Clarifying Before Reasoning: A Coq Prover with Structural Context

Yanzhen Lu[1] **Hanbin Yang**[1] **Xiaodie Wang**[1] **Ge Zhang**[1] **Biao Li**[1]

Chenxu Fu[1] **Chao Li**[1] **Yang Yuan**[1,2,†] **Andrew Chi-Chih Yao**[1,2,†]

[1]Shanghai Qizhi Institute, [2]IIIS, Tsinghua University
`{luyanzhen,yanghanbin,wangxiaodie,zhangge,libiao,fuchenxu,lichao}@sqz.ac.cn`
`yuanyang@tsinghua.edu.cn,andrewcyao@tsinghua.edu.cn`

Abstract

In this work, we investigate whether improving task clarity can enhance reasoning ability of large language models, focusing on theorem proving in Coq. We introduce a concept-level metric to evaluate task clarity and show that adding structured semantic context to the standard input used by modern LLMs, leads to a $1.85\times$ improvement in clarity score (44.5% $\rightarrow$ 82.3%). Using the general-purpose model `DeepSeek-V3`, our approach leads to a $2.1\times$ improvement in proof success (21.8% $\rightarrow$ 45.8%) and outperforms the previous state-of-the-art `Graph2Tac` (33.2%). We evaluate this on 1,386 theorems randomly sampled from 15 standard Coq packages, following the same evaluation protocol as `Graph2Tac`. Furthermore, fine-tuning smaller models on our structured data can achieve even higher performance (48.6%). Our method uses selective concept unfolding to enrich task descriptions, and employs a Planner–Executor architecture. These findings highlight the value of structured task representations in bridging the gap between understanding and reasoning.

1 Introduction

The mainstream approach to improving AI reasoning has largely focused on scaling model architectures, refining datasets, and employing various reinforcement learning techniques. While these combined strategies have undoubtedly made remarkable progress, they overlook a crucial dimension: whether the model truely **understands** the task at hand. Indeed, when a model fails to solve a task, it may not simply be due to insufficient reasoning ability; inadequate or incomplete task understanding could also play a crucial role.

Consider a scenario where a mathematical theorem references a symbol G without providing a clear definition. The surrounding context may offer some clues, but often not enough to fully disambiguate its meaning. As a result, the model can only make an educated guess — G might be a function, a graph, a constant, or a matrix — but its interpretation remains uncertain. As our experiments demonstrate, even when models generate plausible answers, they may rely on incomplete or ambiguous understanding of the input — achieving only 44.5% accuracy when asked to define concepts from raw Coq scripts.

Can we improve a model's reasoning ability by enhancing the **task description**? This direction is orthogonal to existing approaches based on data scaling and reinforcement learning, because the two

[†]Corresponding author.

can be easily combined: once the model better understands the task, it can start solving the task based on other techniques.

Clearly describing a task to the model is often seen as more of an art than a science—something that relies heavily on the intuition and experience of skilled prompt engineers. Given the diversity and ambiguity of real-world tasks, it is indeed challenging to define a universally optimal strategy for task formulation. However, in more formal and well-defined domains such as mathematics, we believe that constructing effective prompts can be approached more systematically. Specifically, we propose breaking the problem down into three questions in the Coq environment:

1. Can we assess how well a model understands a given task?
2. Can we improve the model's understanding by enriching the task description?
3. Does deeper understanding lead to better reasoning performance?

To address the first question, we propose evaluating the model's understanding by having it generate formal definitions of the relevant concepts—a metric we refer to as the **clarity score**. As shown in Table 1, models demonstrate limited conceptual understanding under standard task descriptions, achieving an average clarity score of just 44.5%. This suggests that, during reasoning, models often lack a clear grasp of the task at hand.

For the second question, we adopt the strategy of **concept unfolding** [38]. The idea is to recursively expand each concept with its definition. For example, a triangle may be defined as a closed figure with three sides a, b, c, each of which can further be defined. This process continues until we reach axiomatic definitions, i.e., numbers like 0, 1, 2, or operations like +, -, *.

Naturally, excessive unfolding can lead to redundancy and even hinder model performance. For example, consider defining G as a positive definite matrix. Expanding this into "a matrix with positive eigenvalues" may be helpful, but further decomposing it into elementary operations like addition or multiplication is unnecessary, if the model already has a firm grasp of linear algebra. Such over-expansion not only bloats the input with irrelevant detail, but may also distract the model from the core reasoning steps.

The challenge, however, lies in determining which concepts the model already understands well enough to be left unexpanded. Inspired by the Yoneda Lemma–based view of concepts [38], we propose a heuristic strategy: rather than fully expanding every concept, we selectively enrich them with auxiliary information that helps clarify their role within the task. This approach strikes a balance between completeness and conciseness, dramatically improving clarity score from 44.5% to 82.3%.

For the last question, we found the answer is yes. With more detailed and structured information provided for the task, the model is able to generate better reasoning performance. Different from many existing work [28, 29, 34, 35] that train a specialized model that were trained purely on math data, we use the general purposed model (deepseek-V3), which is not specifical trained on math. It turns out that when models are not specifically trained on formal mathematics, providing structured semantic information becomes even more critical for achieving strong performance. This suggests that our approach has broad applicability beyond specialized mathematical domains.

To fully leverage this structured data, we develop a Planner-Executor architecture that separates high-level strategic reasoning from low-level tactic generation. The Planner analyzes the structured proof state to identify relevant concepts, applicable theorems, and proof strategies, while the Executor translates these strategies into concrete Coq tactics. Our main approach uses general-purpose models (DeepSeek-V3) for both components, demonstrating that with proper task descriptions, specialized mathematical training is not necessary for strong theorem proving performance. Additionally, we explore fine-tuning smaller models on our structured data as an efficiency consideration, showing that a 32B model can achieve competitive results with significantly fewer parameters.

Our experimental results provide compelling evidence for this approach:

- **Concept Understanding**: Models achieve a 2.5× improvement on clarity score (from 44.5% to 82.3%) when provided with structured definitions.
- **Reasoning for Coq**: Using DeepSeek-V3 on 1,300 theorems from 15 standard Coq packages (same evaluation set as Graph2Tac [25]). Our system achieves 45.8% success rate, a 1.36× improvement over the previous state-of-the-art 33.2% (Graph2Tac).

- **Clarity improves reasoning**: We found positive correlation between clarity score and reasoning performance.

These results strongly validate our hypothesis: *improving task clarity through structured semantic information can substantially enhance model reasoning capabilities*.

2 Related Work

Learning-Based Theorem Proving in Proof Assistants. Classical automated theorem provers perform well in first-order logic, but struggle with complex formal verification. This limitation has driven research into learning-based methods aimed at automating interactions with proof assistants such as Coq [27], Isabelle [23] and Lean [6, 21]. Early work used simple machine learning algorithms to predict tactics from proof state features [4]. The field subsequently evolved through several architectural paradigms: from graph neural networks (GNNs) that explicitly encode the syntactic structure of formal expressions [25], to transformer-based large language models (LLMs) that process expressions as text [19, 13, 14, 26]. Subsequent work has explored various directions, such as developing novel proof search methods [15, 35], retrieving useful lemmas [11, 20, 37], applying further learning techniques including expert iteration and curriculum learning [1, 17, 24], enhancing performance through data augmentation [9], and designing practical tools [32, 14]. Although these learning-based approaches have shown promising results, they often treat the Coq (Lean, Isabelle) compiler as a black box without fully exploiting internal information, limiting their semantic understanding to a low level. Our work addresses this gap by providing mechanisms to access and utilize more comprehensive structured data directly from the Coq compiler's internal representation.

Strategies for Proof Automation. Most automated theorem provers adopt a step-by-step approach [3, 26, 34], generating individual tactics that are assembled into complete proofs, with compilers verifying steps and search algorithms guiding the proof. Recent work has demonstrated an alternative paradigm that generates complete formal proofs in a single cohesive process [7, 35, 31, 17], eliminating the need for explicit intermediate step generation and search. Autoformalization represents another promising direction, where LLMs convert informal mathematical statements into their formal equivalents [33, 35, 18, 17].

Leveraging Structured Information in Formal Verification. Recent approaches have leveraged structured information from proof assistants in various ways, including extracting internal data through interfaces like SerAPI [2], utilizing error messages via LSP in CoqPilot [14], analyzing structural information from sub-goals [13], and exploring tactic dependencies [36]. Researchers also incorporate rich metadata into datasets [8] to support more effective model training. Our work extends these approaches by implementing deep Coq compiler modifications to extract comprehensive structured data, significantly enhancing model understanding of the rich semantic information available in proof states and tactics.

Planner-Executor Architectures for Theorem Proving. The planner-executor paradigm in theorem proving has seen advancements such as layered systems separating strategy and tactics [12], multi-agent frameworks in Lean integrating natural language planning with proof assistant verification [30, 22], and approaches that interleave reasoning with formal proving steps [16, 37, 29]. Related explorations in the Coq ecosystem include generate-then-repair techniques [19], proof synthesis from partial attempts [13], and LLM-driven tactic selection within search processes [26]. Despite these efforts, establishing truly effective and seamless collaboration between high-level strategic planning and low-level tactic execution, particularly for complex reasoning chains within Coq, remains a significant challenge. Our work aims to develop a tightly integrated system to foster more effective collaboration between these components, thereby improving proof efficiency for complex theorems.

3 Preliminaries

3.1 The Calculus of Inductive Constructions

The Calculus of Inductive Constructions (CIC) is the type theory underlying Coq. To understand CIC, we must first clarify its fundamental concept: **terms**.

What are terms? In CIC, *term* is the universal syntactic category—everything in the language is a term:

- Values are terms: `42`, `true`, `[1,2,3]`
- Functions are terms: `fun x => x + 1`
- Types are terms: `nat`, `bool`, `list nat`
- Propositions are terms: `2 + 2 = 4`, `forall n, n < n + 1`
- Proofs are terms: any inhabitant of a proposition type

This differs from simply-typed lambda calculus, where "lambda terms" (like $\lambda x.x + 1$) are just the function expressions, separate from types. In CIC, the function $\lambda x : \mathbb{N}.x + 1$, its type $\mathbb{N} \to \mathbb{N}$, and even $\mathbb{N}$ itself are all terms.

CIC extends this unified framework with three key features:

1. **Dependent types**: Types can depend on values. While traditional types like `List Int` are fixed, dependent types like `Vec A n` vary—the type explicitly depends on the length n.
2. **Universe hierarchy**: An infinite hierarchy (Prop, Type_0, Type_1, ...) that organizes types to prevent paradoxes while maintaining logical consistency.
3. **Inductive definitions**: A way to define new types by declaring their constructors, like defining `nat` with constructors `O` and `S`.

Having established that everything in CIC is a term, we now examine the specific syntactic constructs used to build these terms. Understanding this syntax is crucial for grasping how CIC unifies computation and logic.

Term Syntax: Terms in CIC are built from five basic constructs:

- **Variables**: $x, y, z, \ldots$
- **Sorts**: Special constants that classify types—Prop (the type of propositions) and Type_i (a hierarchy of computational universes)
- **Abstractions** (lambda terms): $\lambda x : A.t$ represents a function taking input x of type A and returning t. For example, $\lambda n : \mathbb{N}.n + 1$ is a function that increments a natural number. Here, A is a type (like $\mathbb{N}$ or Prop), while t is a term.
- **Applications**: $f\ a$ applies function f to argument a
- **Products** (dependent function types): $\Pi x : A.B$ represents the type of functions from A to B, where crucially, the return type B may depend on the input value x. This dependency is what makes CIC's type system so expressive.

Among these constructs, the product type deserves special attention as it is the key to understanding both dependent types and the logical power of CIC.

Understanding Products: The product type $\Pi x : A.B$ generalizes ordinary function types. When B does not depend on x, we write it as $A \to B$ (simple function type). When B does depend on x, we get dependent types. For example:

- Simple function: $\mathbb{N} \to \mathbb{N}$ is the type of functions from naturals to naturals
- Dependent function: $\Pi n : \mathbb{N}.\text{Vec}(A, n)$ is the type of functions that take a natural number n and return a vector of length n
- Logical quantification: When the codomain is Prop, we often write $\forall$ instead of Π. For instance, $\forall n : \mathbb{N}.\text{even}(n) \to \text{even}(n + 2)$ expresses that for any natural number n, if n is even, then $n + 2$ is even.

Products thus unify function types (computation) and universal quantification (logic) in a single construct. This unification is not coincidental—it reflects a deeper principle that connects programming and theorem proving at their foundations.

The Curry-Howard Correspondence [5, 10]: This fundamental principle establishes a deep connection between logic and computation by identifying:

- **Propositions as Types**: Each logical proposition corresponds to a type. For example, the proposition "A implies B" corresponds to the function type $A \to B$.
- **Proofs as Programs**: A proof of a proposition corresponds to a term (program) of the corresponding type. For example, a proof of "A implies B" is a function that transforms evidence of A into evidence of B.
- **Proof Checking as Type Checking**: Verifying a proof's correctness reduces to checking that a term has the claimed type.

For example, proving the theorem "for all natural numbers n, if n is even, then $n + 2$ is even" means constructing a function of type $\forall n : \mathbb{N}.\text{even}(n) \to \text{even}(n + 2)$. This function takes two inputs: a natural number n and a proof that n is even, and produces a proof that $n + 2$ is even.

With this theoretical foundation in place, we now turn to how these principles are implemented in practice through the Coq proof assistant.

3.2 Coq and Interactive Theorem Proving

Coq is an interactive theorem prover that implements CIC. Users develop formal proofs in `.v` files containing definitions, theorem statements, and proof scripts. In Coq, definitions are terms that compute values, theorem statements are types (propositions), and proof scripts guide the construction of terms that inhabit these types (i.e., proofs). During interactive proof development, Coq maintains a *proof state* showing the current hypotheses and goals. Consider the following example demonstrating the double_even theorem:

```
Variable A : Type.

Definition double (n : nat) := n + n.

Inductive even : nat -> Prop :=
  | even_0 : even 0
  | even_SS : forall n, even n ->
              even (S (S n)).

Theorem double_even :
  forall n, even (double n).
Proof.
  intros n.
  unfold double.
  induction n.
  - simpl. apply even_0.
  - simpl. apply even_SS.
    assumption.
Qed.
```

```
(* Initial state after intros n *)
n : nat                (* hypotheses *)
============================
even (double n)        (* goal *)

(* After unfold double *)
n : nat                (* hypotheses *)
============================
even (n + n)           (* goal *)

(* After induction n - Subgoal 1 *)
============================
even (0 + 0)

(* After induction n - Subgoal 2 *)
n : nat
IHn : even (n + n)
============================
even (S n + S n)
```

Figure 1: Coq proof script (left) and corresponding proof state for the double-even theorem (right)

3.3 Fundamental Distinction: Entities and Proofs

Coq program can be naturally divided into two distinct aspects: **what we define** and **how we prove**. Through the lens of the Curry-Howard correspondence:

- **Entities** establish our logical universe: they introduce new types (which may represent propositions), constructors (which build terms of these types), and computational functions. These form the static foundation—the "vocabulary" of types and terms available for reasoning. This includes variables, parameters, functions, inductive type definitions, theorem statements, axioms, notations, etc.
- **Proofs** are the dynamic process of constructing terms that inhabit proposition types. When we prove a theorem like $\forall n : \mathbb{N}.\text{even}(n) \to \text{even}(n + 2)$, we're constructing a term of that type—specifically, a function that transforms evidence of even(n) into evidence of even($n + 2$). The proof process unfolds through:
  - **Initial proof state**: the theorem statement as a type to be inhabited
  - **Intermediate states**: partial proof terms with holes (goals) to be filled

- **Tactic sequences**: commands that refine partial terms toward completion
- **Context evolution**: how the typing context grows with new hypotheses

This fundamental distinction between establishing the logical framework (entities) and constructing proofs within that framework guides our approach to structuring Coq's information for language models.

To formalize these concepts, we establish the following definitions:

Definition 1 (Coq Entity) *A Coq Entity is a binding that associates a name with both surface-level source code and internal elaborated representations. Formally, an entity d is a tuple (n, c, o, i) where:*

- *n is the fully-qualified name (e.g., `Coq.Init.Nat.add`)*
- *c is the entity type, corresponding to the entity categories listed above (Variable, Parameter, Definition, Theorem, Inductive, Fixpoint, etc.)*
- *o is the source text as it appears in the `.v` file*
- *i is the internal type representation after Coq's type inference*

For example, the entity `Coq.Init.Logic.True` is represented as:

- n = `Coq.Init.Logic.True`
- c = `Inductive`
- o = `Inductive True := I : True`
- i = `True:  Prop | Coq.Init.Logic.True.I : Coq.Init.Logic.True`

This entity defines the trivial proposition `True` with its single constructor `I`.

Definition 2 (Proof State) *A proof state σ represents the current status of an interactive proof. Formally, $\sigma = \{s_1, ..., s_m\}$ where each state $s_k = (H_k^o, H_k^i, G_k^o, G_k^i)$ consists of:*

- *H_k^o is the surface hypothesis context for state k: hypotheses as shown in the proof interface*
- *$H_k^i = \{(h_{k,1} : t_{k,1}), ..., (h_{k,n_k} : t_{k,n_k})\}$ is the internal hypothesis context, where each $h_{k,j}$ is a hypothesis name and $t_{k,j}$ is its elaborated type*
- *G_k^o is the surface goal representation: the goal as shown in the proof interface*
- *G_k^i is the internal goal representation: the elaborated proposition to be proven*

For example, in a proof about natural numbers where the goal is to prove `a + b = b + a`, the proof state might contain:

- H_1^o = {(`a:nat`), (`b:nat`), (`IHa:a + b = b + a`)}
- H_1^i = {(`a: nat`), (`b: nat`), (`IHa: Coq.Init.Logic.eq nat (Coq.Init.Nat.add a b) (Coq.Init.Nat.add b a)`)}
- G_1^o = `S a + b = b + S a` (the surface goal)
- G_1^i = `eq nat (Coq.Init.Nat.add (S a) b) (Coq.Init.Nat.add b (S a))`

Definition 3 (Interactive Proof) *An interactive proof is a sequence of tactic applications that transforms an initial proof state to a completed state. Formally, a proof π is a sequence:*

$$\pi = [(T_1, \sigma_0, \sigma_1), (T_2, \sigma_1, \sigma_2), ..., (T_k, \sigma_{k-1}, \sigma_k)]$$

where:

- *T_i is a tactic: a command that transforms proof states*
- *σ_i is the proof state after applying tactic T_i*

- *σ_0 is the initial proof state with the theorem statement as the single goal*
- *σ_k is the final proof state where $G_\tau = \emptyset$ (no remaining goals)*
- *Each transition $\sigma_{i-1} \xrightarrow{T_i} \sigma_i$ represents a valid tactic application*

For example, proving `forall n:nat, 0 + n = n` demonstrates how structural data reveals semantic information:

- T_1 = `intros n`
- σ_0 = $\{s_1\}$ where:
  - H_1^o = { }
  - H_1^i = { }
  - G_1^o = `forall n:nat, 0 + n = n`
  - G_1^i = `forall (n:nat), eq nat (Coq.Init.Nat.add 0 n) n`
- σ_1 = $\{s_1\}$ where:
  - H_1^o = {`n:nat`}
  - H_1^i = {`(n : nat)`}
  - G_1^o = `0 + n = n`
  - G_1^i = `eq nat (Coq.Init.Nat.add 0 n) n`
- T_2 = `simpl`
- σ_2 = $\{s_2\}$ where goal becomes `n = n` (surface) / `Coq.Init.Logic.eq nat n n` (internal)
- T_3 = `reflexivity`
- σ_3 = { } (proof completed)

In theorem proving, accurately tracking entity dependencies is crucial. When a proof uses a theorem or definition, we need to identify exactly which entity is being referenced—enabling dependency analysis, concept unfolding, and proper handling of missing imports. However, Coq's flexible naming system creates significant challenges:

- **Section scoping**: Definitions inside and outside sections can share the same name but refer to different entities
- **Module aliasing**: The same entity can be accessed through multiple paths (e.g., `Z.add` vs `BinInt.Z.add`)
- **Implicit arguments**: Surface syntax may omit type parameters that are crucial for identification
- **Notations**: Operators like + can refer to different functions depending on type context

To address these challenges, we introduce a domain-specific tokenization system that assigns unique identifiers to Coq entities, ensuring semantic consistency across all naming variations:

Definition 4 (Domain-specific Token) *A token is a unique identifier id $\in \mathbb{N}$ assigned to each Coq entity. The tokenization function $\mathcal{T}$: Names $\times$ Context $\rightarrow \mathbb{N}$ maps an entity reference in a given context to its unique identifier, where:*

- *Names is the set of all possible names used to reference entities in Coq code*
- *Context represents the resolution context (current module, section parameters, type environment)*
- *Token satisfies semantic consistency:*
  - *If two names refer to the same Coq entity (even with different surface representations), they map to the same token*
  - *If two names refer to different Coq entities, they map to different tokens*

This tokenization enables reliable entity tracking across the entire Coq ecosystem. Whether analyzing dependencies in existing proofs, suggesting relevant lemmas during proof search, or ensuring correct imports, the system provides a consistent way to identify and reference Coq entities regardless of their surface representation.

These formal definitions provide the mathematical foundation for our data extraction and processing pipeline, ensuring precise representation of Coq's semantics throughout our framework.

4 Evaluating Clarity

To address our first question—*Can we assess how well a model understands a given task?*—we develop a methodology for directly measuring how well models comprehend Coq concepts in theorem proving contexts. When a model fails to prove a theorem, it may be due to insufficient reasoning ability or inadequate clarity of the mathematical concepts involved. Our approach provides a systematic way to distinguish between these two possibilities.

4.1 Experimental Design

Given a proof scenario with various concepts, we:

1. Extract all concepts appearing in the proof state
2. Randomly select one concept
3. Ask the model to provide its strict Coq definition
4. Evaluate the semantic correctness of the generated definition

For example, when evaluating clarity in the context of structured prompts(which include concepts like `nat`, `plus`, and `eq`), we might ask:

> *"Given the following structural proof context: [full structural prompt], please provide the strict Coq definition of the concept `plus`."*

4.2 Clarity Score

We introduce the **Clarity Score**, a metric that quantifies how accurately a model can define concepts. Using a language model as an evaluator [39], we assess:

> *"Is the following definition semantically correct for [concept]? [model's generated definition]"*
> *Answer: YES or NO*

The Clarity Score is computed using log probabilities:

$$\text{Clarity Score} = \frac{\exp(\log P(\text{YES}))}{\exp(\log P(\text{YES})) + \exp(\log P(\text{NO}))} \tag{1}$$

This produces a score between 0 and 1, where higher scores indicate better conceptual clarity. We implement this evaluation using the `DeepSeek-V3` model as the judge.

Our experiments (Section 7) demonstrate that current approaches leave models with limited comprehension of the mathematical concepts they encounter. This suggests that many reasoning failures may stem from inadequate task understanding rather than flawed logical reasoning.

This observation motivates our approach: by systematically enhancing how we present tasks to models, we may achieve better reasoning performance. The following section explores how to bridge this understanding gap through enriched task descriptions.

5 Enhancing Task Description

We now address the second question: *Can we improve the model's understanding by enriching the task description?* Our key insight is that while human readers leverage implicit mathematical knowledge to interpret Coq code, language models require explicit access to the same semantic information that Coq's type checker uses internally.

5.1 The Semantic Gap Problem

Current approaches to LLM-based theorem proving face a fundamental limitation: they train models on surface-level `.v` files, missing the rich type-theoretic information that gives Coq code its precise meaning. Consider a simple expression like `a + b`:

- **Surface syntax hides crucial information**: A term like `a + b` could represent natural number addition, list concatenation, or boolean operations depending on context
- **Implicit arguments remain hidden**: Coq infers many type parameters that are essential for clarity but invisible in source code
- **Module aliasing creates ambiguity**: The same entity can be referenced through multiple names (e.g., `Z.add` vs `BinInt.Z.add`)
- **Notations obscure underlying structure**: Operators like + are syntactic sugar that map to different functions based on type context

These issues compound in real proofs, where clarity requires resolving types, tracking implicit arguments, and disambiguating overloaded notations—precisely the computations Coq performs internally but never exposes in source files.

5.2 Coq Data Processing Pipeline

To bridge this semantic gap, we develop a pipeline that intercepts Coq's compilation process to extract the type-theoretic information computed internally. Our approach transforms raw Coq source into structured representations that make implicit knowledge explicit, providing language models with the same semantic precision available to Coq's kernel.

The pipeline addresses three complementary aspects of semantic clarity:

1. **Entity Extraction**: Extracts Coq entities as defined in Definition 1, capturing the complete tuple (n, c, o, i) including both surface representations and kernel-elaborated types.
2. **Proof State Extraction**: Extracts Proofs as defined in Definition 3, recording complete proof state transitions with both surface and internal representations of goals and hypotheses.
3. **Domain-specific Tokenizer**: Realizes the tokenization function $\mathcal{T}$ from Definition 4, ensuring semantic consistency across naming variations and context-dependent references.

How Does This Improve Clarity? This structured context directly addresses each component of the semantic gap:

- **Notation Clarity**: Models receive explicit mappings from surface notation to underlying functions, eliminating guesswork about operator meanings
- **Type Precision**: Complete type information enables models to reason about type compatibility and implicit argument instantiation
- **Reference Resolution**: Unique identifiers for entities prevent confusion from module aliasing and namespace conflicts
- **Proof Transparency**: Access to internal proof state representations reveals the complete logical context for each tactic decision

By providing this structured semantic context, we transform the theorem proving task from parsing ambiguous surface syntax to reasoning with precise mathematical semantics. The complete technical implementation details are provided in Appendix A.

5.3 From Extracted Data to Structured Reasoning

The semantic context extracted by our pipeline provides the foundation for enhanced task descriptions, but raw semantic information alone is insufficient. We must transform this data into structured prompts that guide models toward systematic mathematical reasoning. Our approach mirrors how expert Coq users mentally process proofs: first clarify the types and definitions involved, then track how tactics transform the proof state, and finally select appropriate next steps based on this understanding.

Our enhanced task descriptions follow a systematic format:

1. **Proof State**: Current goals and hypotheses with dual representations
2. **Entity Definitions**: For each referenced concept(examples in Appendix B):
   - Origin: Source code definition
   - Internal: Kernel representation
   - Intuition: Natural language explanation
3. **Context Information**: Module imports, notation definitions, and available theorems
4. **Additional Enhancements**: Other task-specific enhancements detailed in subsequent section 6

These components are integrated into structured prompts that provide comprehensive semantic context for theorem proving tasks. The detailed format and examples of our structured prompts are provided in Appendix C.

With both the evaluation methodology for understanding (Section 4) and the enhancement pipeline now in place, we are ready to develop proof search algorithms that leverage these structured representations. The next section presents our Planner-Executor architecture that transforms this enhanced understanding into effective theorem proving capability. The complete experimental validation of our approach—demonstrating that enhanced task descriptions indeed improve both understanding scores and theorem proving performance—is presented in Section 7.

6 Searching Algorithm

With enhanced clarity through structured semantic information, we now address how to leverage this clarity for effective theorem proving. We introduce a specialized Planner-Executor architecture that transforms our structured semantic data into systematic proof construction, explicitly modeling the hierarchical nature of mathematical reasoning.

6.1 Planner-Executor Proof Architecture

The architecture operates on two fundamental principles:

1. **Structured Data Foundation**: Both Planner and Executor receive the rich structured representation:
   - Current proof state σ with dual surface/internal representations (Definition 2)
   - All entities referenced in the proof state and current context, with complete definitions (n, c, o, i) retrieved via our semantic tokenizer (Definition 1)
2. **Hierarchical Decomposition**: The Planner analyzes this structured data to generate strategic guidance, which the Executor then uses—alongside the same structured data—to produce concrete tactics.
   - **Planner Output**: Structured analysis including:
     - Core mathematical concepts and structures involved
     - Relevant theorems and properties applicable to the current state
     - Proof techniques suitable for the goal (induction, contradiction, case analysis)
     - Key relationships between hypotheses and goals
     - Strategic summary synthesizing the analysis
   - **Executor Output**: Multiple concrete Coq tactics generated via beam search

This hierarchical decomposition transforms our enhanced clarity into actionable proof strategies. While the semantic extraction provides the "what" (precise mathematical meanings), the Planner-Executor architecture provides the "how" (systematic proof construction).

6.2 Enhanced Capabilities

Beyond the core Planner-Executor decomposition, additional techniques further amplify the architecture's effectiveness (detailed algorithms provided in Algorithm 1):

Search and Exploration:

- **Beam Search**: Maintains the k most promising proof states (width=3), efficiently exploring multiple proof paths while pruning less convincing branches
- **Error Reflection**: Compiler error messages from failed tactics guide regeneration, enabling iterative refinement based on concrete feedback
- **Precise External Referencing**: Our tokenizer enables us to trace any theorem back to its source file and automatically generate the correct `Require` statements when referencing external premises

Context Augmentation:

- **Retrieval Augmentation**: Encodes the Planner's strategic analysis to retrieve semantically similar premises and tactics from the corpus
- **Precise External Referencing**: Our semantic tokenizer ensures accurate resolution of all entity references, preventing naming ambiguities across different namespaces
- **Intuitive Annotations**: Each entity is augmented with single-sentence natural language descriptions for enhanced comprehension

Proof Intelligence:

- **Dynamic Summarization**: Generates proof progress summaries including expected completion steps and state quality scores
- **Tactic Explanation**: Produces bilingual (formal/natural) explanations for each successful tactic application, capturing transformations and rationale
- **Tactic Trace**: Maintains complete proof paths with step-by-step explanations of how the current state was reached
- **Public Notebook**: Evaluates and maintains a fixed-size cache (15 items) of key insights from successful proof steps, accessible throughout the proof

These capabilities work together to create a comprehensive reasoning framework. The semantic foundation enables precise clarity, the Planner provides strategic direction, and the enhanced capabilities ensure robust execution even in the face of failures.

6.3 Proof Search Algorithm

Having established the Planner-Executor architecture and its enhanced capabilities, we now present the complete proof search algorithm that operationalizes these components into a systematic theorem proving process.

Core Algorithm Our proof search integrates all previous components into a coherent search strategy. The algorithm maintains a beam of $B = 3$ candidate proof states, using the structured semantic context to guide exploration at each step.

Each search iteration performs three coordinated phases:

1. **State Expansion**: For each candidate state in the beam:
   - Extract structured representation: proof state σ, expanded entities via semantic tokens
   - Generate strategic analysis using the Planner module
   - Retrieve relevant premises and successful tactics based on the strategy
   - Produce multiple tactic candidates using the Executor with beam search
   - Validate tactics through Coq's compiler, applying error reflection on failures

2. **State Selection**: Score and rank all generated states, retaining top-B based on:
    - Progress toward goal completion
    - Strategy coherence from Planner analysis
    - Historical success patterns
3. **Context Maintenance**: Update shared resources:
    - Public notebook with proven insights (max 15 items)
    - Tactic trace for successful paths
    - Error history for failed attempts

The algorithm terminates when any branch achieves σ_k where all goals are discharged, or when reaching the depth limit. Detailed pseudocode is provided in Algorithm 1.

Algorithm 1 Neural Theorem Proving with Structured Coq Representations (Part 1)

```
Require: Theorem $T$, Max depth $D$, Beam width $B$, Max retries $R$
Ensure: Proof trace or FAILURE
1: $S_0 \leftarrow$ COMPILETHEOREM($T$)
2: $layer \leftarrow \{(S_0, [], ``", ``")\}$                ▷ (state, trace, summary, notes)
3: for $d = 1$ to $D$ do
4:     $next \leftarrow \emptyset$
5:     $insights \leftarrow []$
6:     for each $(S, trace, summary, notes) \in layer$ do
7:         /* Step 1: Extract concepts */
8:         $C \leftarrow$ EXTRACTCONCEPTS($S, depth = 1$)
9:         /* Step 2: Generate strategy */
10:        $prompt\_method \leftarrow (S, C, trace, summary, notes)$
11:        $(resp, strategy) \leftarrow$ GENERATESTRATEGY($prompt\_method$)
12:        /* Step 3: Retrieve relevant information */
13:        $(premises, tactics) \leftarrow$ RETRIEVE($resp, k$)
14:        /* Step 4: Generate candidate tactics */
15:        $prompt\_tactic \leftarrow (S, C, premises, tactics, strategy, trace, summary, notes)$
16:        $candidates \leftarrow$ BEAMSEARCHTACTICS($prompt\_tactic$, 10)
17:        /* Step 5: Validate and retry on failure */
18:        $valid \leftarrow []$
19:        $failed \leftarrow []$
20:        for each $t \in candidates$ do
21:            $r \leftarrow$ COMPILETACTIC($t, S$)
22:            if $r.success$ then
23:                $valid$.append($t$)
24:            else
25:                $failed$.append($(t, r.error)$)
26:            end if
27:        end for
28:        /* Retry failed tactics */
29:        $retry \leftarrow 0$
30:        while $retry < R$ and $failed \neq []$ and $|valid| \leq 10$ do
31:            $errors \leftarrow$ [errors from $failed$]
32:            $prompt\_error \leftarrow (S, C, errors, trace, prompt\_method)$
33:            $(resp', strategy') \leftarrow$ GENERATESTRATEGY($prompt\_error$)
34:            $(premises', tactics') \leftarrow$ RETRIEVE($resp', k$)
35:            $prompt\_tactic' \leftarrow (S, C, premises', tactics', strategy', trace, summary, notes)$
36:            $retry\_candidates \leftarrow$ BEAMSEARCHTACTICS($prompt\_tactic'$, 10)
37:            $failed \leftarrow []$
```

Algorithm 1 Neural Theorem Proving with Structured Coq Representations (Part 2)

```
38:                 for each $t \in retry\_candidates$ do
39:                     $r \leftarrow$ CompileTactic($t, S$)
40:                     if $r.success$ then
41:                         $valid$.append($t$)
42:                     else
43:                         $failed$.append($(t, r.error)$)
44:                     end if
45:                 end for
46:                 $retry \leftarrow retry + 1$
47:             end while
48:             /* Step 6: Apply valid tactics */
49:             for each $t \in valid$ do
50:                 $S' \leftarrow$ ApplyTactic($t, S$)
51:                 if IsGoalComplete($S'$) then
52:                     return $trace + [t]$
53:                 end if
54:                 if IsSubgoalComplete($S'$) then
55:                     $S' \leftarrow$ RefreshWithIdtac($S'$)
56:                 end if
57:                 $expl \leftarrow$ GenerateExplanation($S, t, S'$)
58:                 $trace' \leftarrow trace + [(t, expl)]$
59:                 $summary' \leftarrow$ SummarizeProof($trace'$)
60:                 $next$.append($(S', trace', summary', notes)$)
61:                 $insights$.append($expl$)
62:             end for
63:         end for
64:         if $next = \emptyset$ then
65:             return FAILURE
66:         end if
67:         /* Step 7: Update public notes */
68:         if $insights \neq []$ then
69:             $notes \leftarrow$ UpdatePublicNotes($S_0, insights, notes$)
70:         end if
71:         /* Step 8: Beam search selection */
72:         if $|next| > B$ then
73:             $layer \leftarrow$ SelectBest($S_0, next, B$)
74:         else
75:             $layer \leftarrow next$
76:         end if
77:     end for
78:     return FAILURE
```

The algorithm operationalizes the insights from our enhanced clarity: it uses the dual representations to guide search, leverages the Planner's analysis for strategic coherence, and employs beam search to balance exploration with computational efficiency.

6.4 Integrating Components into a Complete Proof System

This section completes our methodological framework by integrating all previously developed components into a unified theorem proving system. We have now established:

Semantic Foundation: Our data processing pipeline (Section 5.2) extracts the precise type-theoretic information that Coq computes internally, providing models with disambiguated entities, complete type information, and resolved proof states.

Strategic Architecture: Our Planner-Executor framework (Section 6.1) separates high-level proof strategy from low-level tactic generation, enabling systematic reasoning that mirrors expert mathematical practice.

Coordinated Search: Our proof search algorithm orchestrates these components into a coherent search process, where semantic clarity guides strategic planning, which in turn directs tactical execution.

The resulting system represents a complete methodology for neural theorem proving that leverages structured semantic information at every level—from individual symbol disambiguation to high-level proof strategy. With our approach fully specified, we are now ready to evaluate its effectiveness. The next section presents comprehensive experiments that measure both clarity improvement and theorem proving performance, providing empirical answers to our three central research questions.

7 Experiments

Having established our complete methodological framework, we now conduct comprehensive experiments to validate our three research questions:

1. **Can we assess how well a model understands a given task?** We evaluate our clarity measurement methodology (Section 4) across different information configurations.
2. **Can we improve the model's understanding by enriching the task description?** We test whether our structured semantic information (Section 5) enhances both clarity and reasoning performance.
3. **Does deeper clarity lead to better reasoning performance?** We analyze the relationship between clarity scores and theorem proving success rates, validating our central hypothesis.

Our experimental design systematically addresses each question in sequence, building evidence for our approach's effectiveness.

7.1 Experiment 1: Assessing Model Clarity and Task Enhancement

Our first experiment simultaneously addresses Questions 1 and 2 by measuring model clarity across different information configurations. This experiment validates our measurement methodology while demonstrating the effectiveness of task enrichment.

7.1.1 Experimental Setup

For evaluation, we construct a dataset from 1,000 structured prompts randomly sampled from our Coq proof generation process. Each structured prompt contains the components described in Section 5, including a list of global definitions referenced in the proof context. We use this reference list to identify relevant semantic content, randomly selecting up to 3 definitions per prompt (or all if fewer than 3) to construct the corresponding information configurations. Consequently, each information condition consists of slightly fewer than 3,000 evaluated instances. The detailed format of our structured prompts is provided in Appendix C.

7.1.2 Information Configurations

We test three types of semantic information extracted by our pipeline:

- **Origin**: Raw Coq source code definitions
- **Internal**: Compiler internal representations (kernel-level)
- **Intuition**: Natural language explanations generated by large language models

This yields 8 different information combinations, plus additional control conditions:

1. **No Context**(current standard approach): Only proof state with simple names (e.g., `plus`)
2. **Qualified Name**: Only proof state with full names (e.g., Coq.Arith.Plus.plus)
3. **Empty Reference**: Proof state + additional techniques
4. **Origin Only**: Proof state + raw Coq definitions + additional techniques
5. **Internal Only**: Proof state + kernel representations + additional techniques

6. **Intuition Only**: Proof state + natural language explanations + additional techniques
7. **Origin + Internal**: Combination of raw code and kernel info + additional techniques
8. **Origin + Intuition**: Combination of raw code and explanations + additional techniques
9. **Internal + Intuition**: Combination of kernel info and explanations + additional techniques
10. **Complete Information**: All three types of information + additional techniques
11. **Chinese Translation**: Complete information translated to Chinese + additional techniques

Before presenting our enhanced configurations, it is crucial to establish the baseline. Current state-of-the-art theorem proving systems, including DeepSeek Prover [35], Kimina-Prover [28], and other advanced neural theorem provers [29, 34], employ a standard approach where models are provided with **No Context**.

Our evaluation reveals that under these standard conditions—the same setup used by current theorem proving approaches—models achieve only 44.5% understanding when asked to define basic concepts. This means that nearly 60% of the time, models cannot accurately comprehend the mathematical concepts they are working with, which may explain why reasoning performance often falls short of expectations.

7.1.3 Clarity Measurement Results

Using our concept definition task methodology, we measure clarity scores across all configurations. The results demonstrate that our measurement approach effectively captures differences in model comprehension:

Information Configuration	**Clarity Score**
Control Conditions	
No Context(current standard approach)	0.445
Qualified names	0.581
Empty Information	
Empty Reference	0.615
Single Information Type	
Origin Only	0.798
Internal Only	0.712
Intuition Only	0.667
Combined Information	
Origin + Internal	0.815
Origin + Intuition	0.803
Internal + Intuition	0.714
Complete Information	**0.823**
Language Variation	
Chinese Translation	0.732

Table 1: Model clarity scores under different information configurations

The results validate our clarity measurement methodology and reveal several important insights:

1. **Measurement sensitivity**: Our methodology successfully discriminates between different levels of clarity, with scores ranging from 44.5% to 82.3%.
2. **Qualified names matter**: Using fully qualified names instead of simple names improves clarity by 13.6% absolute, demonstrating that disambiguation is crucial.
3. **Structured information effectiveness**: Providing origin code definitions improves clarity from 61.5% to 79.8%—an 18.8% absolute gain, showing our semantic extraction provides valuable context.
4. **Cross-language validity**: Chinese translations maintain substantial clarity capability (73.2%), indicating that structural information rather than language-specific patterns drives improvements.

5. **Information complementarity**: While origin code provides the strongest single signal, internal representations and natural language intuitions offer complementary benefits, with complete information achieving 82.3% clarity.

These findings address Questions 1 and 2 directly: our methodology can effectively assess model understanding across different configurations, and enriching task descriptions with structured semantic information dramatically improves comprehension from 44.5% to 82.3%.

7.2 Experiment 2: Clarity-Performance Relationship

7.2.1 Experimental Setup

Dataset Following Graph2Tac's evaluation protocol, we randomly sample 10% from their test set. Due to differences in data processing pipelines, the final dataset contains 1,300 theorems from 15 standard Coq packages, which we believe is sufficient for meaningful comparison. The dataset covers diverse mathematical domains including:

- Pure mathematics: real analysis, number theory, topology, group theory
- Computer science: separation logic, type theory, formal verification
- Computational mathematics: constructive algebra, homotopy type theory

Search Parameters Based on preliminary experiments, we set:

- Maximum proof depth $D = 15$ (sufficient for most theorems)
- Beam width $B = 3$ (balancing exploration and efficiency)
- Maximum retries $R = 3$ (one retry with error feedback)

Computational Budget To ensure fair comparison across different model configurations, we establish a consistent computational budget for proof search. Each theorem is allocated a maximum of 860 tactic evaluations, calculated as follows:

$$\begin{aligned}
\text{Budget} &= \text{Initial layer} \times \text{Tactics per state} \times \text{Reconsider factor} && (2)\\
&\quad + \text{Subsequent layers} \times \text{Beam width} \times \text{Tactics per state} \times \text{Reconsider factor} && (3)\\
&= 1 \times 10 \times 2 + 14 \times 3 \times 10 \times 2 = 860 && (4)
\end{aligned}$$

7.2.2 Theorem Proving Performance

We evaluate our complete system against the Graph2Tac baseline to measure the practical impact of enhanced clarity:

Method	Success Rate	Avg Depth	Avg Tactics
Graph2Tac (baseline)	33.2%	-	-
DeepSeek-V3 (our baseline)	21.8%	3.2	485
DeepSeek-V3 + DeepSeek-V3 (our method)	**45.8%**	2.7	357

Table 2: Main results comparing our method to Graph2Tac baseline

- **Significant improvement over state-of-the-art**: Our Planner-Executor system achieves 45.8% success rate, outperforming Graph2Tac by 12.6 percentage points (38% relative improvement)
- **Efficiency gains**: Our method requires fewer tactics on average (357 vs 485 for baseline) and achieves proofs in fewer steps (2.7 vs 3.2 depth)
- **Bridging the gap**: While our DeepSeek-V3 baseline (21.8%) initially underperforms Graph2Tac, our structured semantic approach not only closes this gap but achieves superior performance

7.2.3 Clarity-Performance Correlation

To directly test our central hypothesis, we analyze the relationship between understanding scores and theorem proving success rates across different information configurations. For this analysis, we use a randomly sampled subset of 100 theorems from our full dataset of 1,300 theorems to ensure computational feasibility while maintaining statistical validity:

Configuration	Clarity Score	Success Rate
No Context(current standard approach)	0.445	21%
Qualified Name	0.581	25%
Internal Only	0.712	38%
Origin Only	0.798	42%
Complete Information	0.823	45%

Table 3: Strong correlation between clarity and proving performance

The correlation analysis reveals a remarkably strong linear relationship (correlation coefficient $r = 0.98$) between understanding and reasoning performance. As understanding scores increase from 44.5% to 82.3%, theorem proving success rates improve proportionally from 21% to 45%.

Combined with our performance results—where enhanced understanding achieves 45.8% success rate versus the 33.2% Graph2Tac baseline—this evidence addresses Question 3 conclusively. The near-perfect correlation demonstrates that understanding improvements directly drive reasoning performance, supporting our central hypothesis that AI reasoning failures in formal mathematics often stem from inadequate task understanding rather than insufficient reasoning capability.

7.3 Additional Results and Analysis

Having answered our three central research questions, we present additional experiments that demonstrate the broader applicability and practical value of our approach.

7.3.1 Fine-Tuned Model Performance

We investigate whether fine-tuning smaller models on our structured data can achieve competitive performance, offering efficiency benefits for practical deployment.

7.3.2 Training Data

Our fine-tuning dataset consists of 400,000 (*structured_context*, *tactic*) pairs extracted from our data processing pipeline, where:

- *structured_context* includes all components: current proof state with dual representations, referenced entities with complete definitions, strategic analysis, retrieved premises/tactics, proof trace, and error history
- *tactic* is the successful tactic applied at that state

7.3.3 Results

The fine-tuning experiment yields promising results:

Method	Success Rate	vs Graph2Tac	Model Size	Efficiency
Graph2Tac (baseline)	33.2%	1.00×	-	-
DeepSeek-V3 + DeepSeek-V3	45.8%	1.38×	671B	1×
Qwen-2.5-7B (fine-tuned)	45.2%	1.36×	7B	96×
Qwen-2.5-32B (fine-tuned)	**48.6%**	**1.46×**	32B	21×

Table 4: Fine-tuned models achieve competitive performance with significant parameter efficiency

Despite using only 400,000 training pairs—a relatively small dataset—the fine-tuning results are promising. The Qwen-2.5-32B model achieves 48.6% success rate, our best overall result, while the

7B model nearly matches the 671B DeepSeek-V3 performance with 96× fewer parameters. This demonstrates the efficiency potential of our structured data approach.

Given that our training data represents only a fraction of available proof data, there is substantial room for improvement. With more comprehensive training data, we anticipate even stronger performance from fine-tuned models, making this an attractive direction for efficient deployment.

7.3.4 Detailed Analysis by Library Type

Our results show consistent improvements across different mathematical domains:

Library	DS Baseline	DS+DS	Qwen-32B-FT
Logic & Foundations			
TLC	0.40	0.65	0.68
SMTCoq	0.31	0.55	0.65
HoTT	0.19	0.39	0.40
Pure Mathematics			
CoRN	0.35	0.52	0.63
Topology	0.10	0.27	0.26
Ceres	0.17	0.29	0.57
Computer Science			
PolTac	0.64	0.81	0.93
Qcert	0.23	0.57	0.65
iris	0.10	0.09	0.20

Table 5: Success rates by library demonstrating broad applicability

Key observations:

- **Consistent improvements**: Both DS+DS and fine-tuned approaches show gains across all library types
- **Domain specialization**: Fine-tuned models excel particularly in specialized domains like PolTac (tactics) and Ceres (geometry)
- **Challenging domains**: Even in difficult areas like topology and HoTT, our approach provides meaningful improvements

7.4 Ablation Studies

We conducted comprehensive ablation studies to understand the contribution of each component. The information component analysis is performed on 1386 theorems, which are randomly sampled 10% from the Coq Standard Library. Others are performed on the Ceres library containing 78 theorems.

Information Components	Success Rate
Proof State + Entities + Structured Internal Representations	37.2%
+ Retrieved Premises and Tactics	42.6% (+5.4%)
+ Proof Trace	46.8% (+4.2%)
+ Public Notebook	**48.6%** (+1.8%)

Table 6: Incremental benefits of information components

Information Component Analysis Each information component provides meaningful improvements:

- **Retrieved premises/tactics** (+5.4%): Largest single improvement from relevant context
- **Proof trace** (+4.2%): Historical context aids strategic planning
- **Public notebook** (+1.8%): Shared insights across proof steps

Architecture	Success Rate
Single DeepSeek-V3	21.8%
DeepSeek-V3 + DeepSeek-V3 (Planner-Executor)	45.8%
DeepSeek-V3 (Planner) + Qwen-32B-FT (Executor)	**48.6%**

Table 7: Architectural design impact on performance(78 theorems from Ceres library)

Architecture Comparison The Planner-Executor decomposition provides substantial benefits (+24%), with fine-tuned executors offering additional gains (+2.8%).

Retry Count	State Selection	Success Rate
0	Shortest proof states	35.1%
3	Shortest proof states	40.3%
1	Model-based selection	44.2%

Table 8: Search strategy optimization results(78 theorems from Ceres library)

Search Strategy Optimization A single retry with error feedback provides optimal performance, while excessive retries hurt due to search space explosion.

8 Conclusion

This work investigates how well models understand mathematical tasks by introducing clarity score. Using this metric, we show that adding structured semantic context significantly improves clarity score. In addition, our experiments reveal a strong positive correlation between clarity score and reasoning ability. Notably, our method yields substantial gains for general-purpose language models like DeepSeek V3 — demonstrating that these models benefit not just from scale, but from principled task representations. While developed for mathematics, our approach generalizes to other domains that demand precise reasoning, such as formal verification, clinical decision-making, and software engineering. Moreover, it is model-agnostic, offering a path toward more principled, structure-aware reasoning systems of the future.

References

[1] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *Advances in neural information processing systems*, 30, 2017.

[2] Emilio Jesús Gallego Arias. Serapi: Machine-friendly, data-centric serialization for coq. 2016.

[3] Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics. *arXiv preprint arXiv:2310.10631*, 2023.

[4] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. The tactician: A seamless, interactive tactic learner and prover for coq. In *International Conference on Intelligent Computer Mathematics*, pages 271–277. Springer, 2020.

[5] Haskell B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[6] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 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*, pages 378–388. Springer, 2015.

[7] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering, pages 1229–1241, 2023.

[8] Andreas Florath. Enhancing formal theorem proving: a comprehensive dataset for training ai models on coq code. *arXiv preprint arXiv:2403.12627*, 2024.

[9] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. *arXiv preprint arXiv:2102.06203*, 2021.

[10] William Alvin Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[11] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. Deepmath-deep sequence models for premise selection. *Advances in neural information processing systems*, 29, 2016.

[12] Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers. *Advances in Neural Information Processing Systems*, 35:8360–8373, 2022.

[13] Saketh Ram Kasibatla, Arpan Agarwal, Yuriy Brun, Sorin Lerner, Talia Ringer, and Emily First. Cobblestone: Iterative automation for formal verification. *arXiv preprint arXiv:2410.19940*, 2024.

[14] Andrei Kozyrev, Gleb Solovev, Nikita Khramov, and Anton Podkopaev. Coqpilot, a plugin for llm-based generation of proofs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2382–2385, 2024.

[15] Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349, 2022.

[16] Haohan Lin, Zhiqing Sun, Sean Welleck, and Yiming Yang. Lean-star: Learning to interleave thinking and proving. *arXiv preprint arXiv:2407.10040*, 2024.

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

[18] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[19] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. Proof automation with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1509–1520, 2024.

[20] Maciej Mikuła, Szymon Tworkowski, Szymon Antoniak, Bartosz Piotrowski, Albert Qiaochu Jiang, Jin Peng Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. Magnushammer: A transformer-based approach to premise selection. *arXiv preprint arXiv:2303.04488*, 2023.

[21] Leonardo de Moura and Sebastian Ullrich. 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, 2021.

[22] Azim Ospanov and Roozbeh Yousefzadeh. Apollo: Automated llm and lean collaboration for advanced formal reasoning. *arXiv preprint arXiv:2505.05758*, 2025.

[23] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.

[24] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. *arXiv preprint arXiv:2202.01344*, 2022.

[25] Jason Rute, Miroslav Olšák, Lasse Blaauwbroek, Fidel Ivan Schaposnik Massolo, Jelle Piepenbrock, and Vasily Pestun. Graph2tac: Learning hierarchical representations of math concepts in theorem proving. 2024.

[26] Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. A language-agent approach to formal theorem-proving. 2023.

[27] The Coq Development Team. Coq.

[28] Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025.

[29] Haiming Wang, Huajian Xin, Chuanyang Zheng, Lin Li, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, et al. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656*, 2023.

[30] Ruida Wang, Rui Pan, Yuxin Li, Jipeng Zhang, Yizhen Jia, Shizhe Diao, Renjie Pi, Junjie Hu, and Tong Zhang. Ma-lot: Multi-agent lean-based long chain-of-thought reasoning enhances formal theorem proving. *arXiv preprint arXiv:2503.03205*, 2025.

[31] Ruida Wang, Jipeng Zhang, Yizhen Jia, Rui Pan, Shizhe Diao, Renjie Pi, and Tong Zhang. Theoremllama: Transforming general-purpose llms into lean4 experts. *arXiv preprint arXiv:2407.03203*, 2024.

[32] Sean Welleck and Rahul Saha. Llmstep: Llm proofstep suggestions in lean. *arXiv preprint arXiv:2310.18457*, 2023.

[33] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35:32353–32368, 2022.

[34] Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Jiayu Wang, Dahua Lin, and Kai Chen. Internlm2. 5-stepprover: Advancing automated theorem proving via expert iteration on large-scale lean problems. *arXiv preprint arXiv:2410.15700*, 2024.

[35] Huajian Xin, ZZ Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, et al. Deepseek-prover-v1. 5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *arXiv preprint arXiv:2408.08152*, 2024.

[36] Yutong Xin, Jimmy Xin, Gabriel Poesia, Noah Goodman, Qiaochu Chen, and Isil Dillig. Automated discovery of tactic libraries for interactive theorem proving. *arXiv preprint arXiv:2503.24036*, 2025.

[37] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

[38] Yang Yuan. Succinct representations for concepts. *arXiv preprint arXiv:2303.00446*, 2023.

[39] Yifan Zhang, Yifan Luo, Yang Yuan, and Andrew C Yao. Autonomous data selection with language models for mathematical texts. *ICLR 2024 Workshop on Navigating and Addressing Data Problems for Foundation Models*, 2024.

A Technical Implementation Details

This part provides detailed technical information about our data processing pipeline and implementation choices.

A.1 Entity Extraction

Implementation Overview Coq's compilation pipeline transforms user-written definitions through multiple stages:

$$\text{Raw AST} \xrightarrow{\text{parsing}} \texttt{glob\_constr} \xrightarrow{\text{pre-typing}} \texttt{constr} \xrightarrow{\text{type inference}} \texttt{typed\_constr}$$

At the `typed_constr` stage, Coq has performed complete type inference and elaboration—all implicit arguments are computed, universe levels are instantiated, and every subterm has been assigned its precise type. We intercept at this stage to capture the fully-elaborated internal representation (i) alongside the original surface syntax (o), while the entity type (c) and name (n) are extracted during vernacular command processing.

Kernel Term Processing Every high-level Coq construct (theorems, definitions, fixpoints, etc.) is internally compiled down to expressions built from 18 fundamental kernel constructors. For example, a theorem statement becomes a `Prod` (forall) expression, while its proof term might combine `Lambda` (function), `App` (application), and `Ind` (inductive type) constructors. We implement a recursive `constr_to_string` method that processes these kernel terms at three levels of detail:

- **Pure constr**: Raw kernel representation
- **Standard constr**: Kernel terms with resolved variable names
- **Processed**: Simplified representation for language models

Local Variable Context Extraction During the compilation process, Coq automatically infers and tracks the types of all local variables. When we intercept at the `glob_constr` stage during pre-typing, we capture all `GVar` references along with their inferred types. This type information becomes naturally embedded in the internal representation.

This context extraction is crucial for semantic clarity. Consider a theorem stating $a + b = b + a$:

- With $a, b : \mathbb{N}$, this is commutativity of natural number addition
- With $a, b : \text{list } A$, this would be about list concatenation (likely false)
- With $a, b : \text{bool}$, this might represent logical operations

The same syntactic pattern has entirely different meanings depending on types. By extracting type annotations implicit in surface syntax, we provide models with the semantic context necessary for type-aware reasoning.

Key Technical Decisions

- **Environment Trimming for De Bruijn Index Resolution**: In Coq's kernel representation, variables are represented using De Bruijn indices—a nameless representation where variables are identified by their position in the binding context rather than by name. For example, in the term $\lambda x.\lambda y.x + y$, the variable y is represented by index 0 (most recently bound) and x by index 1 (bound one level up).

  To correctly resolve these indices back to meaningful variable names, we must maintain a precise environment that mirrors the binding structure. Consider processing the term $\forall(x : \text{nat}), \forall(y : \text{nat}), x + y = y + x$:

  – When processing the outer $\forall$, the environment is empty
  – After binding x, the environment becomes $[x]$
  – After binding y, the environment becomes $[y, x]$ (with y at index 0)
  – When encountering index 1 in the body, we resolve it to x

  - When encountering index 0, we resolve it to y

  The challenge arises with Coq's global environment containing thousands of definitions. When processing nested terms, we must trim this global context to match each subterm's local binding context. This trimming ensures that De Bruijn indices correctly map to their intended variables, handling the complexity of deeply nested scopes, pattern matching branches, and local definitions.

- **Kernel Name Disambiguation**: Coq's module system creates naming complexity—the same entity can have different user-facing names. For instance, `Z.quotrem` might be defined in `BinIntDef` but re-exported through `BinInt`. We create globally unique identifiers by combining the kernel name (internal unique identifier) with the canonical path (user-facing name):

  ```
  Coq.ZArith.BinInt.Z.quotrem<ker>Coq.ZArith.BinIntDef.Z.quotrem
  ```

  This ensures unambiguous entity references regardless of import paths or module aliases.

Through this extraction pipeline, we capture the complete Coq Entity tuple (n, c, o, i) from Definition 1. By intercepting Coq's compilation at multiple stages and implementing comprehensive kernel term processing, we provide language models with parallel representations—surface syntax alongside fully-elaborated internal forms—enabling them to learn the relationship between what users write and what Coq actually computes.

A.2 Proof State Extraction

Implementation Overview To capture the interactive proof sequences defined in Definition 3, we instrument Coq's tactic interpreter to record proof states before and after each tactic application. For each tactic T_i, we capture the complete proof state σ_{i-1} at the entry point and σ_i after execution, extracting both surface and internal representations (H^o, H^i, G^o, G^i) for every goal. The internal representations are processed using the same `constr_to_string` method from entity extraction, as hypotheses and goals are ultimately combinations of the same 18 kernel constructors.

Key Technical Decisions

- **Tactic Linearization**: Coq's semicolon operator allows parallel goal processing, where a single tactic can be applied to multiple goals simultaneously. This creates non-linear proof structures that are challenging for sequence models to learn. We modify Coq's interpreter to use `Goal.enter`, which isolates each goal in its own environment and processes them sequentially. This linearization transforms complex parallel proof steps into sequences that align with Definition 3. For rare cases with sequential dependencies between goals (less than 1% of proofs), we provide a compiler flag to revert to parallel processing.
  - **Data Amplification**: This linearization significantly amplifies training data—a single compound tactic operating on n goals in parallel now generates n distinct proof state transitions, exposing intermediate states that would otherwise be hidden. This fine-grained decomposition provides models with detailed proof trajectories essential for learning tactical reasoning.
- **Ltac Expansion**: User-defined Ltac tactics appear as single atomic steps in surface-level proofs but often expand into complex sequences of primitive tactics. We implement configurable-depth expansion of these tactics, recursively unfolding them to reveal their underlying proof patterns. This expansion is particularly important for tactics with branching structures, where we ensure consistent depth tracking across all branches to maintain structural clarity.
- **Meaningful Transition Filtering**: We validate state transitions by tracking changes in goal structures and existential variable (evar) maps between proof states. This ensures we capture all tactics that modify the proof state, including those that instantiate evars or restructure goals without visibly changing the goal count. Only tactics that genuinely transform the proof state are recorded, providing models with semantically meaningful proof steps.

This extraction pipeline captures complete interactive proofs as defined in Definition 3, with each proof step containing full before/after states including all hypotheses and goals in both surface and internal forms. The linearization and expansion mechanisms expose the fine-grained structure of proof construction, enabling language models to learn from detailed proof trajectories rather than opaque high-level scripts.

A.3 Domain-specific Tokenizer

Design Principle With entities and proof states extracted, we construct a tokenizer that preserves Coq's semantic identity. The core principle, formalized in Definition 4, ensures that tokens represent semantic meaning rather than surface syntax: identical Coq entities receive the same token regardless of how they are referenced (e.g., `Nat.add` and `Coq.Init.Nat.add`), while distinct entities always receive different tokens.

Token Categories Our tokenizer assigns unique integer identifiers to:

- **Global Identifiers**: Fully-qualified names from extracted entities.
  - Inductive types: Each constructor receives its own global ID (e.g., `nat`, `O`, and `S` are assigned distinct tokens)
  - Recursive definitions: Individual IDs for each function in mutually recursive groups, preserving their semantic independence
  - Namespace resolution: Pattern matching strips module and section prefixes to distinguish truly global entities from locally-scoped definitions that appear with qualified names due to Coq's scoping mechanisms
- **Local Variables**: Dynamic identifiers for bound variables, where the token represents the variable's type rather than its name. For instance, in `Definition f a b c : nat := a + b + c`, the variables `a`, `b`, and `c` are tokenized based on their shared type `nat`, reflecting that their specific names are irrelevant to the semantic meaning.
- **Reserved Tokens**: Coq keywords, syntactic markers, hint databases, internal tactics, and special tokens we introduce for proof state representation (e.g., `_Anonymous` for unnamed hypotheses, `goalcompleted` for solved goals, `REL` for De Bruijn indices).

Coverage and Limitations The tokenizer achieves 99.8% coverage of Coq constructs in our corpus. The remaining 0.2% falls back to default handling for two reasons:

- entities we intentionally exclude from extraction, such as primitive axioms and low-level type theory constructs that are not relevant to typical proofs
- corner cases in Coq's syntax that our pattern matching does not capture.

When encountering these cases, the tokenizer defaults to treating them as local variables with empty type information. This fallback strategy ensures robustness without impacting the quality of tokenization for mainstream Coq proofs.

This semantic tokenization enables models to learn from mathematical content rather than syntactic variations, providing the foundation for the Planner-Executor architecture described next.

B Examples of Semantic Information Types

To clarify the different forms of semantic information used in our evaluation—**Origin**, **Internal**, and **Intuition**—we present two examples below. Each example corresponds to a formal concept from Coq's libraries, with its representation shown under the three semantic views.

B.1 Example 1: Fixpoint Construction via `FixFun`

This example uses the definition `TLC.LibFix.FixFun`, a fixed-point combinator to define recursive functions on types with inhabited codomain.

Origin:

```
TLC.LibFix.FixFun A B {IB : Inhab B} (F : (A -> B) -> A -> B) : A -> B :=
  FixFunMod eq F
```

Internal:

```
fun ( A : Type ) =>
  fun ( B : Type ) =>
    fun ( IB : ( TLC.LibLogic.Inhab.Inhab B ) ) =>
      fun ( F : forall ( _Anonymous : forall ( _Anonymous : A ) -> B ) ->
                  forall ( _Anonymous : A ) -> B ) =>
        ( TLC.LibFix.FixFunMod A B IB ( Coq.Init.Logic.eq.eq B ) F )
```

Intuition:

> This function provides a way to define recursive functions in a non-recursive language by finding a fixed point of a higher-order function, ensuring termination and correctness through the use of inhabited types and equality.

B.2 Example 2: Well-Founded Recursion via `FixWf`

This example uses `Equations.Type.Subterm.FixWf`, a definition that supports well-founded recursion in dependent type theory.

Origin:

```
Equations.Type.Subterm.FixWf
  `{WF : WellFounded A R}
  (P : A -> Type)
  (step : forall x : A, (forall y : A, R y x -> P y) -> P x)
  : forall x : A, P x :=
  Fix wellfounded P step
```

Internal:

```
fun ( A : Type ) =>
  fun ( R : Equations.Type.Relation.relation A ) =>
    fun ( WF : Equations.Type.Classes.WellFounded A R ) =>
      fun ( P : A -> Type ) =>
        fun ( step : forall ( x : A ) ->
                       forall ( y : A ) ->
                       forall ( _ : R y x ) -> P y -> P x ) =>
          Equations.Type.WellFounded.Fix
            A R
            (Equations.Type.Classes.wellfounded A R WF)
            P step
```

Intuition:

> This theorem provides a way to define recursive functions on a type `A` with a well-founded relation `R`, ensuring that the recursion terminates by only allowing recursive calls on elements that are 'smaller' according to `R`. It is a foundational tool for defining well-behaved recursive functions in dependent type theory.

B.3 Summary

These examples illustrate how semantic information about a Coq concept can be represented in different forms:

- **Origin**: The original Coq source code, written by developers.
- **Internal**: A machine-level, compiler-oriented abstraction.
- **Intuition**: A natural language explanation that communicates conceptual meaning to humans.

These formats serve as distinct inputs to evaluate the level of understanding that a model comprehends formal concepts.

C Prompt Template Used for Coq Proof Generation Process

The following prompt template is employed to represent the current proof state and context in our experiments. This template captures essential elements such as hypotheses, goals, referenced global definitions, proof tracing history, related premises and tactics, curated notes, hints, and available user actions. It is designed to guide the model to either request additional information or suggest tactics for proof progression.

```
I am currently working on a formal proof in Coq. Here is my current state and context:

=== Current Proof States ===
# Hypotheses:
{hyps}

# Goal:
{goal}

Global definitions referenced:
# Glob def:
{glob_def}
=== Proof Tracing ===
This shows how we reached the current state through previous tactics:

Tactics: {tactic_seq}
{proof_summary}

=== Related Premises ===
Potentially relevant premises (for reference only):
{premises}

=== Related Tactic ===
Commonly used tactics for similar proofstates (for reference only):
{tactics}

=== Public Notes ===
Curated insights relevant to current proof:
{public_notes}

=== Hint ===
Some hints may help you to understand the proof:
{hint}

=== Available Actions ===

Please choose ONE of the following actions:

1. Request more information about specific concepts/tactics mentioned above
Your response must be in this format:
{{
  "info": ["concept_name1", "concept_name2", "tactic1", "tactic2", ...]
}}

2. Suggest a list of up to 10 tactics to try - prefer single atomic tactics over compound
     ↪  ones unless the combination is highly confident. I will provide the compiler's
     ↪ response for each
Your response must be in this format:
{{
  tactics: [
    {{"tactic": "tactic1", "reason": "explanation for why this specific tactic is
     ↪ recommended"}},
    {{"tactic": "tactic2", "reason": "explanation for why this specific tactic is
     ↪ recommended"}},
    ...
  ]
}}
```

This prompt format provides a consistent interaction framework across all conditions. The different information configurations are instantiated by varying the content inserted into specific fields such as `glob_def` and `public_notes`.