and Selections with Repetitions

Mathematical Reasoning

This problem asks for the number of ways to choose a collection of nine coins from four distinct types of coins (pennies, nickels, dimes, and quarters), where the order of selection does not matter and repetition is allowed (since we are picking from "piles" of coins). This is a classic problem of combinations with repetition, often solved using the "stars and bars" method.

Let x_1 be the number of pennies, x_2 be the number of nickels, x_3 be the number of dimes, and x_4 be the number of quarters. We want to find the number of non-negative integer solutions to the equation: $x_1 + x_2 + x_3 + x_4 = 9$

Here, $n = 9$ represents the total number of coins to be picked (the "stars"), and $k = 4$ represents the number of distinct coin types (the "bins" or categories).

The formula for combinations with repetition is given by $\binom{n+k-1}{k-1}$ or equivalently $\binom{n+k-1}{n}$. Using $n = 9$ and $k = 4$: Number of ways = $\binom{9+4-1}{4-1} = \binom{12}{3}$ Alternatively, Number of ways = $\binom{9+4-1}{9} = \binom{12}{9}$

Formalization Steps

Step 1: Define the CoinType inductive type.

The problem specifies four types of coins: pennies, nickels, dimes, and quarters. In Lean, these distinct types are represented using an **inductive** type. The **deriving DecidableEq, Fintype** clause automatically generates necessary instances for equality comparison and to treat **CoinType** as a finite set, which is crucial for later operations like **Fintype.piFinset**.

```
inductive CoinType
 | penny | nickel | dime | quarter
 deriving DecidableEq, Fintype
```

Step 2: Define coinDistributions as a Finset.

This definition constructs the set of all possible ways to pick nine coins. Each "way" is a function **f : CoinType → ℕ**, where **f t** gives the count of coins of type **t**.

* **Fintype.piFinset (fun _ : CoinType => Finset.range 10)**: This part generates a **Finset** of all functions **f** from **CoinType** to ℕ such that for each **t : CoinType**, **f t** is an element of **Finset.range 10**. **Finset.range 10** represents the set **{0, 1, ..., 9}**. This means that for each coin type, the number of coins chosen can be any integer from 0 up to 9. This upper bound of 9 is sufficient because the total number of coins is 9, so no single type can exceed 9.
* **.filter (fun f => ∑ t : CoinType, f t = 9)**: This filters the previously generated set of functions. It keeps only those functions **f** for which the sum of the counts of all coin types (**∑ t : CoinType, f t**) is exactly equal to 9. This precisely captures the condition of picking a total of nine coins.

```
def coinDistributions : Finset (CoinType → ℕ) :=
  (Fintype.piFinset (fun _ : CoinType => Finset.range 10)).filter
    (fun f => ∑ t : CoinType, f t = 9)
```

Step 3: State the theorem

This theorem asserts that the number of elements in the **coinDistributions Finset** (cardinality) is equal to the result of the stars and bars formula.

```
theorem ch5_3_5 :
  coinDistributions.card = (9 + 4 - 1 : ℕ).choose 9 := by
sorry
```

Figure 6: An example problem from the Structured Combinatorics Dataset.

soning includes: 1) Mathematical reasoning and the correct numerical answer; 2) Definition of a **`CoinType`** for the four coin types; 3) Construction of **`coinDistributions`** to model all possible ways to pick nine coins, with precise filters ensuring exact counts for each coin type.

This structural format—encompassing domain, principle, detailed mathematical reasoning, and compilable Lean 4 code—is applied uniformly across 363 human-authored samples.

C Training Detail

We use Qwen3-8B (Yang et al., 2025) as the base model and train it in two stages to achieve robust performance in autoformalization, particularly emphasizing structural fidelity in Combinatorics.

C.1 Supervised Fine-Tuning Stage

We first perform full-parameter supervised fine-tuning combined with expert iteration using LLaMA-Factory (Zheng et al., 2024). The training

Table 5: Chapter-wise statistics of structure numbers and Lean 4 lines of code.

Chapter	Avg. Structures	Max Structures	Avg. LOC	Max LOC
Graph Theory	2.50	10	21.59	39
Covering Circuits and Graph Coloring	2.25	5	21.69	53
Trees and Searching	2.73	8	23.00	59
General Counting Methods for Arrangements and Selections	4.24	**17**	24.68	68
Generating Functions	3.34	11	22.68	**88**
Recurrence Relations	**4.60**	12	**29.47**	74
Inclusion Exclusion	3.07	10	22.44	58

Hyperparameters for SFT

```
stage: sft
do_train: true
finetuning_type: full
cutoff_len: 16384
preprocessing_num_workers: 16
dataloader_num_workers: 0
plot_loss: true
enable_thinking: true
max_grad_norm: 1.0
per_device_train_batch_size: 2
gradient_accumulation_steps: 4
learning_rate: 5.0e-6
num_train_epochs: 3.0
lr_scheduler_type: cosine
optim: adamw_torch
warmup_ratio: 0.1
bf16: true
flash_attn: auto
val_size: 0.1
per_device_eval_batch_size: 2
eval_strategy: steps
eval_steps: 200
```

Figure 7: Training Hyperparameters for SFT.

Hyperparameters for RL

```
data:
  max_prompt_length: 2048
  max_response_length: 16384
  rollout_batch_size: 256
  mini_rollout_batch_size: null
  val_batch_size: 128
  override_chat_template: null
  shuffle: true
algorithm:
  adv_estimator: grpo
  disable_kl: false
  use_kl_loss: true
  kl_penalty: low_var_kl
  kl_coef: 1.0e-2
  filter_low: 0.01
  filter_high: 0.99
trainer:
  total_epochs: 3
  nnodes: 1
  n_gpus_per_node: 4
  max_try_make_batch: 5
  val_freq: 5
  val_before_train: true
  val_only: false
worker:
  actor:
    loss_type: cispo
    clip_ratio_low: 0.5
    clip_ratio_high: 10
    global_batch_size: 128
    max_grad_norm: 1.0
    optim:
      lr: 3.0e-6
      weight_decay: 1.0e-2
      strategy: adamw
      lr_warmup_ratio: 0.2
  rollout:
    n: 8
    temperature: 1.0
    top_p: 1.0
    limit_images: 0
    max_num_batched_tokens : 18432
    gpu_memory_utilization: 0.8
    tensor_parallel_size: 2
    val_override_config:
      temperature: 0.9
      top_p: 0.95
      n: 1
```

Figure 8: Training Hyperparameters for RL.

runs for 3 epochs with a cosine learning rate scheduler and an initial learning rate of $5 \cdot 10^{-6}$. We set a cutoff length of 16384 tokens. We employ mixed-precision training (bf16), AdamW-Torch optimizer, gradient accumulation over 4 steps and a per-device batch size of 2. Thinking mode is explicitly enabled during training to encourage mathematical reasoning and step-by-step formalizing process. More details are provided in Figure 7.

Table 6: Scale of the Training Dataset.

Dataset	Sizes
Combinatorics (Ours)	3,015
Numina-Lean	20,078
FormalMath-Lean	5,560
Iterative Dataset	71,658
Overall	**100,311**

C.2 Reinforcement Learning Stage

We optimize the SFT checkpoint via reinforcement learning using EasyR1 (Sheng et al., 2024; Zheng et al., 2025). Training proceeds for 3 epochs with a maximum prompt length of 2048 tokens and response length of 16384 tokens, a rollout batch size of 256. We apply a KL penalty (coefficient $0.01 \cdot 10^{-2}$), gradient clipping, and global batch normalization. Sampling uses temperature 1.0 and top-p 1.0 during rollouts. The complete hyperparameters are listed in Figure 8. The hyperparameter of reward function is set to $\alpha = 0.4$, encouraging the model to achieve higher alignment during the formalizing process.

C.3 Scale of the Training Dataset.

Table 6 presents the scale of our training dataset, reporting the number of problems from each data source. Specifically, "Combinatorics" corresponds to the high-quality Combinatorics problems augmented using our framework.

D A Case Study of Model Output

This section presents an output example of SAIR-Comb-8B in Figure 9. Given a natural language statement, the model generates the mathematical reasoning process, which includes problem analysis and calculation to obtain the precise answer and

identify the problem's domain and relevant principles. It then uses these as a reference to produce a step-by-step Lean 4 formalizing process. At the end it outputs the complete Lean 4 representation.

E Case Studies of Structural Inconsistency

This section presents specific cases of structural inconsistencies encountered in our experiments. Although these Lean 4 representations pass semantic consistency checks, they still exhibit structural deviations. For each case, we provide the corresponding natural language statement, the inconsistent output, and the explanation for its structural inconsistency, as shown in Figure 10.

F Prompt in SAIR-Comb Framework

This appendix presents the prompt designs used in the SAIR-Comb framework. Specifically, we include the prompt for autoformalization (Figure 11), the prompt for syntactic errors repair (Figure 12), the prompt for semantic and structural consistency check (Figure 13) and the prompt for semantic and structural inconsistency refinement (Figure 14). In each Figure, the input part of the prompt is highlighted in red. We provide a detailed explanation of these inputs below:
few_shot_list: A list of K examples sampled from the Augmented Combinatorics Dataset.
statement: As illustrated in Figure 6 (top-left), the natural language statement of the problem.
math: As shown in Figure 6 (bottom-left), the mathematical reasoning process for this problem.
domain: As shown in Figure 6 (top-left), the domain of this problem.
principle: As shown in Figure 6 (top-left), the principle of this problem.
lean4process: As shown in Figure 6 (right), the complete Lean 4 formalizing process.
lean4code: The complete Lean 4 representation.
current_error/history_error: The current syntactic errors in Lean 4, as well as errors encountered in previous iterations of the repair process.
explanation: A detailed explanation of semantic and structural inconsistencies.

G Prompt in evaluation

The LLM-as-Judge approach typically relies solely on the natural language statement as reference (Guo et al., 2025), or converts the predicted Lean 4 back into natural language and compares the two natural language representations (Gao et al., 2025; Lu et al., 2025). However, using only the natural language statement may result in overly lenient judgments when evaluating semantic and structural consistency (Moore and Shah, 2025). Incorporating the ground-truth formalization makes the evaluation more rigorous and aligned with the intended formal semantics. Since this setting explicitly involves the Lean 4 ground-truth representation, our prompt design cannot be directly adopted from existing evaluation templates. Instead, we develop customized prompts that integrate information from both the natural language statement and the Lean 4 ground-truth representation, enabling joint assessment of semantic and structural consistency.

This appendix presents the prompts used in our majority-voting LLM-as-a-Judge evaluation pipeline. Figure 15 shows the prompt for assessing semantic consistency, while Figure 16 shows the prompt for assessing structural consistency.

Case study of an output from SAIR-Comb

Input:

statement : How many ways are there to select 6 students from a class of 13 to sit in the front row?

Output:

 # Mathematical Reasoning:

The problem asks for the number of ways to select a group of 6 students from a class of 13 to sit in the front row.
This is a combination problem, as the order in which the students are selected does not matter.
The number of ways to choose k items from a set of n items without regard to order is given by the binomial coefficient:
$\binom{n}{k} = \frac{n!}{k!(n-k)!}$
Substituting n = 13 and k = 6:
$\binom{13}{6} = \frac{13!}{6!(13-6)!} = \frac{13!}{6! \cdot 7!}$
Calculating this:
$\binom{13}{6} = \frac{13 \times 12 \times 11 \times 10 \times 9 \times 8}{6 \times 5 \times 4 \times 3 \times 2 \times 1} = 1716$
Thus, there are 1716 ways to select the group.
The domain of the problem is GENERAL COUNTING METHODS FOR ARRANGEMENTS AND SELECTIONS.
The principle used is Simple Arrangements and Selections.

 # Formalization Steps:

Step 1: Define the type for students

We need to represent the 13 students in the class. Since 13 is small,
we can use `Fin 13` to represent them as finite natural numbers from 0 to 12.

```lean
abbrev Student : Type := Fin 13
```

Step 2: Define the set of all possible groups

A group of 6 students is a subset of the 13 students. In Lean, we can represent this as a
`Finset Student` with cardinality 6. We'll define the collection of all such groups.

```lean
def possibleGroups : Finset (Finset Student) :=
  (Finset.univ : Finset Student).powerset.filter (fun s => s.card = 6)
```

Step 3: Formalize the theorem

Mathematical Reasoning shows that the answer is 1716.
Since we're counting the cardinality of a finite set, we can use `Fintype.card` on the `possibleGroups` definition.

```lean
theorem formalizer_synthetic_theorem : Fintype.card possibleGroups = 1716 := by
  sorry
```

Complete Lean 4 Code

```lean
import Mathlib

abbrev Student : Type := Fin 13

def possibleGroups : Finset (Finset Student) :=
  (Finset.univ : Finset Student).powerset.filter (fun s => s.card = 6)

theorem formalizer_synthetic_theorem : Fintype.card possibleGroups = 1716 := by
  sorry
```

Figure 9: A Case Study of Model Output.

More Case Studies of Structural Inconsistency

Kimina-Autoformalizer:

Statement:
How many ways can a teacher select a group of 6 students to sit in the front row if the class has 13 students? (CombiBench)
Output:
```
import Mathlib
theorem hackmath_1 :
  (Nat.choose 13 6) = 13 * 12 * 11 * 10 * 9 * 8 / (6 * 5 * 4 * 3 * 2 * 1) := by sorry
```
Inconsistency explanation:
The structured definition degenerates into a simple calculation.

Statement:
Use the pigeonhole principle to prove that a graph of order n ≥ 2 always has two vertices of the same degree. (CombiBench)
Output:
```
import Mathlib

open Finset

theorem brualdi_ch11_5 {n : ℕ} [NeZero n]
    (hn : 2 ≤ n) (a : Fin n → Fin n → Prop) (ha : ∀ i j, a i j = a j i) (ha' : ∀ i, a i i = False) :
    ∃ i j, i ≠ j ∧ ( { k | a k j }.ncard = { k | a k i }.ncard ):= by sorry
```
Inconsistency explanation:
The formalization declares the existence of two vertices with equal degrees
but does not faithfully model the original problem's structure, as it lacks an explicit graph type

Goedel-Formalizer:

Statement:
Prove that D_n is an even number if and only if n is an odd number. (CombiBench)
Output:
```
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat

def derangement_count (n : ℕ) : ℕ := sorry

theorem brualdi_ch6_21 (n : ℕ) : Even (derangement_count n) ↔ Odd n := by sorry
```
Inconsistency explanation:
The output is inconsistent because derangement_count is left as sorry

Statement:
Let p be an odd prime and let $\mathbb{Z}_p$ denote (the field of) integers modulo p.
How many elements are in the set $\{x^2 : x \in \mathbb{Z}_p\} \cap \{y^2 + 1 : y \in \mathbb{Z}_p\}$? (PutnamBench)
Output:
```
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat

theorem putnam_1991_b5 (p : ℕ) [Fact (Nat.Prime p)] (hp_odd : Odd p) :
  let S1 := {z : ZMod p | ∃ x : ZMod p, z = x^2}
  let S2 := {z : ZMod p | ∃ y : ZMod p, z = y^2 + 1}
  (S1 ∩ S2).ncard = sorry := by
  sorry
```
Inconsistency explanation:
The output is inconsistent because the answer is missing.

stepFun-Formalizer:

Statement:
A ferry with a capacity of 10 people takes a group of 13 men and 7 women across a river.
Find the number of ways in which the group may be taken across the if all women go on the first group. (CombiBench)
Output:

```
import Mathlib
import Aesop

set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat


theorem hackmath_8 : (Nat.choose 13 3) = 286 := by
  sorry
```

Inconsistency explanation:
The structured definition degenerates into a simple calculation.

ATF:

Statement:
A palindrome in base b is a positive integer whose base-b digits read the same backwards and forwards; for example, 2002 is a 4-digit palindrome in base 10. Note that 200 is not a palindrome in base 10, but it is the 3-digit palindrome 242 in base 9, and 404 in base 7. Prove that there is an integer which is a 3-digit palindrome in base b for at least 2002 different values of b. (PutnamBench)
Output:

```
import Mathlib
import Aesop

def is_palindrome_in_base (n b : ℕ) : Prop :=
  n > 0 ∧ b > 1 ∧
  ∃ (digits : List ℕ), digits.length ≥ 1 ∧
    (∀ d ∈ digits, d < b) ∧
    digits.head! ≠ 0 ∧
    digits = digits.reverse ∧
    n = (digits.mapIdx (fun i d => d * b^i)).sum

def is_3_digit_palindrome_in_base (n b : ℕ) : Prop :=
  n > 0 ∧ b > 1 ∧
  ∃ a c : ℕ, a < b ∧ c < b ∧ a ≠ 0 ∧
    n = a * b^2 + c * b + a

theorem putnam_2002_b5 : ∃ n : ℕ,
  ∃ S : Finset ℕ, S.card ≥ 2002 ∧
    ∀ b ∈ S, is_3_digit_palindrome_in_base n b := by sorry
```

Inconsistency explanation:
The output is inconsistent because is_palindrome_in_base is defined but unused, causing structural redundancy.

Statement:
Let n and k be positive integers. Say that a permutation σ of $\{1, 2, \dot{s}, n\}$ is k-limited if $|\sigma(i) - i| \leq k$ for all i. Prove that the number of k-limited permutations $\{1, 2, \dot{s}, n\}$ is odd if and only if $n \equiv 0$ or $1 (\mathrm{mod}\, 2k + 1)$. (PutnamBench)
Output:

```
import Mathlib
import Aesop

def k_limited (n k : ℕ) (σ : Equiv.Perm (Fin n)) : Prop :=
  ∀ i : Fin n, Int.natAbs ((σ i).val + 1 - (i.val + 1)) ≤ k

instance : DecidablePred (fun σ => k_limited n k σ) := by sorry

theorem my_favorite_theorem (n k : ℕ) (hn : 0 < n) (hk : 0 < k) :
  Odd (Finset.card (Finset.filter (fun σ => k_limited n k σ)
  (Finset.univ : Finset (Equiv.Perm (Fin n))))) ↔ (n % (2 * k + 1) = 0 ∨ n % (2 * k + 1) = 1) := by sorry
```

Inconsistency explanation:
The issue is that the instance is defined but not proved, resulting in a missing structural component.

Figure 10: Case studies of Structural Inconsistency.

Prompt for autoformalization

I will give you a mathematical problem statement along with its reasoning process, domain, and principle.
Please think step-by-step and proceed incrementally, ultimately completing the automatic formalization of this problem.
Please refer to these examples for the automated formalization work:

{few_shot_list}

CRITICAL REMINDER - THIS IS NON-NEGOTIABLE:

1. Your SOLE responsibility is automated formalization.
2. You MUST NOT attempt to prove or solve the problem.
3. You MUST use `sorry` to complete any proof in theorems
   (Only *`sorry` in theorem* but you have to prove in `def`, `structure` and `instance` if needed).
4. When performing the formalization process, you must strictly follow the format given in the examples,
   using ### Step1, ### Step2, etc. (Generally, please keep the steps to around 5),
   for each step of the reasoning. Do not deviate from this step-by-step format under any circumstances.
5. Please note: when you have finished the entire formalization, you must paste the complete
   Lean 4 code block at the end, enclosed in a code fence labeled ```lean``` (like examples above).
6. Once you output the final ```lean``` code block, your formalization reasoning is considered complete.
   Do not output anything after the final code block.

Additional Requirement:
You have to define all necessary structures, defs,
   and instances to ensure semantic and structural consistency with the problem statement.
Do not define any unnecessary or unrelated structure or def.
All defined elements must be directly related to the problem statement and theorem.
Follow the example style closely to maintain semantic and structural consistency and relevance.

This is the problem to be formalized:

Statement: {statement}
Mathematical Reasoning: {math}
Domain: {domain}
Principle: {principle}

Now, please formalize the following problem and output your full answer without any other additional content.

Figure 11: Prompt for autoformalization.

Prompt for syntactic error repair

You are given:
1. A Lean4 formalization reasoning process (describing the intended mathematical reasoning).
2. The corresponding Lean4 code (which may contain compilation errors).
3. The compilation error messages produced by the code.

Your task is to **analyze the compilation errors, correct the Lean4 code,
   and produce a revised Lean4 formalization reasoning process**.

Requirements for the corrected process:

1. Keep the **original stepwise style** (`### Step 1: ...`, `### Step 2: ...`, etc) and and make sure there is at least one step.
2. Update all existing ```lean``` code blocks to reflect the corrections.
3. Ensure the **last ```lean``` block** contains the **full, compilable Lean4 code**.
4. Integrate lessons from the compilation errors into the reasoning explanation to avoid repeating them,
   without simply copying the old errors.
5. Present the reasoning in a clear, structured, and reliable manner that accounts for typical pitfalls,
   so that the formalization is sound and consistent.
6. Output **only the new, corrected Lean4 formalization reasoning process**.

Here is the Lean4 formalization reasoning process:

{lean4process}

Here is the corresponding Lean4 code:

{lean4code}

Here are the compilation error messages:

Here are the current Lean4 compilation errors:

{current_error}

Here are some previous compilation error messages (for reference to avoid repeating the mistakes):

{history_error}

Figure 12: Prompt for syntactic error repair.

Prompt for semantic and structural consistency check

You are an expert evaluator for assessing whether a generated Lean 4 theorem statement
(with proof omitted, e.g., using `sorry`) faithfully formalizes a given natural language mathematical statement.
Your task is to judge if the Lean 4 code is semantically and structurally consistent with the natural language statement.
Do NOT evaluate mathematical correctness of the theorem's conclusion (RHS), proof feasibility, or code compilation.

Input:
- Natural language statement: The original mathematical problem statement.
- Mathematical reasoning: Human-provided solution or answer
(DO NOT use this to infer or correct the intended meaning of the statement).
- Lean 4 formalization process: The reasoning trace leading to the code.
- Lean 4 code: The generated theorem statement (proof part must be ignored).

Rules:

1. Semantic Consistency
- Semantic Alignment: The Lean 4 theorem must express exactly the same mathematical claim
as the natural language statement.
- Logical Alignment: The logical structure, including quantifier types (∀/∃),
order, nesting, binding scope, assumptions, dependencies,
and conditions, must precisely match the intended meaning.
- Type and Constraint Alignment: All variables, types, explicit constraints, and implicit constraints
(e.g., non-negativity, distinctness, domain restrictions implied by the statement) must be accurately represented.
- Do not add, omit, or reinterpret elements not explicitly or implicitly required by the statement.

2. Structural Consistency
- Structural Preservation: The formalization must preserve the internal structure of the problem
(entities, relations, processes, scenario-specific rules) rather than
collapsing it into an unstructured or purely computational expression.
- Explicit Representation: All entities, actions, relations, and constraints mentioned or implied
in the statement must be explicitly modeled in Lean 4 (e.g., via `structure`, `def`, inductive types, or typed relations).
Implicit or omitted essential components constitute structural inconsistency.
- Unused Definitions: If any `def`, `structure`, or auxiliary object is defined but not used in the `theorem` statement,
judge as structurally inconsistent.
- Avoiding Structural Collapse: Do not reduce complex combinatorial, relational, or scenario-based problems
to trivial arithmetic or closed-form expressions (e.g., direct binomial coefficients, factorials, or numeric identities),
even if numerically equivalent.
The theorem must reflect the structured process or selection described in the statement.
- Simple Arithmetic Collapse: If defined objects or data structures exist
but the theorem only performs basic arithmetic operations (addition, subtraction, multiplication, division)
without meaningfully using those objects, judge as structurally inconsistent.

3. Natural Number Semantics in Lean 4
- Account for truncation in $\mathbb{N}$ operations (e.g., 0 - 2 = 0, 1 / 2 = 0). If such behavior could alter the intended semantics,
note it in the explanation and suggest using $\mathbb{R}$, $\mathbb{Q}$, or $\mathbb{Z}$ instead.

4. Evaluation Process
- Perform step-by-step analysis covering semantic and structural aspects.
- Provide detailed reasoning in <explanation>...</explanation> tags.
- Final judgment in <output>Yes</output> (fully consistent) or <output>No</output> (any inconsistency found).

Natural language statement:

{statement}

Mathematical reasoning:

{math}

Reasoning process of lean4 formalization:

{lean4process}

Please analyze step by step, explain your judgment clearly in <explanation></explanation>,
then provide <output>Yes/No</output>.

Figure 13: Prompt for semantic and structural consistency check.

Prompt for semantic and structural inconsistency refinement

You are an expert Lean 4 formalizer. You will be provided with a previous Lean 4 formalization reasoning process that contains semantic or structural inconsistencies with the natural language statement and mathematical reasoning. Your task is to produce a corrected Lean 4 formalization reasoning process that fully resolves these inconsistencies while preserving the original mathematical intent.

Inputs:
1. Natural language statement: The original mathematical problem.
2. Mathematical reasoning process: The correct human solution or reasoning.
3. Original Lean 4 formalization process: The flawed stepwise reasoning and code.
4. Explanation of inconsistencies: Detailed description of semantic and/or structural issues in the original formalization.

Requirements for the corrected Lean 4 formalization process:

1. Semantic Alignment: The final theorem statement must precisely capture the meaning of the natural language statement and align with the provided mathematical reasoning.
2. Structural Alignment: Entities, actions, relations, and constraints described or implied in the statement must be explicitly and appropriately modeled (e.g., via `structure`, `def`, inductive types, or typed relations).
3. Style Preservation: Use the same stepwise format with headers like `### Step 1: ...`, `### Step 2: ...`, etc. Include at least one step. Insert exactly one blank line (two line breaks) between each header and its content.
4. Code Updates: All existing ```lean``` code blocks must be updated to reflect the corrections. You may add, remove, or merge steps as needed for clarity and correctness.
5. Theorem Placeholder: The final `theorem` must include no proof and must use `by sorry` as placeholder.
6. Complete Compilable Code: The last ```lean``` block must contain the full, self-contained Lean 4 code (including necessary imports) that would compile if pasted into a Lean file.
7. Meaningful Definitions: All defined `def`, `structure`, or auxiliary objects must be meaningfully used in the theorem; remove any unused or irrelevant definitions.
8. Numerical Semantics: Account for truncation behavior in $\mathbb{N}$ operations (e.g., subtraction or division yielding 0). Use $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, or other appropriate types when truncation would alter the intended meaning.
9. Clear Explanations: Include necessary explanations within steps (e.g., type choices, required imports, potential pitfalls).
10. Mathematical Fidelity: Do not introduce any change to the intended mathematical claim.
11. Output Format: Output ONLY the corrected Lean 4 formalization reasoning process (stepwise reasoning with code blocks). Do not include any additional text, introductions, summaries, or explanations outside the steps.

Use the provided explanation of inconsistencies as primary guidance to avoid repeating the same errors.
The resulting formalization must be clear, robust, and fully consistent with the natural language statement.

Natural language statement:

{statement}

Mathematical reasoning process:

{math}

Lean4 formalization process:

{lean4process}

Explanation of inconsistencies:

{explanation}

Figure 14: Prompt for semantic and structural consistency refinement.

Prompt for judging semantic consistency

Role: Lean4 & Formal Verification Expert
You are given:
(1) a mathematical statement,
(2) a piece of Lean4 code generated by an expert model (without proof) that **compiles successfully**, and
(3) a ground-truth Lean4 formalization that correctly represents the statement. (As a reference)

Your task is to evaluate whether the generated Lean4 code is semantically consistent with the mathematical statement, using the ground-truth code as a reference.

Evaluation Criteria (Semantic Consistency):

- Semantic Alignment: Check whether the generated Lean4 code expresses the same mathematical claim as the natural-language statement.

- Logical Alignment: Verify that the logical structure—such as quantifiers, assumptions, dependencies, and conditions—matches the intended meaning of the statement.

- Answer/Target Alignment: Ensure that the code formalizes the correct numerical answer, conclusion, or output specified by the statement. The correct target must be validated by referencing the ground-truth Lean4 code **if it has**.

- Type/Constraint Alignment: Confirm that the types and constraints used in the code match those required by the mathematical statement.

Use the ground-truth formalization **as a reference**; it is not the gold standard.
Even if the generated code is very different from the ground-truth, it may still be correct — judge semantic faithfulness by the original statement itself.
Provide a clear and concise explanation, then indicate whether the generated Lean4 code is semantically consistent.

Output Format:

<explanation>Your explanation here</explanation>
<answer>(Yes|No)</answer>

Here are the mathematical statement:

{statement}

Here is the generated Lean4 code:

{lean4code}

Here is the ground-truth Lean4 code:

{lean4code_gt}

Figure 15: Prompt for semantic consistency.

Prompt for judging structural consistency

Role: Lean4 & Formal Verification Expert
You are given:
(1) a mathematical statement,
(2) a piece of Lean4 code generated by an expert model (without proof) that
  compiles successfully and has passed semantic checks, and
(3) a ground-truth Lean4 formalization that correctly represents the statement. (As a reference)
Your task is to evaluate whether the generated Lean4 code is structurally consistent
  with the mathematical statement, using the ground-truth code as a reference.

Evaluation Criteria (Structural Consistency):
- Structural Similarity: Assess whether the generated Lean4 code is properly organized and structured,
  reflecting the entities, actions, constraints, and relationships described in the natural-language statement.
  The formalization should preserve the internal structure of the problem
  (e.g., interactions, processes, or scenario-specific rules), rather than collapsing
  it into an unstructured or purely numerical expression.
- Explicit Definitions: For applied or scenario-based problems,
  ensure that all entities, actions, and constraints mentioned in the natural-language statement are
  explicitly represented and appropriately structured in Lean4 (e.g., via structure, def, or typed relations).
  Introducing entities or actions only implicitly, or omitting essential components of the
  scenario, constitutes structural inconsistency.
- Definition Integrity (Modeling vs. Proof Distinction):
In Lean4 formalization, any structure, function, action, or rule introduced during the modeling phase
  must be fully specified and semantically meaningful.
Within `def` or `structure`, no undeclared components, missing fields, or placeholder implementations
  (such as `sorry`) are permitted, as they indicate missing semantic content in the model.
Any defined `structure` must be meaningfully used in subsequent definitions or statements; unused or
  purely nominal `structure`s are considered structurally invalid.
In contrast, `sorry` is allowed only at the end of `theorem` or `lemma` declarations, since the task concerns
  automatic formalization rather than automated theorem proving.
However, using `sorry` inside a term, equation, hypothesis, or condition within a
  `theorem` or `lemma` is not permitted, as it indicates that the statement itself has not been properly formalized.
- **Avoiding Structural Collapse into Simple Calculations:** If the natural-language statement involves complex entities,
  combinatorial structures, or scenario-specific rules, ensure that the generated Lean4 code does not
  reduce the problem to a trivial numeric identity or closed-form computation.
  Replacing a structured selection, process, or constraint-based formulation with a direct calculation
  (e.g., a binomial coefficient or factorial expression) constitutes
  structural inconsistency, even if the numerical result is correct.
- Scope Awareness (Simple vs. Structured Problems):
Not all mathematical statements require explicit structural modeling.
For elementary or purely algebraic statements that involve no scenario-specific entities,
  actions, or constraints, a direct formalization using standard algebraic expressions is sufficient
  and should not be penalized for lacking auxiliary structures.
Structural inconsistency should be judged relative to the intrinsic complexity of the original natural-language statement,
  not by enforcing uniform use of structure or auxiliary definitions.
Use the ground-truth formalization **as a reference**; it is **not the gold standard**.
  Even if the generated code is very different from the ground-truth, it may still be correct —
  judge structural faithfulness by the original statement itself.
Provide a clear and concise explanation, then indicate whether the generated Lean4 code is structurally consistent.

Output Format:

<explanation>Your explanation here</explanation>
<answer>(Yes|No)</answer>

Here is the mathematical statement:

{statement}

Here is the generated Lean4 code:

{lean4code}

Here is the ground-truth Lean4 code (for reference only):

{lean4code_gt}

Figure 16: Prompt for structural consistency.