TopoAlign: A Framework for Aligning Code to Math via Topological Decomposition

Anonymous ACL submission

Abstract

Large Language Models (LLMs) excel at both informal and formal (e.g. Lean 4) mathematical reasoning but still struggle with *autoformalisation*, the task of transforming informal into formal mathematical statements. Yet, the performance of current Math LLMs is constrained by the scarcity of large-scale corpora, particularly those containing pairs of informal and formal statements. Interestingly, the formal languages used in autoformalisation share structural similarities with programming languages, and code data is available at scale. However, current models trained on code do not transfer effectively to formal math, due to structural and syntactic differences between them. To address this, we propose *TopoAlign*, a framework that unlocks widely available code repositories as training resources for Math LLMs. TopoAlign decomposes code into docstrings, main functions, and dependency functions, and reassembles these components into analogues that structurally mirror formal statements. We train three state-of-the-art models, DEEPSEEK-MATH, QWEN-3 and HERALD, and evaluate them on the MiniF2F, Putnam, and ProofNet benchmarks. TopoAlign provides substantial gains for DEEPSEEK-MATH, improving performance by 17.77% on BEq@10 and 68.82% on typecheck@10, and also measurably improves HERALD by 0.12% on BEq@10 and 1.09% on typecheck@10 despite introducing no new mathematical knowledge.

1 Introduction

Neuro-symbolic approaches that pair LLMs with proof assistants, such as Isabelle (Nipkow et al., 2002a) or Lean 4 (Moura and Ullrich, 2021), enable advanced math reasoning by enforcing rule-based logical consistency (Welleck and Saha, 2023). These assistants operate on Formal Languages (FL), such as Lean 4. However, proficiency in these formal languages requires specialized expertise; consequently most math problems are initially expressed in Natural Language (NL). While NL is ideal for human communication, its inherent flexibility and contextual dependence make it challenging to translate into a formal system. Bridging this gap requires *autoformalisation*, the process of faithfully translating informal NL math problems into FL. This step is essential for interacting with automated verifiers for tasks such as proof generation (Wu et al., 2022a; Ahn et al., 2024).

Despite recent advances, LLMs still struggle with autoformalization, in part due to the lack of large-scale, high-quality parallel datasets pairing natural language problem descriptions with corresponding formal statements or proofs (Wu et al., 2022a). Synthetic datasets such as Herald Statements (Gao et al., 2025), Numina v1.5 ATF (Guo et al., 2025), and Goedel-Pset (Lin et al., 2025) partially address this gap, but their scale and diversity remain limited and not annotated by human expert. Furthermore, these approaches rely on generating synthetic informal-formal pairings, which are inherently constrained by the quality and coverage of the underlying generative models, and do not scale naturally with the availability of existing large-scale resources such as code repositories. As a result, current models either fail outright or require thousands of attempts and auxiliary retrieval systems to produce accurate formalisations of even simple math problems (Li et al., 2024).

To overcome this data bottleneck, we look to the vast resources available in software engineering. Building on the assumption that models can learn the *structure* of a task from syntactically aligned data, even if the data is semantically distinct from the final domain. This insight unlocks widely available code repositories as large-scale data sources for training Math LLMs. Beyond providing scale, code data offers structural diversity, requiring complex problem-solving skills that we posit are transferable to formal mathematics.

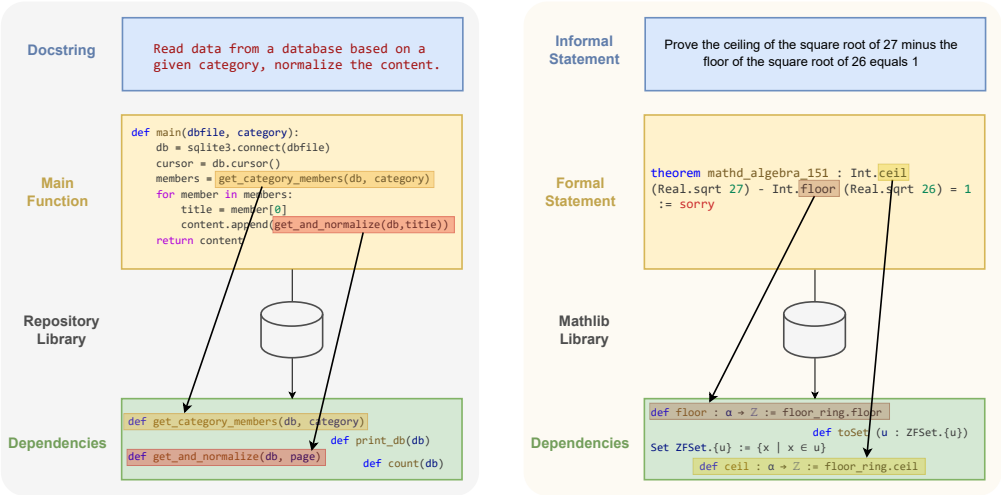

Figure 1: Structural similarity between code (left) and formal statements in Lean 4 (right). Code samples extracted from GitHub repositories are decomposed into: the docstring, which maps to informal statements in mathematical problems, the main function, which corresponds to the formal statements, and the dependency functions, which correspond to supporting lemmata and theorems, included in external libraries (e.g. Mathlib for Lean 4).

Leveraging these corpora requires aligning the compositional patterns of code with those of formal proofs. We propose *TopoAlign*, a framework that structurally aligns programming code with formal math. TopoAlign decomposes code into docstrings, main functions, and dependency functions, then reassembles these components into sequences that mirror the structure of Lean 4 formal statements (see Figure 1). This alignment teaches the model the compositional structure of formal math and enables the transfer of problem-solving capabilities learned from code without introducing new mathematical knowledge. Using this aligned corpus, we introduce *code autoformalisation* (CAF). This task emulates autoformalisation by aligning dependency functions, and main function bodies with informal descriptions (code docstrings), supporting lemmata, and formal statements in Lean. Unlike regular code generation, our setting provides a synthetic docstring that already includes the solution intent. We train DEEPSEEK-MATH (Shao et al., 2024), HERALD (Gao et al., 2025), and QWEN-3 (Yang et al., 2025) with TopoAlign and the CAF objective. On the MiniF2F, Putnam, and ProofNet benchmarks, the method yields consistent gains, achieving relative BEq improvements of 36.7% for DEEPSEEK-MATH and 6.2% for HERALD.

Contributions: 1) We introduce TopoAlign, a novel method addressing the shortage of training corpora for Math LLMs by structurally aligning code with formal mathematical languages. **2)** We propose CAF, a training task that leverages the structurally aligned code dataset to emulate autoformalisation, thereby reducing the dependence on annotated pairs of informal and formal mathematical statements. **3)** We release a large-scale semi-synthetic pre-training dataset of 300 million tokens, consisting of high-quality, structurally aligned code designed for autoformalisation tasks. **4)** Through detailed ablation studies, we demonstrate that a balanced ratio of our aligned code data and formal mathematical statements yields optimal autoformalisation performance.

2 Related Work

Previous methods for autoformalisation draw inspiration from machine translation literature (Wang et al., 2018; Dwivedi et al., 2022), i.e. Szegedy (2020) propose encoding NL and FL in a shared latent space and selecting translation candidates based on embedding similarity. Some approaches focus on rule-based methods, such as GFLean, which uses the Grammatical Framework for parsing and linearization (Pathak, 2024). However, these methods struggle to adapt to diverse inputs, as their rules require frequent updates. In contrast, LLMs provide more flexibility and consequently show strong autoformalisation performance (Jiang et al., 2022; Jiang, 2024; Wu et al., 2022b).

Despite their success in narrow domains (Soroco et al., 2025; Zhu et al., 2024), these methods face a common challenge: the scarcity of paired NL-FL math datasets. Various approaches are aimed to extend the training datasets: ATLAS (Liu

et al., 2025b) employs a student-teacher framework to generate synthetic data, but its performance is bounded by the teacher model's mathematical knowledge. Additionally, ATLAS generates fully synthetic data from existing Lean code. Herald Statements (Gao et al., 2025) are synthetically generated and limit the novelty of the resulting data. Jiang et al. (2024) demonstrate that high-quality multilingual data yields further performance gains in autoformalization, a finding echoed by Chan et al. (2025). Several high-quality, human-annotated datasets have been curated, including ProofNet (Azerbayev et al., 2022), MiniF2F (Zheng et al., 2022), Putnam (Tsoukalas et al., 2024), and the Mathlib library itself (mathlib Community, 2020). While invaluable, creating these datasets is resource-intensive, requires domain experts, and is consequently limited in scale. To address this, we find code data as a scalable alternative to mathematical data.

This is motivated by Chen et al. (2021a), who demonstrate that pretraining on code transfers to autoformalization, with Codex achieving notable few-shot performance on such tasks. As such, Math LLMs are initialised from LLMs trained on extensive code data and progressively fine-tuned on mathematical datasets. Llemma (Zhang et al., 2024), Kimina (Wang et al., 2025) and DEEPSEEKMATH (Shao et al., 2024) follow this paradigm. Li et al. (2024) argue that the autoformalization potential of Math LLMs remains underexplored when leveraging general-purpose code data during pretraining. To address this, we propose topologically decomposing and aligning code from repositories with formal math statements, unlocking a scalable new source of training data for LLMs.

Finally, given data scarcity, previous work explored techniques for efficient usage of available data. Related methods aim to extract the inherent relationships between NL and FL in the data, proposing alignment methods based on symbolic equivalence and semantic consistency (Li et al., 2024). However, aligning NL and FL remains challenging when their formats and structures differ, making it difficult to transfer the LLMs knowledge and reasoning capabilities between NL and FL.

3 Methodology

We posit that current Math LLM performance is constrained primarily by the scarcity of large-scale training corpora. Large code datasets have already proven valuable for initializing these models (Shao et al., 2024; Wang et al., 2025), yet code data remains largely untapped during subsequent training on formal mathematics. Our TopoAlign framework and CAT address this gap, detailed as follows.

3.1 Topological Decomposition of Code for Structural Alignment with Formal Math

TopoAlign builds on the premise that code and formal mathematical statements share a compositional structure. Functions in code solve distinct subproblems and may rely on auxiliary functions, analogous to how formal statements resolve informal problem descriptions using lemmata and theorems. We therefore decompose code at the function level (see Figure 1) into three transferable components: (i) the docstring, corresponding to the informal problem statement, (ii) the main function body, serving as a proxy for the formal statement; and (iii) its dependency functions, corresponding to supporting lemmata or library theorems (i.e., from Mathlib in Lean 4). We select Lean as the representative mathematical formal language in this work, while other systems such as Isabelle (Nipkow et al., 2002b) and Coq (The Coq Development Team, 2020) are discussed in Appendix H. Disassembling code into components that mirror those in formal mathematics enables structural transfer. Importantly, TopoAlign targets structural alignment and does not assume semantic equivalence between programming-language type systems and Lean's dependent type system. The framework provides complementary pretraining data that **enhances, rather than replaces**, training on formal mathematical corpora.

To extract functional dependencies, we employ a topological dependency parser that performs a breadth-first search to build function-level dependency graphs (see Algorithm 1 in Appendix G). This contrasts with file-level dependency extraction (i.e., DeepSeek (Guo et al., 2024)), which captures inter-file execution order but omits intra-file functional relationships. Our parser leverages abstract syntax trees to trace calls and parent definitions across files, and is designed to handle standalone functions, class or instance methods, recursive calls, and imports. The result is a tree-structured representation of code dependencies, as shown in Figure 2.

To obtain informal problem statements, we extract natural language descriptions from docstrings and README files. To obtain informal problem

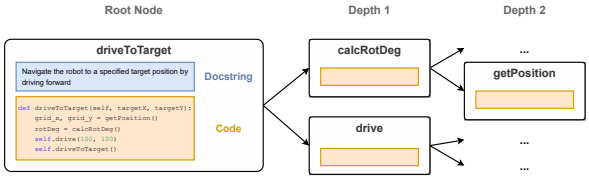

Figure 2: Function-level dependency tree showing the hierarchy of function calls, starting from the root node. Each child node represents a function called by its parent. The docstring for the root node is extracted to represent the description of the problem addressed.

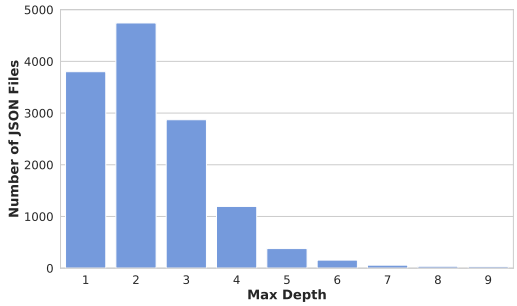

Figure 3: Distribution of maximum dependency tree depths across a random sample of 200 repositories.

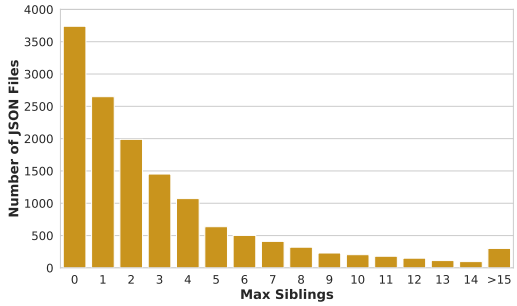

Figure 4: Distribution of the maximum number of siblings in dependency trees across a random sample of 200 repositories.

statements, we extract natural language descriptions from docstrings and README files. However, as docstrings describe a function's interface rather than its implementation, they do not always align with the needs of autoformalization. We therefore use QWEN-3 (Yang et al., 2025) to generate concise summaries of each function's implementation logic, augmenting missing or low-quality documentation. We perform 10-gram contamination analysis, following Guo et al. (2024), to ensure no information leakage between our dataset and the test set, and confirm no meaningful overlap.[1]

We analyse the structural properties of function-level dependency trees in the collected code to select repositories whose hierarchical patterns resemble those found in formal mathematics. For each repository we compute the maximum tree depth, reflecting overall complexity and the maximum number of sibling nodes at any depth, indicating the breadth of direct dependencies. We retain repositories with depth between 3 and 6 and maximum sibling counts between 3 and 10, thus excluding overly complex codebases as well as simple scripts. Figures 3 and 4 show the distributions of maximum depth and maximum sibling count for a random sample of 200 repositories. After filtering, the corpus contains 156,684 functions comprising 324.5 million tokens.[2]

3.2 Code Autoformalisation (CAF)

To leverage structurally aligned code for training Math LLMs on autoformalisation, we introduce CAF, which emulates the autoformalisation process on code data. Treating each aligned code instance as an analogue of a formalisation scenario allows the model to learn structural patterns while transferring general problem-solving strategies acquired in programming. Importantly, our method focuses on transferring structural and problem-solving capabilities, rather than introducing new mathematical knowledge. While we recognize that this technique could be adapted to emulate other formal reasoning tasks like theorem proving, in this work, we focus specifically on addressing the prevalent data scarcity bottleneck in autoformalisation.

We train Math LLMs using a mixture of TopoAlign code data and formal mathematical data. This combined approach integrates mathematical knowledge and structure with problem-solving capabilities from code, while also mitigating catastrophic forgetting during fine-tuning (Chen et al., 2019). In our multi-task training approach, each training sample consists of an input x, a docstring for code or an informal statement for math, and a set of dependencies d, a set of dependency functions for code or supporting lemmata and theorems for math. The model is conditioned on x and d and trained to generate the target y: the main function for code data or the formal statement for math data, as illustrated in Figure 5.

The proportion of code and math samples during training is controlled by parameter α, where α determines the fraction of math samples and $1-\alpha$ the fraction of code samples. The overall objective is defined as $\mathcal{L} = \alpha\,\mathcal{L}_{\text{math}} + (1-$

[1] Additional details are provided in Appendix F.

[2] Dataset available at: https://anonymous.4open.science/r/TopoAlign-3BAD.

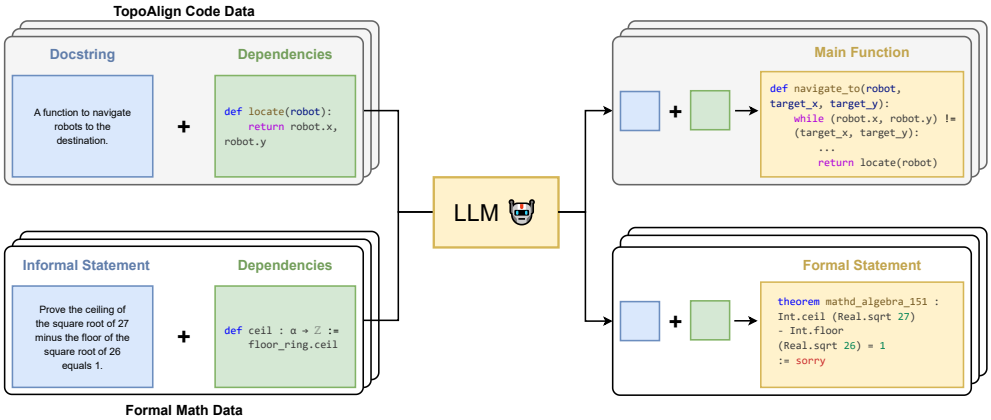

Figure 5: Overview of the training pipeline. Input consists of a problem description (for either code or math) and its dependencies. The training objective is to generate the corresponding solution: the root code block for code inputs, or the formal statement for math inputs.

$\alpha)\,\mathcal{L}_{\text{CAF}}$, where $\mathcal{L}_{\text{math}}$ and $\mathcal{L}_{\text{CAF}}$ denote the losses for math and code tasks, respectively. For each task, the loss is computed using next-token prediction, formulated as the negative log-likelihood $\mathcal{L} = -\sum_{i=1}^{N} \log P_\theta(y_i \mid y_{<i}, x)$, where x is the input sequence, $y = (y_1, y_2, \ldots, y_N)$ is the target sequence, and P_θ is the model with parameters θ.

4 Experiments

Training Data. Our code data is sourced from Python repositories in the Stack v2 dataset (Lozhkov et al., 2024). For formal mathematical statements, we use the Herald Statements corpus (Gao et al., 2025), a synthetic dataset built from Mathlib (mathlib Community, 2020). It contains Lean statements similar in style and structure to those found in math pretraining corpora, while avoiding overlap with downstream evaluation benchmarks such as MiniF2F, ProofNet, and Putnam. Dependency functions for the formal statements are extracted using the jixia library.[3]

Models. We evaluate three base models: DEEPSEEK-MATH (7B) (Shao et al., 2024), HERALD (7B) (Gao et al., 2025) and QWEN-3 (4B) (Yang et al., 2025). We evaluate three base models: DEEPSEEK-MATH (7B) (Shao et al., 2024), HERALD (7B) (Gao et al., 2025) and QWEN-3 (4B) (Yang et al., 2025) with their instruct versions. HERALD is specialised for autoformalization, having been trained on the synthetic Herald Statements data. In contrast, DEEPSEEK-MATH is trained on a broader range of mathematical data from DeepSeek-Coder (Guo et al., 2024), optimised for general mathematical problem-solving rather than explicit autoformalization. We additionally evaluate QWEN-3, a state-of-the-art general-purpose reasoning model pre-trained on large-scale mathematical corpora to enhance its reasoning capabilities. This setup allows us to compare a dedicated autoformalization model against models with broader mathematical and code understanding.

We should note that TopoAlign should be understood as a continual pre-training methodology for math autoformalisation from code repositories, intended for use in the pre- and mid-training phases of LLMs providing more structural information and problem understanding information. As such, it is orthogonal to fine-tuning-based approaches. It follows that to meaningfully compare TopoAlign against fine-tuned SOTA formalizers, it would require to retrain the latter with TopoAlign data under identical conditions, which is beyond the scope of this study. Instead, we compare against baselines using different data distributions as follows, which we argue provides a fair and interpretable evaluation of TopoAlign's contribution.

Settings. For each base model, we evaluate four distinct variations, namely **Baseline**, **Math**, **Code** and **TopoAlign**. The **Baseline** setting evaluates the pretrained models directly on downstream tasks without any additional fine-tuning. The **Math** setting involves fine-tuning models exclusively on formal mathematical data from the Herald Statements corpus, which is equivalent to applying the CAF objective with a mixing ratio of $\alpha = 1$; this allows us to assess the impact of training on purely mathematical data. In the **Code** setting, models are trained only on unaligned code data extracted

[3] https://github.com/frenzymath/jixia

from the Stack-V2 corpus, providing a control to test the effect of training without structural alignment via TopoAlign. Finally, the **TopoAlign** setting trains models on a balanced combination of formal mathematics and structurally aligned code using CAF, with a mixing ratio of $\alpha = 0.5$. This setting uses 4,000 randomly selected samples from GitHub repositories and 4,000 from the Herald Statements dataset. For consistency, the number of training samples is kept equal across all settings. Hyperparameter and prompt details are provided in Appendix C.

Benchmarks. We evaluate models on several Lean 4 autoformalisation benchmarks: Putnam (Tsoukalas et al., 2024), the validation and test sets of MiniF2F (Zheng et al., 2022), and ProofNet (Azerbayev et al., 2022). Each benchmark consists of paired natural language and formal language statements. Among these, Putnam is the most challenging problems, ProofNet comprises textbook theorems, and MiniF2F is considered the most in-distribution benchmark, as it overlaps with Mathlib.

Evaluation Metrics. We measure model performance using two primary metrics. First, we use Typecheck (TC) with the Lean 4 compiler (v4.21.0) to verify the syntactic correctness of generated statements (Poiroux et al., 2025; Limperg, 2025; Rabe et al., 2020; Wu et al., 2022b). For a more rigorous assessment of semantic fidelity, we employ bidirectional equivalence (BEq) (Liu et al., 2025a), which uses an LLM to generate proof tactics establishing logical equivalence between the model's output and the reference statement. This provides a stronger signal of faithful autoformalization than typechecking alone, and is consistent with the established LLM-as-judge evaluation paradigm (Gu et al., 2024).

5 Results and Discussion

Table 1 presents the Typecheck and BEq results under the pass@k metric (Chen et al., 2021b) with $k = \{1, 10\}$, where a problem is considered solved if at least one of the k sampled solutions is correct, evaluated on the MiniF2F-valid, MiniF2F-test, ProofNet, and Putnam datasets. We additionally report unbiased pass@1 results in Appendix A.1.

Our proposed TopoAlign method outperforms all benchmarked methods across all datasets and model families on the BE@10 metric, with the exception of HERALD on ProofNet. Table 5 shows that TopoAlign achieves the best overall model performance, measured by average rank across datasets, for DEEPSEEK-MATH, QWEN-3, and HERALD. For example, DEEPSEEK-MATH BEq@1 on MiniF2F-valid increases by 9.65% to 14.47%. Some variance in BEq@1 is expected due to stochastic sampling, which can account for occasional drops, such as the slightly lower score for HERALD on Putnam. In the more robust BEq@10 evaluation, TopoAlign yields consistent improvements across DEEPSEEK-MATH, QWEN-3, and HERALD on Putnam, MiniF2F-valid, and MiniF2F-test. These results demonstrate that topological alignment effectively transfers structural knowledge from code to formal mathematics even, though it introduces no additional mathematical knowledge. These gains are observed across model architectures and datasets.

TopoAlign consistently surpasses code-only variants for both HERALD and DEEPSEEK-MATH on BEq and Typecheck, indicating that unaligned code data alone does not provide structural information beneficial for autoformalization. This is further evidenced by HERALD, which already has more math knowledge than other base models, trained solely on code data. Its BEq scores and typecheck accuracy decreases significantly, with generated outputs frequently syntactically invalid despite occasional semantic plausibility (see Appendix D). This suggests that raw code data not only fails to benefit autoformalization but may actively degrade an already capable model's formalization ability, highlighting the necessity of structural alignment and a proper training paradigm. In contrast, topological alignment and the CAF task enable effective transfer of both problem-solving and structural knowledge to formal mathematics. Notably, QWEN-3 trained on raw code frequently generates degenerate outputs such as endless type definitions (e.g. Nat $\rightarrow$ Nat), which pass typechecking but are semantically meaningless, suggesting prior exposure to code typechecking during training.

These findings confirm that TopoAlign and CAF effectively leverage code data to enhance Math LLM training, addressing the scarcity of formal mathematical corpora by unlocking code repositories as a scalable source of structurally aligned pretraining data. Principally, TopoAlign unlocks code repositories as structurally aligned pretraining data while leaving the underlying code content unchanged, and CAF then facilitates the transfer of this knowledge from code to mathematics.

Table 1: Auto-formalisation performance in percent for MiniF2F, ProofNet, and Putnam datasets under pass@1 and pass@10 metrics for Typecheck (TC) and BEq. Baseline setting refers to the pretrained model, Math is trained on additional formal math data and code is trained on additional code data that is not structurally aligned. Topoalign mixes math and structurally aligned code data.

Dataset	Base Model	Setting	TC@1	BEq@1	TC@10	BEq@10
MiniF2F-valid	DEEPSEEK-MATH	Baseline	0.00	0.00	0.00	0.00
		Math	45.18	9.65	79.82	19.74
		Code	42.98	10.53	**88.16**	23.68
		TopoAlign	**51.75**	**14.47**	87.28	**26.32**
	QWEN-3	Baseline	2.63	0.00	21.93	0.00
		Math	81.58	**13.16**	**100.00**	82.02
		Code	**97.37**	0.00	**100.00**	0.00
		TopoAlign	77.19	**13.16**	**100.00**	**85.53**
	HERALD	Baseline	73.25	25.44	92.98	38.16
		Math	75.44	24.12	93.86	**41.23**
		Code	75.00	20.61	90.79	32.02
		TopoAlign	**76.75**	**26.75**	**94.74**	**41.23**
MiniF2F-test	DEEPSEEK-MATH	Baseline	0.00	0.00	0.00	0.00
		Math	43.11	9.33	77.33	21.78
		Code	52.89	7.56	**91.52**	26.79
		TopoAlign	**54.67**	**16.89**	88.89	**29.78**
	QWEN-3	Baseline	6.22	0.00	37.33	0.00
		Math	84.44	10.22	**100.00**	74.22
		Code	**97.78**	0.00	**100.00**	0.00
		TopoAlign	88.89	**23.11**	**100.00**	**84.00**
	HERALD	Baseline	78.22	24.44	**95.54**	41.96
		Math	**80.89**	24.44	**95.54**	40.36
		Code	**80.89**	20.89	90.63	34.38
		TopoAlign	79.56	**27.56**	94.20	**42.86**
ProofNet	DEEPSEEK-MATH	Baseline	0.00	0.00	0.00	0.00
		Math	21.12	5.35	43.42	12.57
		Code	22.73	2.67	49.47	7.49
		TopoAlign	**32.89**	**9.09**	**56.95**	**14.97**
	QWEN-3	Baseline	6.42	0.16	31.02	0.80
		Math	28.34	**5.88**	88.77	37.70
		Code	**56.15**	0.00	**99.47**	0.00
		TopoAlign	31.02	4.55	95.72	**41.71**
	HERALD	Baseline	46.52	10.16	74.87	**20.32**
		Math	**46.79**	**10.43**	75.40	19.77
		Code	38.24	4.55	63.37	9.36
		TopoAlign	43.85	9.63	**75.67**	16.84
Putnam	DEEPSEEK-MATH	Baseline	0.00	0.00	0.00	0.00
		Math	10.69	0.00	27.04	0.00
		Code	12.58	0.00	37.42	0.00
		TopoAlign	**17.30**	0.00	**42.14**	0.00
	QWEN-3	Baseline	1.89	0.00	16.98	0.00
		Math	25.79	0.63	98.11	5.35
		Code	**70.44**	0.00	**100.00**	0.00
		TopoAlign	19.50	**1.26**	97.48	**9.12**
	HERALD	Baseline	37.42	2.20	73.58	**4.72**
		Math	**43.71**	**2.52**	70.44	4.09
		Code	35.85	0.00	69.18	0.00
		TopoAlign	36.16	1.57	**76.73**	**4.72**

5.1 Qualitative Error Analysis

To better understand model behavior, we conduct a qualitative error analysis on 200 failure cases, comprising 50 randomly selected samples per dataset generated by QWEN-3 with TopoAlign. We categorize the errors into four distinct types as follows, with representative cases detailed in Appendix E.

Type 1: Missing or Incorrect Conditions, i.e. conditions are omitted or improperly specified, such as missing universal quantifiers or weakened constraints. **Type 2: Semantically Equivalent but Structurally Different,** i.e. generated statements are semantically correct but differ in structural representation, such as using set membership in place of disjunctive equality. **Type 3: Variable Type Mismatch,** i.e. variables are assigned incorrect or incompatible types, producing statements that are syntactically plausible but semantically invalid.

Type 4: Representation Inconsistency, i.e. variables or expressions are inconsistently represented, leading to semantic mismatches, such as conflating absolute value notation with distance functions.

The distribution of error types across datasets is summarised in Table 2. The majority of errors fall under Type 1, accounting for approximately 70% of all failures. The primary bottleneck is deemed as deep problem comprehension.

Table 2: Distribution of error types across datasets (50 samples per dataset) generated by QWEN-3.

Dataset	Type 1	Type 2	Type 3	Type 4	Others
Minif2f-test	23	15	5	4	3
Minif2f-valid	28	14	5	2	1
ProofNet	43	2	1	1	3
Putnam	47	1	2	0	0

This distribution also explains the uneven performance gains across datasets. For harder benchmarks such as ProofNet and Putnam, Type 1 errors overwhelmingly dominate, averaging over 90% of failures, indicating that implicit natural language constraints are too complex for the model to parse reliably. Since TopoAlign provides structural alignment rather than additional mathematical knowledge, models with stronger inherent reasoning capabilities benefit more substantially, as they can actively leverage the aligned structure. Weaker models, by contrast, remain bottlenecked by their inability to comprehend the underlying math.

5.2 Impact of Code–Math Data Ratio

We further explored the effect of varying the ratio of code to mathematical data in the CAF objective by adjusting the α parameter. We trained additional DEEPSEEK-MATH models with α values of 0.25 and 0.75, and report the results in Table 3. The balanced ratio ($\alpha = 0.5$) consistently yields the highest or near-highest performance for both Typecheck@1 and BEq@1. Lowering the mathematical content ($\alpha = 0.25$) leads to the weakest results, likely due to the token-level loss being dominated by the larger code samples, biasing the model toward code generation. Increasing the proportion of mathematical data ($\alpha = 0.75$) improves Typecheck@1 scores, especially on benchmarks closer to Mathlib, such as MiniF2F, but does not consistently improve BEq@1. These findings indicate a balanced mix of code and mathematical data is crucial for optimal autoformalisation performance: math data enhances syntactic accuracy, while code enhances problem-solving capabilities.

Table 3: Ablation study on training data composition for DEEPSEEK-MATH. The table compares Typecheck (TC) and BEq scores for pass@1 to identify the optimal ratio of formal math data to our aligned code data.

Dataset	Data ratio	TC@1	BEq@1
Putnam	$\alpha = 0.25$	12.58	0.00
	$\alpha = 0.50$	**17.30**	0.00
	$\alpha = 0.75$	12.89	0.00
ProofNet	$\alpha = 0.25$	24.60	6.15
	$\alpha = 0.50$	**32.89**	**9.09**
	$\alpha = 0.75$	26.20	8.29
MiniF2F-valid	$\alpha = 0.25$	44.74	11.84
	$\alpha = 0.50$	51.75	**14.47**
	$\alpha = 0.75$	**52.63**	10.96
MiniF2F-test	$\alpha = 0.25$	42.67	7.56
	$\alpha = 0.50$	54.67	**16.89**
	$\alpha = 0.75$	**56.44**	14.67

5.3 Base model performance

The DEEPSEEK-MATH base model, which lacks pretraining on autoformalisation tasks, fails to generate meaningful outputs, often producing repeated symbols or malformed syntax that result in typecheck failures. In contrast, HERALD, which is pretrained on a large mathematical corpus, provides strong BEq performance across multiple datasets. Notably, the introduction of CAF training further improves its results: for instance, BEq@1 on MiniF2F-valid increases from 23.07% to 25.08%, and on MiniF2F-test from 22.70% to 25.64%. These improvements confirm that TopoAlign provides significant benefits even for models that already possess strong autoformalisation capabilities.

6 Conclusion

This work shows that widely available code repositories are a valuable, previously untapped resource for pretraining Math LLMs. We introduce TopoAlign, which structurally aligns code with formal mathematics, yielding a 324.5 million token dataset that mirrors the structure of formal mathematical statements. Training on this dataset leads to substantial improvements across four autoformalization benchmarks in both Typecheck and BEq metrics, confirming that with a principled alignment paradigm, structural and problem-solving knowledge can be effectively transferred from code to formal mathematics. Ablation studies further reveal that a balanced mix of code and formal mathematical data is essential. Beyond autoformalization, TopoAlign's alignment framework may extend to tasks such as proof generation in future work.

Limitations

Due to computational and resource constraints, our main experiments focus on a subset of approximately 4,000 code samples and 4,000 math samples. We would like to emphasize 324.5M tokens are currently only considering code repositories in Python (around 28% in StackV2). This corpus can be further extended using other programming languages in StackV2. Additionally, this work could be expanded for training more samples.

Moreover, while TopoAlign is designed to augment and structurally enrich existing training data rather than replace high-quality mathematical corpora, its performance on research-level benchmarks such as FATE-H (Jiang et al., 2025) remains limited, with all models yielding a BEq of 0.00. This reflects an inherent constraint: structural alignment cannot substitute for the advanced mathematical knowledge that only high-quality formal datasets can provide, and such datasets remain scarce at the research level. Nevertheless, this is consistent with the intended scope of TopoAlign, which aims to enhance the structural quality of training data rather than expand its mathematical coverage.

Ethic consideration

This research focuses exclusively on computational methods for mathematical reasoning and the development of machine learning frameworks to improve autoformalisation. It does not involve human subjects, personal data, or any intervention in real-world populations. All experiments are conducted on publicly available datasets or synthetic data derived from code repositories, and no sensitive or private information is used. Therefore, this work does not raise any ethical concerns or require special ethical review.

References

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 225–237, St. Julian's, Malta. Association for Computational Linguistics.

Zhangir Azerbayev, Bartosz Piotrowski, and Jeremy Avigad. 2022. Proofnet: A benchmark for autoformalizing and formally proving undergraduate-level mathematics problems. In *Workshop on MATH-AI: Toward Human-Level Mathematical Reasoning, NeurIPS 2022*. Poster presented virtually at NeurIPS 2022.

W. Chan, M. Souliman, J. Nordhagen, and 1 others. 2025. Lean-ing on quality: How high-quality data beats diverse multilingual data in autoformalization. *arXiv preprint arXiv:2502.15795*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021a. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021b. Evaluating large language models trained on code.

Xinyang Chen, Sinan Wang, Bo Fu, Mingsheng Long, and Jianmin Wang. 2019. Catastrophic forgetting meets negative transfer: Batch spectral shrinkage for safe transfer learning. *Advances in neural information processing systems*, 32.

Vijay Prakash Dwivedi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. 2022. Graph neural networks with learnable structural and positional representations. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

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 *Proceedings of the International Conference on Learning Representations (ICLR)*. Poster presented at ICLR 2025, Singapore EXPO; slides available online.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, and 1 others. 2024. A survey on llm-as-a-judge. *The Innovation*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence, 2024. *URL https://arxiv.org/abs/2401.14196*, 5:19.

Qi Guo, Jianing Wang, Jianfei Zhang, Deyang Kong, Xiangzhou Huang, Xiangyu Xi, Wei Wang, Jingang Wang, Xunliang Cai, Shikun Zhang, and 1 others. 2025. Autoformalizer with tool feedback. *arXiv preprint arXiv:2510.06857*.

A. Jiang, C. E. Staats, C. Szegedy, and 1 others. 2022. Autoformalization with large language models. *2022*.

A. Q. Jiang, W. Li, and M. Jamnik. 2024. Multi-language diversity benefits autoformalization. *Advances in Neural Information Processing Systems*, 37:83600–83626.

Jiedong Jiang, Wanyi He, Yuefeng Wang, Guoxiong Gao, Yongle Hu, Jingting Wang, Nailin Guan, Peihao Wu, Chunbo Dai, Liang Xiao, and 1 others. 2025. Fate: A formal benchmark series for frontier algebra of multiple difficulty levels. *arXiv preprint arXiv:2511.02872*.

Q. Jiang. 2024. *Language Models for Verifiable Mathematical Automation Interaction, Integration, and Autoformalization*. Ph.D. thesis, University of Cambridge, United Kingdom.

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

Jannis Limperg. 2025. Tactic script optimisation for aesop. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 98–111.

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

Q. Liu, X. Zheng, X. Lu, and 1 others. 2025a. 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, Yuntian Liu, Yu Chen, Yang Jiao, and Tao Luo. 2025b. Atlas: Autoformalizing theorems through lifting, augmentation, and synthesis of data. *Preprint*, arXiv:2502.05567.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

The mathlib Community. 2020. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, POPL '20, page 367–381. ACM.

Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, pages 625–635. Springer.

Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002a. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002b. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer.

Shashank Pathak. 2024. Gflean: An autoformalisation framework for lean via gf. *Preprint*, arXiv:2404.01234.

Auguste Poiroux, Gail Weiss, Viktor Kunčak, and Antoine Bosselut. 2025. Improving autoformalization using type checking. *Preprint*, arXiv:2406.07222.

Markus N Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. 2020. Mathematical reasoning via self-supervised skip-tree training. *arXiv preprint arXiv:2006.04757*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Mauricio Soroco, Jialin Song, Mengzhou Xia, Kye Emond, Weiran Sun, and Wuyang Chen. 2025. Pde-controller: Llms for autoformalization and reasoning of pdes. *Preprint*, arXiv:2502.00963.

C. Szegedy. 2020. A promising path towards autoformalization and general artificial intelligence. In *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings 13*, pages 3–20, Bertinoro, Italy. Springer International Publishing.

The Coq Development Team. 2020. *The Coq Proof Assistant Reference Manual*. INRIA. https://coq.inria.fr.

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. In *NeurIPS 2024 Poster Session*. Virtual poster presented at the Thirty-eighth Annual Conference on Neural Information Processing Systems (NeurIPS), Vancouver.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, and 1 others. 2025. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*.

Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. 2018. First experiments with neural translation of informal to formal mathematics. *Preprint*, arXiv:1805.06502.

Sean Welleck and Rahul Saha. 2023. llmstep: Llm proofstep suggestions in lean. In *Proceedings of the 3rd Workshop on Mathematical Reasoning and AI (MATH-AI), NeurIPS 2023*. Poster presented virtually at NeurIPS 2023.

Y. Wu, A. Jiang, W. Li, and 1 others. 2022a. Autoformalization for neural theorem proving. *Artificial Intelligence and Theorem Proving (AITP)*.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022b. Autoformalization with large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 32353–32368. Curran Associates, Inc.

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*.

Lan Zhang, Xin Quan, and Andre Freitas. 2024. Consistent autoformalization for constructing mathematical libraries. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, page 4020–4033. Association for Computational Linguistics.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2022. minif2f: A cross-system benchmark for formal olympiad-level mathematics. In *Proceedings of the International Conference on Learning Representations (ICLR)*. OpenReview.net.

N. Zhu, X. Zhang, Q. Huang, and 1 others. 2024. Fgeoparser: Autoformalization and solution of plane geometric problems. *Symmetry*, 17(1):8.

A Additional results

A.1 Unbiased results

We report the unbiased results in Table 4 as supplementary evidence supporting our main findings. These results, presented alongside prior literature for direct comparison, show a trend where TopoAlign appears to outperform other training frameworks in terms of BEq score and Typecheck. Due to resource constraints, we reported pass@1, but they still provide supporting evidence for our hypothesis.

Table 4: Auto-formalisation performance in percent for MiniF2F, ProofNet, and Putnam datasets under **unbiased** pass@1 metrics for Typecheck (TC) and BEq. Baseline setting refers to the pretrained model, Math is trained on additional formal math data and code is trained on additional code data that is not structurally aligned. Topoalign mixes math and structurally aligned code data.

Dataset	Base Model	Setting	TC	BEq
MiniF2F-valid	DEEPSEEK-MATH	Baseline	2.19	0.00
		Math	38.16	10.92
		Code	36.44	12.41
		TopoAlign	**44.92**	**15.90**
	QWEN-3	Baseline	2.19	0.00
		Math	78.46	**13.95**
		Code	**97.37**	0.00
		TopoAlign	77.68	**15.31**
	HERALD	Baseline	78.34	23.07
		Math	75.74	24.69
		Code	78.59	21.49
		TopoAlign	**78.38**	**25.08**
MiniF2F-test	DEEPSEEK-MATH	Baseline	0.00	0.00
		Math	46.40	7.91
		Code	48.89	7.47
		TopoAlign	**55.83**	**15.61**
	QWEN-3	Baseline	4.13	0.00
		Math	84.00	13.33
		Code	**98.67**	0.00
		TopoAlign	79.60	**17.33**
	HERALD	Baseline	77.78	22.70
		Math	**81.02**	21.64
		Code	80.84	21.29
		TopoAlign	80.45	**25.64**
ProofNet	DEEPSEEK-MATH	Baseline	0.00	0.00
		Math	21.65	4.33
		Code	20.46	1.79
		TopoAlign	**33.72**	**8.32**
	QWEN-3	Baseline	4.84	0.80
		Math	27.54	4.60
		Code	**54.55**	0.00
		TopoAlign	30.03	**4.63**
	HERALD	Baseline	46.92	10.13
		Math	**47.00**	9.12
		Code	27.77	2.55
		TopoAlign	46.63	**9.84**
Putnam	DEEPSEEK-MATH	Baseline	0.00	0.00
		Math	8.14	0.00
		Code	10.82	0.00
		TopoAlign	**16.26**	0.00
	QWEN-3	Baseline	1.73	0.00
		Math	29.84	0.57
		Code	**69.18**	0.00
		TopoAlign	22.96	**0.91**
	HERALD	Baseline	41.98	2.76
		Math	**43.87**	**2.87**
		Code	34.84	0.00
		TopoAlign	36.03	1.70

A.2 Rank

We also provide a rank-based comparison in Table 5, statistically summarized via the standard pass@k metric, to offer a concise overview across different training settings. This is more interpretable than the raw numbers reported in Table 4 and Table 1. From the average-rank perspective, TopoAlign consistently achieves an average rank between 1 and 2, outperforming all other settings, which further corroborates the effectiveness of TopoAlign.

Table 5: Average rank across different settings.

Model	Setting	Avg.Rank ↓
DEEPSEEK-MATH	Baseline	3.63
	Math	2.50
	Code	2.00
	TopoAlign	**1.13**
QWEN-3	Baseline	3.50
	Math	1.94
	Code	2.13
	TopoAlign	**1.63**
HERALD	Baseline	2.25
	Math	1.81
	Code	3.75
	TopoAlign	**1.69**

B Software and Data

To ensure full reproducibility, all data sources, tools, and code are publicly available. All three basemodels we select from huggingface checkpoint (https://huggingface.co/deepseek-ai/deepseek-math-7b-instruct, https://huggingface.co/Qwen/Qwen3-8B, https://huggingface.co/FrenzyMath/Herald_translator).The downstream evaluation benchmarks were sourced directly from their official repositories: MiniF2F (https://github.com/openai/miniF2F), Putnam (https://github.com/trishullab/PutnamBench), and Herald (https://huggingface.co/datasets/FrenzyMath/Herald_statements). Ground truth dependencies were processed using Jixia (https://github.com/frenzymath/jixia), as detailed in Section 4. Our complete codebase and the generated dataset have been released at https://anonymous.4open.science/r/TopoAlign-3BAD.

C Training Hyperparameters

The hyperparameters used for training the LLMs are listed in Tables 6 and 7. We adhere to the default settings provided in the original papers. In our setup, we fine-tune the base models using supervised fine-tuning. The implementation is based on the Hugging Face Transformers library (version 4.52).

D Generated Sample from Code-Only Trained Model

These samples are generated by HERALD trained exclusively on code data. The outputs demonstrate that while the model captures the underlying logical structure correctly, it lacks proficiency in Lean 4 syntax. However, this suggests the presence of semantic similarity between programming code and formal mathematical language.

```
from math import sqrt, ceil, floor

def evaluate_sqrt_expressions_1():
    assert(int(ceil(sqrt(27))) -
     int(floor(sqrt(26))) == 1)


def is_increasing_function(a, b):
    '''
    The function checks if the function f(x) =
     4bx + (a+1)x^2 is increasing for x >= 0.
    '''
    return 4*b <= 4*b**2 + (a+1)**2


def is_3_(girls: list[int]) -> int:
    """
    >>> is_3_([1,2,3,4,5,6,7])
    3
    """
    return 3
```

Hyperparameters	DEEPSEEK-MATH	HERALD
learning rate	2e-4	1e-6
weight decay	0.1	0.1
epochs	1	1
batch size	32	32

Table 6: Training hyperparameter settings.

Hyperparameters	DEEPSEEK-MATH	HERALD
temperature	0.70	0.70
top p	0.95	0.95
max token	1024	1024

Table 7: Inference hyperparameter settings used for autoformalisation in the pass@10 setting.

E Error Analysis Examples

Type 1: Missing or Incorrect Conditions. The generated output losses the universal quantification over all positive k and all integer multiples m. The hypothesis `h₁ : 0 < n` is also entirely omitted.

Reference:
```
h₀ : ∀ k : ℕ, 0 < k → ∀ m : ℤ, x
≠ m * π / (2^k)
h₁ : 0 < n
```
Generated:
```
hx : x ≠ π / (2^(n+1))
```

Type 2: Semantically Equivalent but Structurally Different. The generated output uses set membership notation $\in \{\cdots\}$ in place of explicit disjunctive equality. While the two expressions are semantically equivalent, they differ structurally, which may cause mismatches during formal verification or proof search.

Reference:
```
(x, y) = (1, 1) ∨ (x, y) = (16, 2)
∨ (x, y) = (27, 3)
```
Generated:
```
(x, y) ∈ { (1, 1), (16, 2), (27, 3)
}
```

Type 3: Variable Type Mismatch. The generated output assigns an integer type $\mathbb{Z}$ to a variable that should be typed as a natural number $\mathbb{N}$. Although syntactically plausible, this type mismatch renders the statement semantically invalid within the formal type system.

Reference:
```
n : ℕ
```
Generated:
```
n : ℤ
```

Type 4: Representation Inconsistency. The reference uses `abs` for absolute value and `Real.sqrt` with explicit coordinate arithmetic for Euclidean distance. The generated output instead applies the `dist` function inconsistently: once as a unary distance from zero (conflating absolute value), and once as a binary point distance, mixing two distinct representations within the same statement.

Reference:
```
abs y = 6
Real.sqrt ((x - 8)^2 + (y - 3)^2)
```
Generated:
```
dist x 0 = 6
dist (x, y) (8, 3)
```

F Training Instructions and Inference Prompts

To generate docstrings describing the main functions in the code repositories, we employ Qwen3-14B model (https://huggingface.co/Qwen/Qwen3-14B) using the following prompt format:

Prompt

Provide a concise description of the problem solved in the code snippet below. Format the response as a docstring.
{code}

We manually verified 50 randomly sampled docstrings from our dataset, and all of them exhibit accurate and complete descriptions of the corresponding code. Selected samples are provided below:

```
def sync(self, title: str, keywords:
    Union[str, list[str]]) -> dict:

    title = validate_title(title)
    keywords = validate_keywords(keywords)
    filter_item = self.filter(title)
    remote_keywords = filter_item["keywords"]
    [...]
    params = OrderedDict(
        {
            "title": title,
            "context[]":
    filter_item["context"],
            "filter_action":
    filter_item["filter_action"],
        }
    )

    params.update(self._keyword_params(add_key
    + delete_key))
    response = self._call_api(
        "put",
    f"/api/v2/filters/{filter_item['id']}",
    params=params
    )
    response["added"] = add_keywords
    response["deleted"] = delete_keywords
    return response
```

Generated Docstring

This function synchronizes local and remote keywords for a filter by comparing provided keywords with existing ones. It identifies keywords to add and remove, logs changes, constructs API parameters, and updates the filter via a PUT request. The response includes added/deleted keywords to confirm synchronization.

```
def attention_mlp_stack(inputs, out_channel,
    num_heads, depth, key_dim, attn_ratio,
    mlp_ratio, strides, stack_drop=0,
    activation="hard_swish", name=""):

    nn = inputs
    embed_dim = nn.shape[-1]
    stack_drop_s, stack_drop_e = stack_drop
    if isinstance(stack_drop, (list, tuple))
    else [stack_drop, stack_drop]
    for ii in range(depth):
        block_name = name +
    "block{}_".format(ii + 1)
        drop_rate = stack_drop_s +
    (stack_drop_e - stack_drop_s) * ii /
    depth
        nn =
    res_mhsa_with_multi_head_position(nn,
    embed_dim, num_heads, key_dim,
    attn_ratio, drop_rate,
    activation=activation, name=block_name)

    if mlp_ratio > 0:
    [...]
    if mlp_ratio > 0:
        nn = res_mlp_block(nn, mlp_ratio,
    drop_rate, activation=activation,
    name=block_name + "mlp_")

    return keras.layers.Activation("linear",
    name=name + "output")(nn)
```

Generated Docstring

This function constructs a neural network stack with multi-head attention and MLP blocks. It applies a series of attention mechanisms (res_mhsa_with_multi_head_position) and optional MLPs (res_mlp_block) across 'depth' layers, adjusting dropout rates progressively. The stack supports configurable heads, key dimensions, ratios, and activation functions, returning a linear output layer.

These examples demonstrate that generated docstrings faithfully capture the code: the first enhances a minimal existing docstring with concise descriptions of keyword synchronization logic, while the second provides a detailed description of the multi-layer attention–MLP architecture for a function that originally lacked a docstring.

For math-related tasks, we use the following prompt format:

Prompt

Use the following pre-defined Lean 4 dependencies:
{dependencies}

Based on the context and the problem description, generate a single, syntactically correct Lean 4 formal statement that accurately captures the problem's meaning.

Problem Description:
{problem description}

For code-related tasks, we employ a parallel prompt structure:

Prompt

Use the following pre-defined functions:
{pre-defined functions}

Based on the context and the problem description, generate a syntactically correct function implementation that accurately captures the problem's meaning.
Problem Description:
{problem description}

G Dependency Tree Extraction Algorithm

The dependency tree extraction process is formally described in Algorithm 1. The algorithm iterates through all files within a given directory, systematically identifying user-defined functions, including both class methods and standalone functions. We then analyse inter-file and inter-class dependency relationships. Through static analysis of function call patterns, we construct a directed dependency graph where each function call establishes a parent-child relationship. The calling function serves as the parent node, while the called function becomes the child node.

H Extending TopoAlign to other Formal Languages

Formal statements in other proof assistants (e.g., Coq and Isabelle) share structural similarities with Lean. Below we show Isabelle and Coq versions of the informal statement: *Prove the ceiling of the square root of 27 minus the floor of the square root of 26 equals 1.*

Isabelle:
theorem ceil_sqrt27_minus_floor_sqrt26:
"ceiling (sqrt 27) - floor (sqrt 26) = (1::int)"

Coq:
Theorem ceil_sqrt27_minus_floor_sqrt26 :
(up (sqrt 27) - Z.floor (sqrt 26))%Z = 1%Z.

While these formal languages show similar structural characteristics to the decomposed code data in TopoAlign, supporting multiple formal languages would require either training dedicated models for each target language or constructing a mixed pre-training corpus that incorporates multiple formal languages. Although this direction offers promising practical applications, it lies beyond the scope of the present work.

Algorithm 1 Function Call Dependency Analysis

Input: Python project directory
Output: Dependency graph of function calls
Data Structures:
$function_definitions \leftarrow \{\}$ ▷ Record defined functions
$imports \leftarrow \{\}$ $object_types \leftarrow \{\}$ ▷ Imported names → full paths; Object names → class names; Save these two to track their parent function definitions or the file where they're defined
ANALYZEFILE(f) for all Python files f in directory (recursively)
procedure ANALYZEFILE(file_path)
    Parse AST from $file_path$
    Initialize $function_calls \leftarrow \{\}$, $object_types \leftarrow \{\}$
    for all nodes n in AST **do**
        if n is **ClassDef then**
            Track current class context ▷ Class defined function
        else if n is **Import/ImportFrom then**
            Record $imports[alias] \leftarrow$ full module path ▷ Function from other files
        else if n is **FunctionDef then**
            Register function with $current_class.name$ if applicable
            Initialize $function_calls[name] \leftarrow []$ ▷ Save its dependency
        else if n is **Assign** with constructor call **then**
            Map $object_types[var] \leftarrow$ class name ▷ Record the codes that initialise this function
        else if n is **Call then** ▷ Record the codes that call this function
            Resolve full function name (direct call($obj()$), method call($obj.method()$), or imports $imports[name]$
            Append to $function_calls[current_func]$ ▷ The list to record the dependency information
        end if
    end for
end procedure
procedure BUILDDEPENDENCYGRAPH
    Filter to keep only project-internal calls
    Detect recursive calls ($f \rightarrow f$) ▷ Self-recursive call
    Topological Sort: ▷ BFS search
    Construct nested call tree from sorted order
    Insert recursion markers at tree root if needed
end procedure