SITA: A Framework for Structure-to-Instance Theorem Autoformalization

Chenyi Li[1], **Wanli Ma**[2], **Zichen Wang**[1], **Zaiwen Wen**[2*]

[1]School of Mathematical Sciences, Peking University
[2]Beijing International Center for Mathematical Research, Peking University
lichenyi@stu.pku.edu.cn, wlma@pku.edu.cn, princhernwang@gmail.com, wenzw@pku.edu.cn

Abstract

While large language models (LLMs) have shown progress in mathematical reasoning, they still face challenges in formalizing theorems that arise from instantiating abstract structures in concrete settings. With the goal of auto-formalizing mathematical results at the research level, we develop a framework for structure-to-instance theorem autoformalization (SITA), which systematically bridges the gap between abstract mathematical theories and their concrete applications in Lean proof assistant. Formalized abstract structures are treated as modular templates that contain definitions, assumptions, operations, and theorems. These templates serve as reusable guides for the formalization of concrete instances. Given a specific instantiation, we generate corresponding Lean definitions and instance declarations, integrate them using Lean's typeclass mechanism, and construct verified theorems by checking structural assumptions. We incorporate LLM-based generation with feedback-guided refinement to ensure both automation and formal correctness. Experiments on a dataset of optimization problems demonstrate that SITA effectively formalizes diverse instances grounded in abstract structures.

Code — https://github.com/chenyili0818/SITA

1 Introduction

Recent advances in large language models (LLMs) have demonstrated impressive capabilities in solving mathematical problems and generating natural language proofs (Ahn et al. 2024; Welleck et al. 2021). However, such informal outputs often lack the formal rigor required for verification by proof assistants. To address this limitation, interactive theorem provers such as Lean (De Moura et al. 2015), Coq (Huet, Kahn, and Paulin-Mohring 1997), and Isabelle (Nipkow, Paulson, and Wenzel 2002) have been developed to rigorously validate each step of a proof. These systems enhance the soundness and reliability of machine-generated mathematical reasoning (Yang et al. 2024).

Several recent efforts have explored the use of LLMs to construct formal proofs from given formal statements. Two main paradigms have emerged: stepwise generation and whole-proof generation. The stepwise approach, exemplified by Leandojo (Yang et al. 2023), decomposes the proof process into premise selection and tactic generation, modeling proof search as a sequence of local decisions. BFS-prover (Xin et al. 2025) utilizes best-first tree search in generation. In contrast, whole-proof generation attempts to synthesize an entire proof in a single pass, typically by first producing a rough draft and then refining it into a valid proof (Jiang et al. 2023). These approaches often integrate natural language reasoning with formal symbolic reasoning, and are further improved through reinforcement learning techniques via expert iterations (Xin et al. 2024; Ren et al. 2025; Wang et al. 2025; Lin et al. 2025).

In parallel, the task of autoformalization, i.e. translating mathematical problems stated in natural language into formal statements, has also received increasing attention. Early approaches leverage few-shot prompting (Brown et al. 2020) to perform this translation (Wu et al. 2022; Azerbayev et al. 2023). Building on this foundation, subsequent work improves output quality through sampling strategies that select optimal outputs from multiple generations (Li et al. 2024b; Poiroux et al. 2025), or by retrieving related theorems from formal libraries (Liu et al. 2025a). While these methods are relatively easy to implement, their effectiveness remains limited by the capabilities of the underlying language model. Training-based approaches have also been explored for statement formalization (Jiang, Li, and Jamnik 2023). The Lean Workbook (Ying et al. 2025) introduces an iterative data generation and filtering pipeline to formalize problems from online math forums in Lean 4. ATLAS (Liu et al. 2025b) further improves formalization performance through expert iteration and knowledge distillation.

In this paper, we focus on a specific and widely used form of mathematical reasoning: deriving concrete instances from abstract structures. This reasoning pattern involves applying abstract theorems, defined over general mathematical structures, such as convergence guarantees of gradient descent on convex functions, to derive instance-specific results, such as the convergence of gradient descent for ridge regression. While this form of reasoning might seem straightforward, verifying that a specific instance satisfies the assumptions of an abstract theorem often demands non-trivial effort. This process may involve reformulating definitions, proving auxiliary lemmas, or performing symbolic manipulations, and in some cases, it occupies entire sections of theoretical papers (Duan et al. 2020). Although this reasoning paradigm is

*Corresponding author

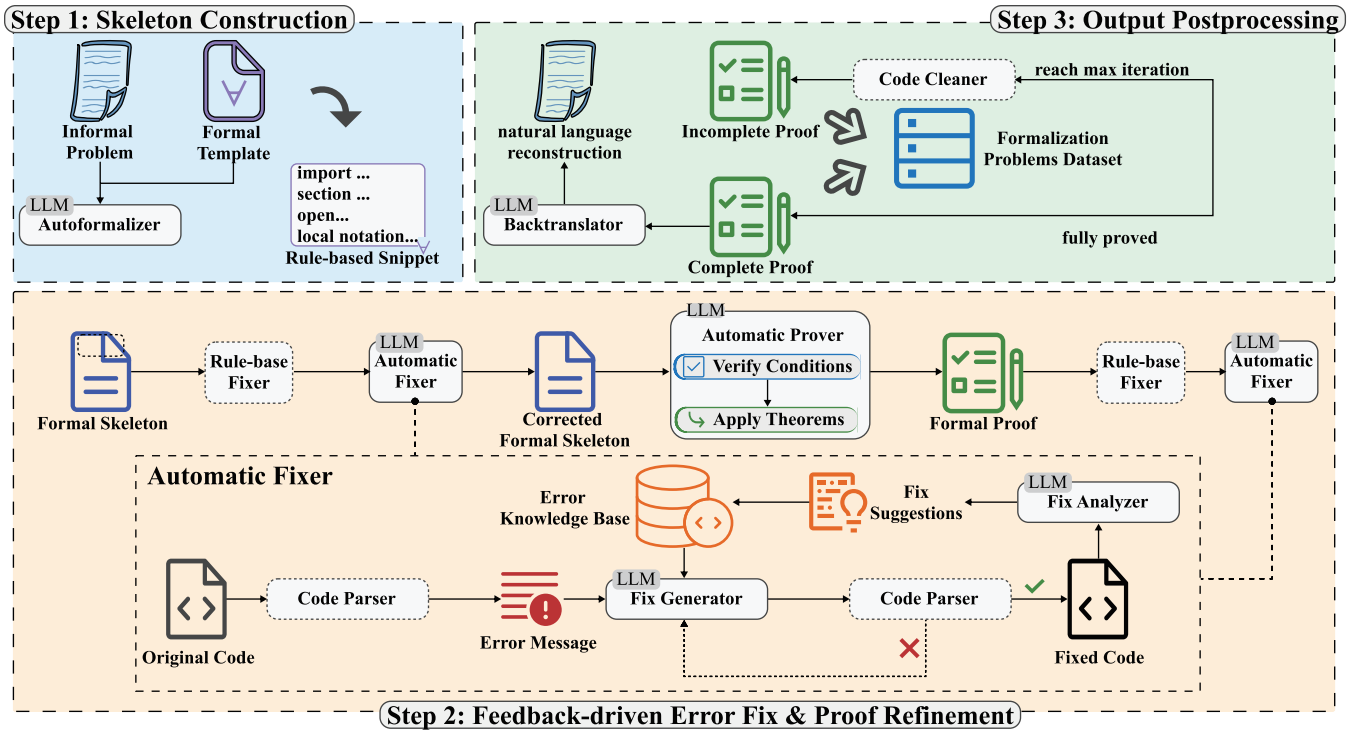

Figure 1: Overall pipeline of SITA

common in algorithmic mathematics and research-level papers (Bao et al. 2014), it heavily relies on expert intuition and manual work. It has not been systematically explored in the context of automated formalization yet. Our goal is to automate this process using the Lean proof assistant by translating natural language statements into formal definitions, verifying structural assumptions, and applying general theorems to construct formal proofs.

While most prior formalization research has focused on Olympiad-style mathematical problems, our study targets a broader and practically important form of reasoning: structure-to-instance theorems, with the goal of autoformalizing mathematical results at the research level. We introduce SITA, an end-to-end automated pipeline (see Figure 1), which leverages LLMs to formalize structure-to-instance reasoning end-to-end. Given a formalized abstract structure and an informal instance description, the pipeline generates instance-specific formal definitions, verifies structural conditions, and applies the abstract result to produce a complete formal proof. This pipeline enables the reliable reuse of high-level mathematical theories in domain-specific applications, supporting the construction of modular and verifiable formal reasoning workflows. Our framework is applicable to domains such as optimization, signal processing, and algebraic computation, and contributes toward the development of scalable, theorem-grounded formal libraries.

Our main contributions are as follows.

1) We define the structure-to-instance reasoning task by identifying relevant mathematical structures, their concrete instances, and the key sub-tasks involved to apply general results to specific problems.

2) We propose SITA, a framework that leverages LLMs to automatically generate formal definitions, lemmas, and theorems for instance-specific problems. An iterative refinement and error-feedback mechanism–guided by a growing knowledge base–enhances both the reasoning accuracy and the quality of generated formal code.

3) We construct a benchmark of optimization problems grounded in real-world applications to evaluate the proposed framework. Experimental results demonstrate that SITA successfully generates formal definitions, instantiates abstract theorems, and performs assumption verification.

2 The Structure-to-Instance Problem

2.1 Mathematical Structures and Their Instances

We mainly focus on operations defined over abstract mathematical structures. Following the approach (Baanen 2022) of bundling parameters and conditions into unified structures, a mathematical structure together with its associated operations is defined as follows.

Definition 1 (Mathematical Structures with Operations) *A* mathematical structure *is a four-tuple*

$$\mathcal{S} = \langle \mathcal{D},\ \mathcal{O},\ \mathcal{C},\ \mathcal{T} \rangle.$$

- $\mathcal{D}$ *(*Definitions*): Primitive notions or axioms introducing the fundamental objects of the theory.*
- $\mathcal{O}$ *(Operations): Computational procedures, symbolic operations, or constructive rules built on* $\mathcal{D}$*, serving as the algorithmic or operational core of the structure.*

These may include algorithms, transformation rules, or abstract procedures defined over the definitions.

- $\mathcal{C}$ *(*Conditions*): Assumptions imposed on elements in* $\mathcal{D}$ *and* $\mathcal{O}$ *to guarantee the correctness or performance.*
- $\mathcal{T}$ *(*Theorems*): Theorems and lemmas concerning the behavior of* $\mathcal{O}$*, derived from the definitions in* $\mathcal{D}$ *under the assumptions in* $\mathcal{C}$*.*

Such abstract structures are ubiquitous in mathematics. We provide an illustrative example below, with additional cases in the Appendix B. Operations as optimization algorithms are mainly studied in this paper.

Example 1 *The gradient descent method for unconstrained optimization can be expressed as* $\mathcal{S}_{\mathrm{GD}} = \langle \mathcal{D}, \mathcal{O}, \mathcal{C}, \mathcal{T} \rangle$.

- $\mathcal{D}$*: The unconstrained optimization problem is defined as* $\min_x f(x)$*, where* $f : \mathbb{R}^n \to \mathbb{R}$*.*
- $\mathcal{O}$*: The update scheme of the gradient descent method to solve the problem is* $x_{k+1} = x_k - \eta_k \nabla f(x_k)$*.*
- $\mathcal{C}$*: Function* $f(x)$ *is smooth and convex;* ∇f *is* L*-Lipschitz continuous;* $0 < \eta_k < 1/L$*.*
- $\mathcal{T}$*: A collection of theoretical results and guarantees related to the operations in* $\mathcal{O}$*, including:*
  - *the descent lemma;*
  - *sublinear convergence theorems.*

A concrete instance $\mathbb{I}$ of an abstract structure $\mathcal{S}$ is formed by specifying concrete realizations of its components, the definitions $\mathcal{D}$ and operations $\mathcal{O}$, within a specific problem domain. Although the abstract structure remains fixed, it can generate a wide range of instances that share the same formal framework but differ in the functions or variables used. Applying $\mathcal{S}_{\mathrm{GD}}$ to logistic regression or least-squares problems produces distinct instances with different objectives, while preserving similar structural properties.

Theoretical results in $\mathcal{T}$ can be applied to a concrete instance $\mathbb{I}$, provided that the conditions in $\mathcal{C}$ are satisfied. Verifying these conditions is often simpler than proving the results directly, as it only requires checking the set of assumptions. However, some verifications, such as the Kurdyka–Łojasiewicz (KL) property (Bolte, Sabach, and Teboulle 2014) in nonconvex optimization, can be challenging. The use of abstract structures enables the modular reuse of theoretical results across diverse instances, supporting the systematic generation of theoretical results.

2.2 Autoformalization Targets

Given a formally defined abstract structure $\mathcal{S}$ and a natural language description of a concrete instance where $\mathcal{S}$ applies, we consider the task of auto-formalizing such instances $\mathbb{I}$. Each instance includes its own problem data, operation and the corresponding theorems. We call this process the autoformalization of structure-to-instance theorems. The main idea is to use the formalized structure to guide the formalization of instances in application domains, allowing theoretical results to be reused. We use Lean as the theorem prover in this paper. A basic introduction to formalization using Lean is provided in Appendix A.

The structure-to-instance autoformalization paradigm consists of the following two steps. An illustration is also provided in Figure 2.

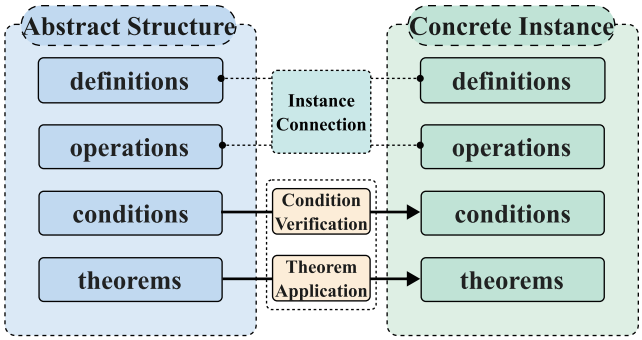

Figure 2: Illustration of structure-to-instance formalization.

1. **Instance specification and structural integration.** We begin by formally specifying the instance-level problem data and associated operations, instantiating the abstract components $\mathcal{D}$ and $\mathcal{O}$. The instance is then integrated into the abstract framework $\mathcal{S}$ through `instance` declarations in Lean, which certify that the concrete problem provides definitions, operations, and structural properties compatible with those of $\mathcal{S}$. This alignment enables the generic definitions, operations, and theorems of $\mathcal{S}$ to be uniformly applied to the concrete setting.
2. **Theorem instantiation and verification.** Once structural alignment is established, we verify that the instance satisfies the conditions in $\mathcal{C}$, concrete assumptions on the data and operations required for sound application of the theorems in $\mathcal{T}$. After proving these assumptions, we instantiate the abstract theorems in $\mathcal{T}$ to derive concrete results, thus avoiding redundant proof efforts.

The structure-to-instance formalization paradigm embodies a central pattern in mathematical reasoning: once a property is established under general assumptions, it can be systematically instantiated across all problems that meet those conditions. Our framework systematizes this reasoning at scale, automating the process of generating verified proofs for a wide variety of instances from a single abstract theory.

This task fundamentally differs from traditional autoformalization efforts, which primarily focus on translating informal theorems into formal statements. Such approaches often neglect the formalization of definitions and concrete instances. In contrast, we consider a setting where the abstract structure $\mathcal{S}$ is already formalized, and the goal is to automate its instantiation: specializing defitions $\mathcal{D}$ and operations $\mathcal{O}$ to domain-specific elements, verifying that the structural conditions $\mathcal{C}$ are satisfied, and applying general theorems $\mathcal{T}$ to derive instance-level guarantees. This process enables the systematic reuse of abstract formal results across diverse application domains.

Moreover, this framework facilitates the creation of large, composable datasets of verified application theorems. These datasets support practical applications such as safety verification and certified manipulation, while also providing essential training data for future machine learning systems in formal mathematics, bridging symbolic reasoning and automated synthesis in a principled and extensible manner.

2.3 Example: Lasso

To illustrate the structure-to-instance autoformalization paradigm shown in Fig. 2, we present an example involving the proximal gradient method (Parikh and Boyd. 2014) applied to the Lasso problem (Tibshirani 1996). We only show part of the code and highlight the main idea of our structure-to-instance formalization pipeline. The full code and detailed explanations are provided in the Appendix C. The code builds on Mathlib (mathlib Community 2020) and Optlib (Li et al. 2024a, 2025).

1. Abstract Structure $\mathcal{S}$ We first obtain the formalized definitions of the abstract structure. In this setting, $\mathcal{D}$ as the composite optimization problem $\min_x \psi(x) = f(x) + h(x)$ is formalized as below.

```
class composite_pro (f h : E → ℝ)
def composite_pro.tar (_ :
    composite_pro f h) := f + h
```

The corresponding optimization algorithm, the proximal gradient method, is given as follows.

```
class pg (pro : composite_pro f h) (x₀ :
    E) := ...
```

Assuming appropriate conditions $\mathcal{C}$ (e.g., convexity, Lipschitz smoothness), the convergence theorem $\mathcal{T}$ is stated as:

```
theorem pg_method_converge (conditions):
∀ k, (pro.tar (alg.x k) - pro.tar xm)
  ≤ 1 / (2 * k * alg.t) * || x₀ - xm ||
    ^ 2 :=
```

2. Concrete Instance $\mathbb{I}$: Lasso The Lasso problem minimizes $\frac{1}{2}\|Ax-b\|^2 + \mu\|x\|_1$. The proximal gradient method solves the Lasso problem with $(x_{k+1})_i = \text{sign}(z_i) \cdot \max\{|z_i| - t\mu, 0\}$, where $z = x_k - tA^\top(Ax_k - b)$. We expect to auto-formalize the corresponding structures as:

```
class Lasso_pro (A b μ) := ...
class pg_Lasso (pro : Lasso_pro A b μ)
    (x₀ : E) := ...
```

Instances below are used to link the Lasso problem class with the abstract class using Lean's instance mechanism, as illustrated in the central arrows of Figure 2:

```
instance Lasso_pro.composite_pro :
    composite_pro f g := ...
instance pg_Lasso.pg : pg pro x₀ := {
  t := self.t, x := self.x, initial := ...
    , update := ...}
```

3. Theorem Transfer via Instance By verifying the assumptions in `pg_method_converge`, we obtain the Lasso-specific convergence result through reuse:

```
theorem Lasso_convergence : ∀ (k : ℕ+),
    (pro.target (alg.x k) - pro.target
    xm) ≤ 1 / (2 * k * alg.t) * || x₀ -
    xm || ^ 2 := by sorry
```

This example demonstrates the complete mapping from an abstract structure $\mathcal{S}$ to a concrete instance $\mathbb{I}$ as shown in Figure 2. Instance declarations enable modular reuse of definitions and theorems, facilitating scalable formalization of optimization algorithms.

3 The SITA Framework

The **SITA** pipeline, illustrated in Figure 1, provides an end-to-end framework for structure-to-instance theorem autoformalization. It transforms a natural language description of a concrete problem into a verified Lean file by aligning the underlying concepts with a formalized abstract structure. The pipeline comprises three main stages: (1) skeleton construction, (2) feedback-driven error fix and proof refinement, and (3) output postprocessing. Each stage integrates large language models[1] with Lean's type system to ensure both automation and formal soundness.

3.1 Skeleton Construction

Given an *informal problem description*, the process begins by identifying a suitable abstract structure, either automatically or via user input, that aligns with the concrete setting. Each abstract structure is represented as a reusable *formal template* consisting of definitions, assumptions, and theorems. Using one-shot prompting, the system generates the core formal components of the instance, including problem-specific definitions, operation classes, and target theorem statements. It also produces *instance lemmas* that declare Lean typeclass instances, connecting the concrete definitions to the abstract structure and certifying that the necessary structural conditions are satisfied. In addition, the system incorporates *rule-based snippets* from the formal template, such as `import` statements, `section` headers, and `open` declarations, providing a structured formal context that guides and constrains the generation process. This helps improve correctness and ensures compatibility with existing formal libraries. The output of this stage is a *formal skeleton*, i.e. a Lean file containing the complete problem setup and theorem declarations, prepared for subsequent refinement.

3.2 Error Fix and Proof Refinement

Error Fix To address errors in automatically generated Lean code, we employ a feedback-driven correction framework implemented in the *error fix* stage. This component integrates two complementary steps: a *rule-based fixer*, which applies deterministic edits targeting common syntactic and structural issues, and an *automatic fixer*, which leverages Lean's type-checking diagnostics and a self-updating *error knowledge base* to suggest and validate adaptive repairs.

The first step is a static, rule-based correction module responsible for addressing common syntactic and structural issues. It performs a series of deterministic transformations, including formatting declarations, standardizing overloaded or non-canonical notations, removing unused hypotheses or empty section blocks, and enforcing conventions consistent with Lean's standard libraries. The correction rules are implemented through symbolic rewriting and pattern matching, allowing efficient and general-purpose repair prior to more context-sensitive analysis.

Complementing the static step is an iterative feedback-driven correction module powered by Lean's type checker and retrieval-augmented (Lewis et al. 2020; Fan et al. 2024) prompt construction. Given an error message e, the system queries an evolving error knowledge base $\mathcal{K}$, which stores structured correction strategies, illustrative fix examples, and references to relevant theorems or tactics. The retrieved entries $\mathcal{K}(e)$ are assembled into a custom prompt tai-

[1]Detailed prompt templates are provided in Appendix D.2.

lored to the specific error and passed to the language model. For instance, if a definition involves incorrect usage of a term T, the prompt includes the full statement of T along with common usage patterns. When the error involves an undefined or non-canonical lemma, the retrieval module suggests similar alternatives from Lean's libraries, which are automatically incorporated into the prompt context.

After each correction attempt, the updated code is recompiled and rechecked by Lean. If new errors are encountered, the feedback loop continues. For each resolved errors, the system logs the original error message, faulty code, and corrected code. These logs are processed by a secondary model to generate new *fix suggestions*. Both the logs and the suggestions contribute to updating $\mathcal{K}$. This feedback mechanism progressively improves correction capabilities by capturing and generalizing emerging error patterns. The iteration terminates once the file type-checks successfully or a predefined retry limit is reached.

This hybrid mechanism, combining symbolic rewriting, retrieval-augmented generation, and dynamic knowledge adaptation, yields a robust correction system capable of transforming flawed or incomplete formalizations into valid Lean code. The iterative process can be viewed as an explicit form of chain-of-thought reasoning (Wei et al. 2022) in the context of autoformalization. By treating the sequence of errors, prompts, and corrections as a structured evolving context, the system enables the model to reflect on prior failures and refine its responses accordingly, resulting in more accurate and coherent formalizations.

Proof Refinement After obtaining correct formal definitions and instance lemma statements, the next objective is to eliminate all remaining `sorry` placeholders by constructing valid Lean proofs. To this end, a whole-proof generation pipeline is employed. For each `sorry`, the system extracts the corresponding local environment, including hypotheses, definitions, and contextual information, and feeds it into the language model. Then, the model attempts to generate a proof term appropriate for the given goal.

In most cases, the generated proof attempts to verify that a concrete instance satisfies the assumptions of an abstract lemma or theorem. While these subgoals are typically less complex than proving the full statement, they can still be challenging, particularly in domain-specific contexts. To address this, the system maintains a retry counter to regulate repeated attempts. The error knowledge base is also leveraged at this stage to guide correction when a generated proof fails to type-check. Relevant proof construction suggestions, known tactic patterns, and examples of common pitfalls are retrieved and integrated into the prompt to improve model robustness. This mechanism reuses the infrastructure developed in the error correction stage, adapted here for proof synthesis. If the maximum number of attempts is reached without success, the pipeline falls back to a partial completion strategy: the language model attempts to construct as much of the proof as possible, retaining `sorry` placeholders for unresolved fragments.

3.3 Output Postprocessing

In the final stage, the system compiles the formalization results into a clean, verifiable Lean file, free of type errors, extending the approach of (Ospanov, Farnia, and Yousefzadeh 2025) to handle both statements and proofs. For files that still contain errors, a hybrid postprocessing strategy is applied. First, rule-based techniques are utilized to patch the proof context by inserting appropriate `sorry` placeholders where necessary. If this fails to produce a well-typed file, an LLM-based fallback rewrites the file into a harmless, error-free version. If all proof goals are successfully discharged, the system outputs a complete formal artifact. Otherwise, any remaining incomplete components are explicitly marked and logged for downstream analysis or future refinement.

To support interpretability and broader applicability, SITA integrates a *backtranslator* module that maps formal Lean constructs, such as definitions, assumptions, and theorems, back into natural language. This step facilitates human verification, improves documentation, and enables data augmentation. The *reconstruction* outputs are aligned with the original informal problem descriptions, laying the groundwork for training bi-directional models that bridge formal and informal mathematical reasoning.

All intermediate artifacts, including incomplete proofs, corrected code fragments, Lean error traces, and natural language reconstruction outputs, are stored in a structured dataset. This archive supports fine-grained evaluation of model performance and serves as a targeted training resource for advancing proof generation, error correction, and formal-informal alignment in future models.

4 Numerical Experiments

4.1 Experimental Setup

To assess the effectiveness of the proposed SITA paradigm, we conduct experiments on a collection of research-level mathematical problems that naturally conform to the structure-to-instance formalization setting. A particularly representative class of such problems arises in optimization, where many concrete applications can be viewed as instantiations of abstract algorithmic frameworks. By formalizing these instances, we demonstrate that SITA can correctly and efficiently generate Lean definitions and proofs that link concrete cases with their underlying abstract structures. While our current experiments focus on optimization, the proposed framework is not domain-specific. It can be extended to other areas of mathematics where formal abstraction and reusable operational structure play a central role.

Dataset We construct a benchmark dataset comprising 42 representative optimization problems collected from widely used textbooks in numerical optimization. The problems span a broad spectrum of settings, including both constrained and unconstrained formulations, as well as convex and nonconvex objectives, thereby covering key categories commonly studied in the optimization literature. Each problem in the dataset is selected for its reliance on a standard optimization algorithm, including gradient descent (GD), proximal gradient (PGM), Nesterov's acceleration (Nesterov 1983), block coordinate descent (BCD)

(Bolte, Sabach, and Teboulle 2014), and the alternating direction method of multipliers (ADMM) (Fazel et al. 2013). These algorithms have been formally verified in the Lean library Optlib, and we build on their existing formalizations by adopting them as reusable, abstract components. More dataset information can be found in Appendix F. Here are some typical examples.

- Sparse recovery in signal processing (PGM)

$$\min_x \frac{1}{2}\|Ax - b\|^2 + \mu\|x\|_1.$$

- Total variation denoising (ADMM)

$$\min_{x,z} \frac{1}{2}\|x - y\|^2 + \|z\|_1, \quad \text{s.t.} \quad Dx = z.$$

Base Model We use DeepSeek-R1 (DeepSeek-AI et al. 2025a) and DeepSeek-V3 (DeepSeek-AI et al. 2025b) as base models in our autoformalization pipeline. Unlike theorem proving specific LLMs, our structure-to-instance autoformalization task requires not only formal reasoning but also understanding informal language, identifying abstract structures, and generating Lean definitions and proofs. These models are chosen for their strong language understanding and multi-step instruction-following abilities. Model hyperparameters are listed in Appendix D.3.

4.2 Results and Analysis

With no former autoformalization work related to whole file generation, we propose the following four aspects to evaluate the output: (1) Definition (syntax-correct definitions without sorry); (2) Theorem (syntax-correct theorem statements, possibly with sorry); (3) Instance (syntax-correct instance declarations, possibly with sorry); (4) Full file (considering all four aspects above). To obtain the score, we interact with the Lean environment to collect the corresponding error messages, assigning each message to its associated definition, theorem, or instance. The success ratio is defined as the proportion of syntactically correct definitions, theorem statements, and instance declarations relative to their respective totals. Besides evaluation through type check, majority voting is also used to examine the semantic correctness. We follow the settings in (Liu et al. 2025a) and use DeepSeek-V3 with temperature $T = 0.7$ with 16 rounds[2]. The model is required to rate the output from the aspects of problem formalization, algorithm correctness, update scheme explicitness, theoretical analysis and proof completion rate. The score ranges from 0 to 100.

We compare the completion rate of SITA under two different base models with that of direct generation. Existing proof generation and autoformalization models are not designed to handle the structured formalization task addressed in our work, which involves generating definitions, instances, and theorems in a unified pipeline. To the best of our knowledge, there is currently no dedicated method capable of performing such structure-to-instance autoformalization. Therefore, we limit our comparison to general-purpose LLMs as a baseline for assessing the effectiveness of our approach. The results are shown in Table 1. All the results are under generation with 3 attempts and fix iteration with 3 iterations. A case study of both successful and failed cases generated by SITA is provided in Appendix G.1. More detailed results are given in Appendix H.

Model	Def	Thm	Instance	File	MV
Direct-V3	27.9%	28.0%	22.8%	0.0%	50.2
Direct-R1	62.8%	25.6%	25.7%	0.0%	46.0
SITA-V3	91.0%	86.7%	90.8%	27.2%	66.1
SITA-R1	93.8%	95.6%	95.4%	57.14%	76.9

Table 1: Formalization completion rate comparison. **Direct-V3**: direct generation with DeepSeek-V3. **Direct-R1**: direct generation with DeepSeek-R1. **SITA-V3**: our framework using DeepSeek-V3. **SITA-R1**: our framework using DeepSeek-R1. **Def**, **Thm**, **Instance** and **File** denotes four criteria for the evaluation of entire file generation. **MV**: results from majority voting.

As shown in Table 1, our proposed SITA framework significantly outperforms the direct generation baselines across all evaluation dimensions. Notably, SITA-R1 achieves an overall file-level success rate of 57.14%. In contrast, direct generation methods, whether using DeepSeek-V3 or DeepSeek-R1, fail to produce any fully correct formalization at the file level, showing the substantial difficulty of end-to-end file level automatic formalization. One of the challenge lies in effectively leveraging newly synthesized concepts to instantiate appropriate type-class patterns and formulate theorems. Although current models are partly capable of generating definitions, they often fall short when integrating these definitions into semantically coherent lemmas or instances.

For those files fail to generate fully correct files, we examine the success rate of the statement generation. In addition to analyzing definitions and statements, we also study the generation of proofs in cases where the files are correctly generated. Proofs are evaluated only when the generation of definitions and statements succeeds. This is because Lean's type-checking mechanism cannot reliably report errors in proofs if the definitions or statements contain type errors. As shown in Table 2, we report two key metrics: the syntactic correctness rate of the definitions and statements in files that contain generation errors (SC), and the proof success rate in files that are generated without any syntax-level problems (PS). We report statistics including the number of definitions and theorems, and average file length for each class.

From the results, we observe that even among the files that are not fully correct, a significant portion of their component are still syntactically valid.

This suggests that most of the errors are concentrated in a small subset of complex or ambiguous definitions or statements, while the majority of the content is already valid and usable. Therefore, with SITA as the backbone, the cost of human intervention can be reduced. Users only need to focus on refining a few critical pieces rather than writing the entire file from scratch.

[2]Detailed prompt templates are provided in Appendix D.2.

Class	SC	PS	DM	TM	FL
GD	83.36%	53.77%	4	7	96
PGM	98.1%	62.96%	6	9	116
Nesterov	97.8%	63.28%	6	8	119
BCD	88.82%	50.55%	9	17	194
ADMM	85.02%	20.00%	6	8	121
Overall	90.72%	51.23%	6.21	9.93	129.83

Table 2: Syntactic correctness rate and proof completion rate. **SC**: syntactic correctness rate of statements in the failed cases; **PS**: proof completion rate in the success cases; **DM**: total number of definitions; **TM**: total number of lemmas and theorems; **FL**: average file length (lines).

On the other hand, for files that are syntactically correct, the proof success rate exceeds 50% in the simpler classes. This indicates that the model is capable of completing a substantial portion of proofs autonomously, especially for elementary or structurally well-defined lemmas. The remaining failures are mostly associated with intricate reasoning steps or subtle dependencies, which are inherently harder for current models to resolve without fine-grained guidance.

Generated Formal Problems From this structure to instance autoformalization procedure, we generate 88 correctly complied files and 449 theorems and 222 of them are with correct proofs. Most of these problems concentrates on the properties of concrete functions, such as the convexity, the Lipschitz continuity, or the KL property. We reorganize the file and obtain a benchmark Opt-bench consisting of formalized problems focus on analysis and optimization.

4.3 Ablation Study

We conduct ablation studies to understand each component's contribution in the SITA framework. More concrete results can be found in Appendix H.

Procedure Ablation Study We study the effect of key modules in the SITA pipeline. As shown in Table 3, removing the linking error recovery module leads to only a moderate drop in both syntactic correctness and proof success, suggesting that the model possesses a certain capacity to adapt based on Lean's error messages. Retrieval-augmented error correction contributes to improves the stability and precision of error correction. Furthermore, disabling the proof refinement stage is associated with a decrease in performance, indicating that this component plays a role in improving partially correct roofs. The increase in file length also validates the effectiveness of proof refinement. In addition, removing in-class examples from the prompt (replaced with generic examples) leads to a decline in all metrics, demonstrating that in-context alignment plays a vital role in guiding the generation toward syntactically correct and semantically aligned outputs.

Time Consumption of Each Part We also compare the time consumption of each part. Correction steps dominate the runtime: the corrections for the backbone and the corrections for the proofs account for 56.5 % and 22.5 %. While the remaining three stages, harmless fixing, generation, and proof generation, each consume roughly the same, comparatively minor share of the total time.

Configuration	FS (%)	SC (%)	PS (%)	FL
w/o example	9.5%	74.1 %	42.5%	100
w/o link errs	45.8 %	83.4 %	49.8 %	131
w/o proof refine	57.14%	90.7%	23.45%	119
Full pipeline	57.14%	90.7%	51.23%	134

Table 3: Ablation study of pipeline stages. **FS**: File success compilation rate; **w/o example**: pipeline without an exemplar within the same category in the prompt, uses general examples instead. **w/o link errs**: the pipeline without correction tips retrieval from error knowledge base; **w/o proof refine**: the pipeline without secondary proof generation.

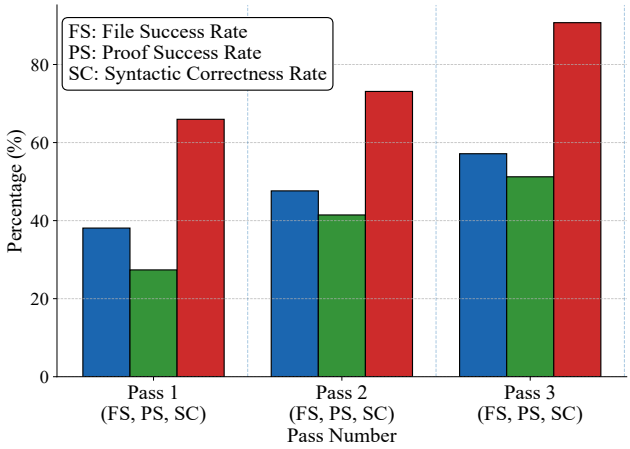

Figure 3: Evaluation performance across generation passes.

Generation Pass Ablation Study We study the impact of the number of pass rounds on three evaluation metrics: file success rate, SC rate and the PS rate. As shown in Figure 3, all three metrics improve with the number of passes. This indicates that the number of passes can influence the overall quality and correctness of the generated outputs.

5 Conclusion

We propose SITA, a novel pipeline for structure-to-instance theorem autoformalization, introducing a new paradigm of structured mathematical reasoning for research level mathematics. By aligning abstract formal structures with concrete problem instances, our pipeline enables scalable, modular proof reuse. Experiments on diverse optimization problems show its effectiveness in generating correct formal definitions, instance declarations, and verified proofs. This framework paves the way for reusable, verified libraries and structured datasets to support future advancements.

Acknowledgments

Z. Wen was supported in part by National Key Research and Development Program of China under the grant number 2024YFA1012903, the National Natural Science Foundation of China under the grant numbers 12331010 and 12288101, and the Natural Science Foundation of Beijing, China under the grant number Z230002.

References

Ahn, J.; Verma, R.; Lou, R.; Liu, D.; Zhang, R.; and Yin, W. 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*, 225–237. Association for Computational Linguistics.

Azerbayev, Z.; Piotrowski, B.; Schoelkopf, H.; Ayers, E. W.; Radev, D.; and Avigad, J. 2023. ProofNet: Autoformalizing and Formally Proving Undergraduate-Level Mathematics. arXiv:2302.12433.

Baanen, A. 2022. Use and Abuse of Instance Parameters in the Lean Mathematical Library. In Andronick, J.; and de Moura, L., eds., *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 4:1–4:20. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-252-5.

Bao, C.; Ji, H.; Quan, Y.; and Shen, Z. 2014. l0 Norm Based Dictionary Learning by Proximal Methods with Global Convergence. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Bolte, J.; Sabach, S.; and Teboulle, M. 2014. Proximal alternating linearized minimization for nonconvex and nonsmooth problems. *Mathematical Programming*, 146(1-2): 459–494.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; et al. 2020. Language Models are Few-Shot Learners. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M.; and Lin, H., eds., *Advances in Neural Information Processing Systems*, volume 33, 1877–1901. Curran Associates, Inc.

De Moura, L.; Kong, S.; Avigad, J.; Van Doorn, F.; and von Raumer, J. 2015. The Lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, 378–388. Springer.

DeepSeek-AI; Guo, D.; Yang, D.; Zhang, H.; Song, J.; Zhang, R.; et al. 2025a. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948.

DeepSeek-AI; Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; et al. 2025b. DeepSeek-V3 Technical Report. arXiv:2412.19437.

Duan, Y.; Wang, M.; Wen, Z.; and Yuan, Y. 2020. Adaptive Low-Nonnegative-Rank Approximation for State Aggregation of Markov Chains. *SIAM Journal on Matrix Analysis and Applications*, 41(1): 244–278.

Fan, W.; Ding, Y.; Ning, L.; Wang, S.; Li, H.; Yin, D.; Chua, T.-S.; and Li, Q. 2024. A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '24, 6491–6501. New York, NY, USA: Association for Computing Machinery. ISBN 9798400704901.

Fazel, M.; Pong, T. K.; Sun, D.; and Tseng, P. 2013. Hankel Matrix Rank Minimization with Applications to System Identification and Realization. *SIAM Journal on Matrix Analysis and Applications*, 34(3): 946–977.

Huet, G.; Kahn, G.; and Paulin-Mohring, C. 1997. The Coq proof assistant a tutorial. *Rapport Technique*, 178.

Jiang, A. Q.; Li, W.; and Jamnik, M. 2023. Multilingual Mathematical Autoformalization. arXiv:2311.03755.

Jiang, A. Q.; Welleck, S.; Zhou, J. P.; Lacroix, T.; Liu, J.; Li, W.; Jamnik, M.; Lample, G.; and Wu, Y. 2023. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *The Eleventh International Conference on Learning Representations*.

Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V.; Goyal, N.; Küttler, H.; Lewis, M.; Yih, W.-t.; Rocktäschel, T.; et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33: 9459–9474.

Li, C.; Wang, Z.; Bai, Y.; Duan, Y.; Gao, Y.; Hao, P.; and Wen, Z. 2025. Formalization of Algorithms for Optimization with Block Structures. arXiv:2503.18806.

Li, C.; Wang, Z.; He, W.; Wu, Y.; Xu, S.; and Wen, Z. 2024a. Formalization of Complexity Analysis of the First-order Algorithms for Convex Optimization. arXiv:2403.11437.

Li, Z.; Wu, Y.; Li, Z.; Wei, X.; Zhang, X.; Yang, F.; and Ma, X. 2024b. Autoformalize Mathematical Statements by Symbolic Equivalence and Semantic Consistency. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Lin, Y.; Tang, S.; Lyu, B.; Wu, J.; Lin, H.; Yang, K.; Li, J.; Xia, M.; et al. 2025. Goedel-Prover: A Frontier Model for Open-Source Automated Theorem Proving. arXiv:2502.07640.

Liu, Q.; Zheng, X.; Lu, X.; Cao, Q.; and Yan, J. 2025a. Rethinking and Improving Autoformalization: Towards a Faithful Metric and a Dependency Retrieval-based Approach. In *The Thirteenth International Conference on Learning Representations*.

Liu, X.; Bao, K.; Zhang, J.; Liu, Y.; Liu, Y.; Chen, Y.; Jiao, Y.; and Luo, T. 2025b. ATLAS: Autoformalizing Theorems through Lifting, Augmentation, and Synthesis of Data. arXiv:2502.05567.

mathlib Community, T. 2020. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, 367–381. New York, NY, USA: Association for Computing Machinery. ISBN 9781450370974.

Nesterov, Y. 1983. A method of solving a convex programming problem with convergence rate O (1/k** 2). *Doklady Akademii Nauk SSSR*, 269(3): 543.

Nipkow, T.; Paulson, L. C.; and Wenzel, M. 2002. A Proof Assistant for Higher-Order Logic. *Lecture Notes in Computer Science*.

Ospanov, A.; Farnia, F.; and Yousefzadeh, R. 2025. APOLLO: Automated LLM and Lean Collaboration for Advanced Formal Reasoning. arXiv:2505.05758.

Parikh, N.; and Boyd., S. 2014. *Proximal Algorithms*, volume 1. Foundations and Trends in Optimization.

Poiroux, A.; Weiss, G.; Kunčak, V.; and Bosselut, A. 2025. Improving Autoformalization using Type Checking. arXiv:2406.07222.

Ren, Z. Z.; Shao, Z.; Song, J.; Xin, H.; Wang, H.; Zhao, W.; et al. 2025. DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. arXiv:2504.21801.

Tibshirani, R. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58(1).

Wang, H.; Unsal, M.; Lin, X.; Baksys, M.; Liu, J.; et al. 2025. Kimina-Prover Preview: Towards Large Formal Reasoning Models with Reinforcement Learning. arXiv:2504.11354.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Ichter, B.; Xia, F.; Chi, E. H.; Le, Q. V.; and Zhou, D. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22. Red Hook, NY, USA: Curran Associates Inc. ISBN 9781713871088.

Welleck, S.; Liu, J.; Bras, R. L.; Hajishirzi, H.; Choi, Y.; and Cho, K. 2021. NaturalProofs: Mathematical Theorem Proving in Natural Language. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Wu, Y.; Jiang, A. Q.; Li, W.; Rabe, M.; Staats, C.; Jamnik, M.; and Szegedy, C. 2022. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35: 32353–32368.

Xin, H.; Ren, Z. Z.; Song, J.; Shao, Z.; Zhao, W.; Wang, H.; Liu, B.; Zhang, L.; Lu, X.; Du, Q.; Gao, W.; Zhu, Q.; Yang, D.; Gou, Z.; Wu, Z. F.; Luo, F.; and Ruan, C. 2024. DeepSeek-Prover-V1.5: Harnessing Proof Assistant Feedback for Reinforcement Learning and Monte-Carlo Tree Search. arXiv:2408.08152.

Xin, R.; Xi, C.; Yang, J.; Chen, F.; Wu, H.; Xiao, X.; Sun, Y.; Zheng, S.; and Shen, K. 2025. BFS-Prover: Scalable Best-First Tree Search for LLM-based Automatic Theorem Proving. arXiv:2502.03438.

Yang, K.; Poesia, G.; He, J.; Li, W.; Lauter, K.; Chaudhuri, S.; and Song, D. 2024. Formal Mathematical Reasoning: A New Frontier in AI. arXiv:2412.16075.

Yang, K.; Swope, A.; Gu, A.; Chalamala, R.; Song, P.; Yu, S.; Godil, S.; Prenger, R.; and Anandkumar, A. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Neural Information Processing Systems (NeurIPS)*.

Ying, H.; Wu, Z.; Geng, Y.; Wang, J.; Lin, D.; and Chen, K. 2025. Lean workbook: a large-scale Lean problem set formalized from natural language math problems. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc. ISBN 9798331314385.