CNLVERIFIER: A CONTROLLED-NATURAL-LANGUAGE VERIFIER FOR LLM-DRIVEN MATHEMATICAL REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models can generate mathematical proofs in natural language, but such informal proofs are difficult to use as reliable, automatic, and actionable feedback for post-training and inference-time compute. Formal proof assistants provide strong correctness guarantees, but their proof languages are often costly for current models to generate and interact with. We present **CNLVerifier** (Controlled Natural Language Verifier), a lightweight verifier for proofs written in a controlled, LaTeX-flavored natural language. **CNLVerifier** provides structured diagnostics that can serve both as a binary acceptance signal and as feedback for self-refinement. We use these signals for verifier-filtered supervised fine-tuning and reinforcement learning. Experiments on miniF2F show that verifier-based training and inference-time compute improve the generation of verifier-accepted CNL proofs.

1 INTRODUCTION

Large language models (LLMs) can generate long mathematical proof attempts in natural language, but these proofs are difficult to verify reliably and automatically. This limits their use as feedback for inference-time compute and post-training, where the verifier should not only reject invalid reasoning but also identify where a proof attempt fails and how to fix it. Existing approaches based on final-answer matching (Shao et al., 2024; Wang et al., 2025b), self-consistency (Wang et al., 2023; Liu et al., 2025), LLM-as-a-judge (Zheng et al., 2023; Stephan et al., 2025), or automated evaluators (Zhang et al., 2025; Ma et al., 2026) provide useful supervision, but they do not enforce a mechanically checkable proof structure.

A natural way to obtain reliable verification is to use formal proof languages and proof assistants such as Lean (De Moura et al., 2015) and Isabelle (Wenzel et al., 2008), which provide strong correctness guarantees by checking derivations against a trusted logical foundation. Recent LLM-based theorem provers, particularly Goedel-Prover (Lin et al., 2025) and Goedel-Prover-V2 (Lin et al., 2026), have demonstrated that LMs can effectively interact with formal verifiers and use verifier feedback for proof generation and self-refinement. However, fully formal proof languages remain a heavy interface for LLM reasoning, and as LLMs generate ever more mathematics, making their formal statements understandable to human mathematicians is an important challenge. Lean-style proofs are optimized for machine checking rather than readability: arguments require highly structured syntax and substantial interaction with formal libraries and proof environments (Agrawal et al., 2022; Yang et al., 2026). This motivates a lighter verification interface that preserves machine checkability while remaining closer to natural language reasoning—and thus easier for mathematicians to read.

Controlled Natural Language (CNL) provides a middle ground between informal reasoning and fully formal proof languages. Systems such as Naproche and MathNat show that mathematical prose can remain human-readable while retaining enough structure for automated checking (Cramer et al., 2009; Humayoun & Raffalli, 2010). We adopt this idea for LLM reasoning.

Our contributions. We introduce **CNLVerifier**, a verifier for LLM-generated mathematical proofs in CNL, with a unified interface for verification, structured feedback, and verifier-based training and inference. Our contributions are:

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

- **A human-readable language for LLM-generated proofs.** We design a LaTeX-flavored CNL in which proof steps follow a small set of sentence templates while mathematical expressions remain in standard LaTeX. The resulting language is easy for LLMs to generate and structured enough for deterministic parsing and verification, and readily readable by mathematicians—addressing the growing need to understand AI-generated formal mathematics.
- **An efficient verifier for step-by-step proof checking.** We implement a five-stage verification pipeline (preprocess, parse, translate, typecheck, and kernel/tactic checking) that verifies each asserted proof step. The verifier provides approximately 65 tactics covering common algebraic, arithmetic, logical, and set-theoretic reasoning.
- **Verifier-based post-training and inference-time compute.** We demonstrate that **CNLVerifier** can support verifier-filtered SFT, RL with kernel-based rewards, and inference-time self-refinement through structured diagnostic feedback.

2 Related Work

LLMs for formal theorem proving. A large body of work uses language models to interact with proof assistants, including GPT-f (Polu & Sutskever, 2020), PACT (Han et al., 2022), LeanDojo (Yang et al., 2023), and DeepSeek-Prover and its successors (Xin et al., 2024a;b; Ren et al., 2025). Particularly relevant are Goedel-Prover (Lin et al., 2025) and Goedel-Prover-V2 (Lin et al., 2026), which combine large-scale formal proof generation with verifier-guided refinement and self-correction. Related systems also use compiler feedback to repair formal proofs, including Draft-Sketch-Prove (Jiang et al., 2023) and Kimina-Prover (Wang et al., 2025a). CNLVerifier follows the same verifier-in-the-loop principle, but uses a restricted LaTeX-style CNL rather than a full formal proof language.

Autoformalization and controlled mathematical languages. Autoformalization translates natural-language mathematics into formal languages such as Lean (Wu et al., 2022; Ying et al., 2025). In contrast, CNLVerifier verifies controlled prose directly, removing proof-body autoformalization from the loop; problem statements are still provided as hand-written CNL prefixes. Our language is related to controlled and declarative mathematical languages such as ForTheL/Naproche (Cramer et al., 2009), Mizar (Grabowski et al., 2010), Isar (Wenzel, 1999), MathNat (Humayoun & Raffalli, 2010), GFLean (Pathak, 2024), and Verbose Lean 4 (Massot, 2024). Elfe and Diproche (Doré & Broda, 2018; Carl et al., 2022) similarly use verification as feedback, but target human learners rather than LLM agents. CNLVerifier instead uses a small sentence grammar, idiomatic LaTeX expressions, and machine-readable error records intended for model consumption.

Verification and verifier-filtered training. Informal mathematical reasoning is commonly evaluated by final-answer matching, self-consistency (Wang et al., 2023), or process-reward models (Lightman et al., 2023). These signals do not mechanically certify the full derivation. Automatic checkers have also been widely used for rejection sampling and self-improvement in code and reasoning (Chen et al., 2023; Zelikman et al., 2022; Gulcehre et al., 2023; Yuan et al., 2023). Our approach similarly filters generated data with an automatic verifier, but uses CNL proof verification rather than program execution or a full formal proof assistant.

3 Syntax and Semantics

A CNL proof has two nested layers. The *outer* layer is a controlled subset of LaTeX-flavored natural language: every line begins with one of a small set of sentence openers (Table 1), and a non-matching line is rejected at parse time. Comments opened by `\informal` or `\wild` are ignored and never enter the kernel. The *inner* layer lives between dollar signs: each fragment is a term or formula of a many-sorted first-order logic (FOL). This section specifies both layers and the map into FOL; block structure is in §3.4.

3.1 CNL syntax: sentence openers

Table 1 lists the only sentence openers a CNL proof can use. A line that does not match an opener fails to parse, and the proof is rejected; comment lines opened by `\informal` or `\wild` are ignored

Opener	Role	Example
`Let`	introduce a variable or set	`Let $x\in\mathbb{Z}$.`
`Assume`	introduce a hypothesis	`Assume $0<y$.`
`We have`	assert a derivable fact	`We have $x^2\ge 0$.`
`Then`	assert a derivable fact, chained	`Then $x=5$.`
`The goal is to prove`	state the top-level goal	`The goal is to prove $N=1$.`
`It's enough to prove`	strengthen / reduce the goal	`It's enough to prove $...$.`
`We prove ... by cases`	open a case block	see §3.4
`We prove ... by induction`	open an induction block	see §3.4
`Contradiction.`	contradiction in current branch	`Contradiction.`
`\item Case`	a branch inside a block	`\item Case $x>0$.`
`\informal{...}`	natural-language comment	`\informal{...}`

Table 1: Allowed CNL sentence openers. Non-opener lines fail parse and the proof is rejected; `\informal`/`\wild` comments are ignored.

and never enter the kernel. Internally the openers map onto templates of the shape

$$\texttt{'Let \{clarify\}.'} \Longrightarrow \texttt{let formula},$$

where slots such as `{prop}`, `{clarify}`, `{set}`, `{formula}` are filled by a Lark context-free grammar (CFG). The parser is intentionally template-driven: only legal sentence shapes pass, and each parsed line carries a tag (`goal`, `have`, `let`, `Assume`, `then`, `cases_start`, `induction_case`, ...) that drives the kernel's control flow.

3.2 FOL SYNTAX: SORTS, SIGNATURE, TERMS, AND FORMULAS

Every math fragment `$...$` denotes an object of a many-sorted FOL with equality. The base sorts are

$$\mathcal{S} = \{\, \mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}, \mathbb{B}, \mathrm{Set} \,\} \cup \{\, \mathbb{F}_p \mid p \text{ prime} \,\} \cup \{\, \mathbb{Z}/n\mathbb{Z} \mid n \in \mathbb{Z}_{>1} \,\},$$

with $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}$. The unparameterized sort Set classifies collections; nested collections still have sort Set, and element sorts come from membership hypotheses. The surface form `\mathbb{Z}_p` denotes $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$, not the p-adic integers; $\mathbb{Z}/n\mathbb{Z}$ is a separate family of sorts.

Sort promotion fires only at overload resolution: an exact signature match is tried first, otherwise arguments may be copied along the numeric chain. Declared variables keep their sort in Γ.

Definition 3.1 (Signature, terms, atomic formulas, and formulas)**.** *Fix a many-sorted signature $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{P}, \mathrm{sort})$, where $\mathcal{S}$ is the set of sorts above, $\mathcal{F}$ is a set of function symbols (constants are 0-ary functions), $\mathcal{P}$ is a set of predicate symbols, and* sort *assigns each symbol a sorted arity: $f : (\sigma_1, \ldots, \sigma_n) \to \sigma$ for $f \in \mathcal{F}$ and $P : (\sigma_1, \ldots, \sigma_n) \to \mathbb{B}$ for $P \in \mathcal{P}$. Terms t, atomic formulas A, and formulas φ are defined by mutual induction:*

$$\begin{aligned} t &::= x \mid c \mid f(t_1, \ldots, t_n) \\ A &::= P(t_1, \ldots, t_n) \\ \varphi &::= A \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \to \psi \mid \varphi \leftrightarrow \psi \mid \forall x{:}\sigma.\, \varphi \mid \exists x{:}\sigma.\, \varphi \mid \exists! x{:}\sigma.\, \varphi, \end{aligned}$$

where x is a variable of sort σ, c ranges over constants, and each t_i has the sort prescribed by the arity of f or P.

3.3 SEMANTICS: FROM CNL MATH FRAGMENTS TO FOL

Parsing a fragment yields an AST; a denotation $[\![\cdot]\!]_\Gamma$ maps it to a FOL term or formula of Definition 3.1. Sort tokens map in the obvious way ($[\![\texttt{\textbackslash mathbb\{Z\}}]\!] = \mathbb{Z}$, $[\![\texttt{\textbackslash mathbb\{Z\}_\{p\}}]\!] = \mathbb{F}_p$, ...), and connectives are homomorphic. Relation symbols map to internal predicates (Appendix A.1).

Definition 3.2 (Quantifier mapping)**.** *Let $\bar{x} = x_1, \ldots, x_k$ be variables declared with domain D, and write $\sigma(D)$ for the* ambient sort *of D: D itself if $D \in \mathcal{S}$, otherwise the sort of its elements. Binding is always at a sort.*

$$[\![\forall \bar{x} \in D,\ \varphi]\!]_\Gamma = \forall x_1{:}\sigma(D) \cdots \forall x_k{:}\sigma(D).\, \mathit{guard}(D, \bar{x}) \to [\![\varphi]\!]_{\Gamma, \bar{x}:\sigma(D)}$$
$$[\![\exists \bar{x} \in D,\ \varphi]\!]_\Gamma = \exists x_1{:}\sigma(D) \cdots \exists x_k{:}\sigma(D).\, \mathit{guard}(D, \bar{x}) \wedge [\![\varphi]\!]_{\Gamma, \bar{x}:\sigma(D)}$$

where guard is membership in D when D is a set term, and $\top$ when D is already a sort. Surface $\exists!$ expands as $\exists x.\, \varphi \wedge \forall y.\, (\varphi[x \mapsto y] \to y = x)$. The kernel still stores $\exists!$ as a primitive for α-equality; Table 2 has no dedicated rule, so uniqueness steps are discharged by a tactic or endorsement.

3.4 Special blocks: cases, induction, suffices

The more common special modules in proofs are case analysis and mathematical induction. A case analysis is opened by `We prove φ by cases:` and bracketed by a LaTeX `itemize` whose items are case headers. Induction reuses the case-analysis surface shape but fixes a *two-branch* pattern: a base case `\item Case $n=b$.` and an inductive case `\item Case $n>b$.`, where b is a numeric constant appearing in both headers (typically 0 or 1). The inductive branch does *not* receive the induction hypothesis automatically: the proof must state it with an explicit `Assume $...$.` line. The kernel expects that hypothesis to be α-equal to the literal formula $\varphi(n-1)$ obtained by replacing the induction variable in the block goal $\varphi(n)$ with $n-1$; if the prose hypothesis is equivalent but not literally that shape (for example a shifted sum index), assembly requires an independently verified bridge before the step branch is admitted into the schema. Each block uses the *final-goal-only* export policy. Like cases and SUFFICES, a completed induction block is assembled into a single kernel theorem (§4.2): the base and step branches are discharged and generalised, then closed through a registered induction-schema instance together with the block's completeness obligation $n = b \vee n > b$.

4 The CNL Verifier

4.1 System overview

The agent transforms a natural-language problem into a CNL proof; the verifier accepts or rejects, and on rejection emits a `VerifierError` record that the agent uses to repair the proof. The same verifier is re-used as a *filter* in the offline distillation pipeline of Section 5.

Two modules must be distinguished from the start. The *driver* walks the CNL line by line, maintains the proof tree, and dispatches tactics. The *trusted kernel* Section 4.2 is the only module that can construct a theorem.

A CNL proof passes through five stages. **Preprocess.** Each problem is a *CNL prefix* (`Let`/`Assume` hypotheses and one `The goal is to prove` line); fill-in-the-blank goals contain '`{?}`', and the kernel always checks `prefix` + `completion`. **Parse.** The surface is matched against Section 3.1. **Translate.** The translator implements $[\![\cdot]\!]_\Gamma$ (Section 3.3), with specialised handlers for idioms such as "p is prime" or "f is a bijection". **Typecheck.** Every assumption, goal, and intermediate fact is checked against the numeric tower and finite-field sorts before it enters the tree; failures become codes such as `E_TYPE_VAR_UNDECLARED`. **Drive.** The driver maintains a tree of `ProofNodes`, each with a `ProofState` and completion flags, and tracks summary nodes (`cases`, `induction`, `suffices`, `lemma`) so that branch-local facts cannot leak. A `We have φ.` or `Then φ.` line sets the current goal to φ and dispatches tactic search; conjunctive goals are split first. The gateway is responsible for certifying theorem-backed proof steps (Section 4.2).

Structured verifier feedback. When verification fails, **CNLVerifier** returns a structured diagnostic rather than a binary rejection. Each `VerifierError` records the verification phase, error code, offending statement and location, diagnostic message, and a repair hint. The phase indicates whether the failure occurred during preprocessing, parsing, translation, type checking, tactic search, or kernel verification. These diagnostics are passed back to the model as feedback for proof refinement, allowing it to distinguish and repair different types of verification failures.

4.2 The trusted kernel

Tactics are responsible for proof search, while the kernel is responsible for proof certification. Rather than relying directly on a tactic's success signal, the verifier uses kernel theorem objects to certify proof steps. A theorem object can be constructed only through the kernel's inference rules and marked extension points.

Group	Kernel rules	Natural-deduction reading
Structural	`assume`, `weaken`	assumption, thinning
Implication	`impl_intro`, `impl_elim`	$\to$I (discharge), $\to$E (modus ponens)
Conjunction	`conj_intro`, `conj_elim`	$\wedge$I, $\wedge$E
Disjunction	`disj_intro_left/right`, `disj_elim`	$\vee\mathrm{I}_1$, $\vee\mathrm{I}_2$, $\vee$E
Negation and falsum	`neg_intro`, `neg_elim`, `falsum_elim`†, `by_contradiction`‡	$\neg$I, $\neg$E, $\bot$E, RAA
Universal	`forall_intro`, `forall_elim`	$\forall$I (eigenvariable), $\forall$E
Existential	`exists_intro`, `exists_elim`	$\exists$I, $\exists$E (eigenvariable)
Biconditional	`iff_intro`, `iff_elim_left/right`	$\leftrightarrow$I, $\leftrightarrow$E
Equality	`refl`, `eq_subst`	reflexivity, Leibniz substitution
Extension points	`new_axiom`, `oracle`	background theory, per-instance endorsement

Table 2: Complete kernel rule set. Judgements are sequents $\Gamma \vdash \varphi$. † *ex falso* (intuitionistic); ‡ *reductio* (classical).

LCF architecture. Following HOL Light Harrison (2009), the kernel exports an abstract type `Thm` of sequents $\Gamma \vdash \varphi$ together with twenty-two inference rules and two marked extension points (Table 2). The constructor is private, so every `Thm` is produced by one of those rules. Soundness is then induction on construction history: each rule is a short, side-effect-free shape check. Tactics and heuristics live outside this boundary; a bug there may cause proof search to fail, but cannot by itself produce a kernel theorem.

The kernel implements its own representation normalisation (so rules see one formula AST), capture-avoiding substitution with on-demand α-renaming (refusing substitutions it cannot rename safely), and α-equality, rather than importing these from the untrusted term library.

The gateway. A search-free gateway sits between tactics and the kernel. It hosts *derived rules* (equality symmetry/transitivity, congruence, n-ary $\vee$E, SUFFICES composition, induction-block assembly, hypothesis cut), each built from Table 2. It exposes `certify`, which attaches a theorem to a tactic result by, in order, a derivation the tactic supplied, a derivation the gateway can find, or a labelled oracle (Section 4.3). Finally it enforces acceptance: a theorem discharges a goal only if its conclusion is α-equal to the goal *and* its hypotheses are a subset of those currently available. The subset direction is weakening; the converse would smuggle case-local hypotheses outward.

Block assembly and induction. A certified theorem is reused when the enclosing block is assembled. Cases fold $\Gamma \vdash D_1 \vee \cdots \vee D_n$ with branch theorems $\Gamma, D_i \vdash C$ through `disj_elim`; SUFFICES uses `impl_intro` and modus ponens. An induction block discharges

$$\forall n.\,(n = b \to \varphi(n)) \quad \text{and} \quad \forall n.\,(n > b \to (\varphi(n-1) \to \varphi(n))),$$

registers the corresponding schema instance via `new_axiom`, and closes with modus ponens against $n = b \vee n > b$ and a solver-checked lemma $n = b \vee n > b \vdash n \geq b$. FOL cannot state a single second-order induction principle, so a per-formula schema is the standard first-order treatment (Kaye, 1991; Hajek & Pudlak, 1998), as in ACL2 and Vampire (Kaufmann et al., 2000; Hozzová et al., 2021).

Two notions of acceptance. Of 57 tactic modules, 15 currently produce kernel derivations or per-instance solver certificates; the rest report success without returning a derivation or requesting an endorsement. A proof is thus *driver-accepted* when the driver marks every node complete and every block exhaustive—a property of its bookkeeping flags, which a tactic can set by reporting success alone—and *theorem-backed* when the gateway additionally holds a final theorem whose conclusion is α-equal to $[\![G]\!]$ and whose hypotheses are a subset of $[\![\Gamma]\!]$. The second implies the first but not conversely; Theorem 4.1 is stated for the second, and §4.4 reports how often it is met.

Tactics. Tactics operate outside the trusted kernel: they search for derivations or request endorsements, but cannot construct a `Thm` directly. Each proof step is represented by a goal together with its available assumptions, including the original premises and previously certified intermediate conclusions.

4.3 Acceptance and trust accounting

The kernel is a logical engine over uninterpreted symbols. Mathematical content enters through two doors, and the accounting below turns the doors a given accepted proof used into a recorded, and therefore measurable, quantity.

Definition 4.1 (Axioms and endorsements)**.** *A* registered axiom *is a formula admitted permanently into the background theory through* `new_axiom`*. We admit only definitional or schema-level statements this way: the definition of $\neq$ at each sort, $\top$, and, when an induction block is assembled, the instance of the induction schema displayed in §4.2, registered under a content-dependent name* `induction:{sort}:{hash}` *and listed in* `kernel.axioms()`*. Decidable one-off facts such as $x^2 \geq 0$ are deliberately* not *axioms: they are re-checked on each instance, since an axiom stated wrongly is wrong forever.*

An endorsement *is a triple (ℓ, Δ, ψ) recording that an external procedure named ℓ vouched for the single sequent $\Delta \vdash \psi$. Endorsements enter through* `oracle`*, which returns the theorem $\Delta \vdash \psi$ carrying the* label *ℓ, an identifier for that procedure. Labels propagate by union through every rule of Table 2, so the label set of a theorem is the set of endorsements its derivation transitively depends on. For a run of the verifier on a proof P we write $\mathcal{A}$ for the registered axioms and $\mathcal{E}(P)$ for the endorsements it invoked.*

Write ND for derivability using the natural-deduction rules of Table 2—every row except the two extension points `new_axiom` and `oracle`—and ND $+ \mathcal{A} + \mathcal{E}$ for derivability when each registered axiom and each endorsed sequent may additionally be used as a leaf.

Theorem 4.1 (Audit completeness)**.** *Let P be theorem-backed with final theorem θ and label set O. Then $[\![\Gamma]\!] \vdash_{\mathrm{ND}+\mathcal{A}+\mathcal{E}(P)} [\![G]\!]$, and every leaf of the derivation of θ is an instance of* `assume` *or* `refl`*, a registered axiom in $\mathcal{A}$, or an endorsement in $\mathcal{E}(P)$ whose label lies in O; conversely every label in O marks a leaf the derivation uses. Hence O is neither an over- nor an under-approximation: it is exactly the set of external procedures on which P depends.*

Proof sketch. By induction on the construction of θ, which is well founded because the constructor of `Thm` is private. The base cases `assume`, `refl`, `new_axiom` and `oracle` are exactly the four kinds of leaf, the last two contributing $\mathcal{A}$ and $\mathcal{E}(P)$. Every other rule implements its ND side conditions—eigenvariable checks and capture-avoiding substitution, refused rather than performed when renaming is unsafe—and returns the union of its premises' label sets while introducing no leaf of its own, so no leaf escapes O and no label survives in O without a leaf. Acceptance additionally requires the conclusion of θ to be α-equal to $[\![G]\!]$ and its hypotheses a subset of $[\![\Gamma]\!]$. □

The second half of the statement is what carries the weight. Derivability alone is weak by design: $\mathcal{E}(P)$ holds exactly the sequents the endorsing procedures asserted, so a tactic that endorses $[\![\Gamma]\!] \vdash [\![G]\!]$ outright yields a one-step derivation that says nothing about G. What the theorem rules out is that this happens silently; it surfaces as $O = \{\ell\}$ over a one-step derivation. The strength of an accepted proof is therefore not a property of Theorem 4.1 but a measured quantity, falling as O grows and as endorsed sequents move closer to the goal, and we claim no more than the ledger of §4.4 shows.

Corollary 4.1. *If $O = \varnothing$, then $[\![\Gamma]\!] \vdash_{\mathrm{ND}+\mathcal{A}} [\![G]\!]$: the kernel derivation uses only the rules of Table 2 and the registered axioms, and it carries no tactic, solver, or heuristic* endorsement*. This still trusts the CNL-to-FOL translation (Remark 4.1), the preprocessor, and the kernel's own α-equality and capture-avoiding substitution.*

Remark 4.1 (What remains inside the trusted base)**.** *Theorem 4.1 is stated over* translated *formulas, and the translation $[\![\cdot]\!]$ of Section 3 is itself trusted: a mistranslated CNL sentence yields a perfectly valid kernel derivation of the wrong proposition, and unlike an endorsement it leaves no label behind. Narrowing this remaining gap—by certifying the translation, or at least by making it auditable in the way solver calls now are—is the obvious next target, and we return to it in the discussion of limitations.*

4.4 ORACLE LEDGER

Because endorsement labels propagate through the derivation, each accepted theorem records the external procedures on which it depends, and by Theorem 4.1 that record is complete: the ledger enumerates the trust dependencies rather than summarising them. Table 3 reports them at the step level on a reference run of 100 problems.

Tier	Label form	Top labels (count)
Kernel-derived	(empty label set)	109 steps, 12.9% of all kernel-backed steps
Solver-endorsed	`external_solver:z3`, `sympy:*`, `compute:*`	`sympy:substitution` (96), `sympy:arith_eq` (77), `compute:float_eval` (26)
Tactic-asserted	`tactic:⟨name⟩`	`tactic:eq_transform` (48), `tactic:function_instantiation` (44), `tactic:mod_simp` (42)

Table 3: Trust tiers on 100 problems / 843 kernel-backed steps (step-level; a step may carry several labels). A solver-endorsed label trusts a mature external procedure on this instance only; a tactic-asserted one trusts that tactic's unaudited implementation. `contradiction_detect` is re-checked by z3 before endorsement.

4.5 WORKED KERNEL EXAMPLE

Let $x, y \in \mathbb{Z}$ satisfy $0 < y < x$ and $x + y + xy = 80$, with goal $x = 26$. An accepted CNL proof reads as follows; three of the five case branches are elided here and Appendix D gives the full text.

```
Let $x\in\mathbb{Z}$.
Let $y\in\mathbb{Z}$.
Assume $0 < y$.
Assume $y < x$.
Assume $x+y+xy=80$.
The goal is to prove $x=26$.
We have $1+x+y+xy=81$.
Then $(1+x)(1+y)=81$.
We have $y \in \mathbb{Z} \land 0 < y$.
Then $y \ge 1$.
Then $1+y \ge 2$.
We prove $x=26$ by working through different cases :
\begin{itemize}
\item Case $1+x=1$.
Then $x=0$. We have $1+y=81$. Then $y=80$.
We have $y < x$. Then $80 < 0$. Contradiction.
\informal{the cases $1+x=3,9,81$ are analogous and close by
    contradiction}
\item Case $1+x=27$.
Then $x=26$.
\end{itemize}
```

The block is certified by first establishing that the listed cases are exhaustive and then combining the certified branch conclusions through $\vee$E; four branches close by contradiction and $1 + x = 27$ yields $x = 26$ directly. The example also shows the split of Theorem 4.1: the logical structure is assembled from kernel rules, while arithmetic leaves such as $y \geq 1$ may rely on solver-backed endorsements, which are recorded and propagated into the final theorem.

5 VERIFIER-BASED POST-TRAINING AND INFERENCE-TIME COMPUTE

5.1 VERIFIER-BASED SFT

We construct SFT data through rejection sampling with a verifier-based agent. For each problem, the teacher model generates multiple candidate CNL proofs, which are combined with the problem statement and submitted to the verifier. Verification is performed in isolated subprocesses with fixed wall-clock limits so that failures or crashes do not affect other samples. Only verifier-accepted proofs are retained, deduplicated by their sanitized proof bodies, and capped at a fixed number per problem. Each retained proof is rendered as a chat example:

`system = STEP3_SYSTEM, user = statement, assistant = sanitized_completion.`

The student model is initialized from **QWEN3-4B-INSTRUCT-2507** and trained with full-parameter SFT. During training, it receives only the minimal theorem-statement prompt, while at inference time it is given the full CNL prompt used by the teacher, including formatting rules, syntax tables, requirement dictionaries, and worked examples.

The SFT model is trained exclusively on verifier-accepted proofs from `miniF2F`-valid. Table 4 reports the results on `miniF2F`. Verifier-based SFT improves verifier-accepted proof generation on both `miniF2F`-valid and the held-out `miniF2F`-test split. The gain is relatively larger on `miniF2F`-valid, from which the training problems are drawn, while the improvement on `miniF2F`-test indicates that the learned proof-generation behavior also transfers to held-out problems.

	miniF2F-valid		**miniF2F-test**	
Setup	pass@1	pass@4	pass@1	pass@4
Base	35.25	42.21	30.99	36.97
SFT	**48.87**	**63.11**	**37.82**	**48.32**

Table 4: Verifier-accepted performance on `miniF2F`. The SFT model is trained on 614 verifier-accepted samples from `miniF2F`-valid. Compared with the base model, SFT improves pass@4 by 20.90 percentage points on miniF2F-valid and by 11.35 percentage points on miniF2F-test.

5.2 VERIFIER-BASED RL

We further train the model with RL using verifier-based binary rewards. Unlike the SFT stage, which trains on verifier-accepted teacher samples, the RL stage optimizes the current policy on its own generated proof attempts, using the same verifier-based agent as the rollout environment.

For each problem x, the policy π_θ samples G responses. Failed proofs receive kernel feedback and are allowed one refinement attempt. The final proof attempt receives a binary acceptance reward together with a simple format reward.

We train the base **QWEN3-4B-INSTRUCT-2507** model with GRPO (Shao et al., 2024) on `miniF2F` (Zheng et al., 2022). The RL training set contains only problems from `miniF2F`-valid, while `miniF2F`-test remains unseen during training and is used only for evaluation. The verifier provides the task-level supervision: each sampled proof attempt receives a binary reward indicating whether the final proof is accepted. The one-step refinement mechanism exposes the policy to kernel feedback during rollout, while the optimization signal depends only on the final verification outcome and the format reward. The rollout structure is motivated by Search-R1 (Jin et al., 2025) and illustrated in Figure 1.

Iterative refinement with feedback. At inference time, we allow the model to perform iterative refinement based on kernel feedback. For each problem, the model first generates a CNL proof attempt which is submitted to the verifier. If the proof is accepted, the procedure terminates. Otherwise, the kernel diagnostic is appended to the context and used to generate a refined proof. This process repeats for several iterations, stopping early whenever a proof is accepted. This pipeline is visualized in Figure 2. We denote this setting by $\mathrm{refine} \times x@y$, where y is the number of independently sampled initial proof attempts and each attempt is allowed up to $x - 1$ rounds of refinement. This evaluates

whether additional test-time compute can improve proof success through verifier feedback rather than independent resampling. The results are shown in Table 5.

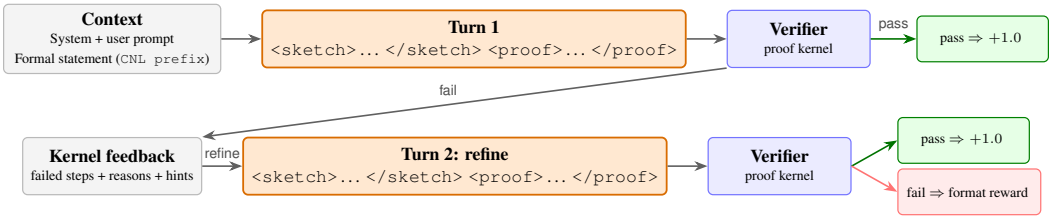

Figure 1: Verifier-guided rollout for RL. A failed first attempt receives structured verifier feedback and is allowed one refine attempt.

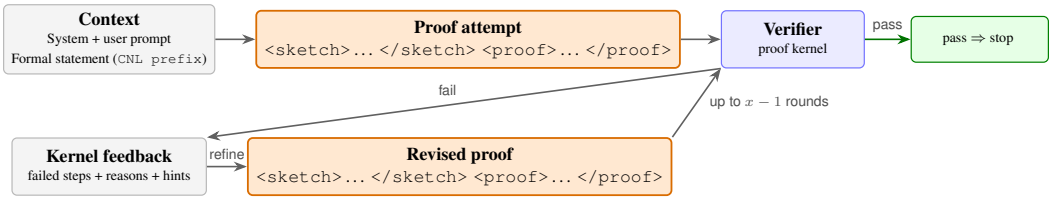

Figure 2: Iterative self-refinement with verifier feedback. After a proof attempt fails verification, the kernel feedback is appended to the context and used to generate a revised proof.

	miniF2F-valid				miniF2F-test			
Setup	pass@1	refine×5@4	refine×5@8	refine×10@8	pass@1	refine×5@4	refine×5@8	refine×10@8
Base	29.63	45.68	50.21	54.32	28.15	42.44	43.70	46.64
RL	**45.27**	**58.85**	**60.91**	**65.02**	**35.71**	**52.10**	**54.62**	**59.66**

Table 5: Verifier-accepted performance on `miniF2F`. The pass@1 results are different from Table 4 since the evaluation configurations are different.

6 Conclusion

We presented **CNLVerifier**, a lightweight verifier for mathematical proofs written in a controlled, LaTeX-style natural language. By combining a structured proof language with machine-readable verification errors, **CNLVerifier** provides a common interface for proof checking, post-training, and inference-time compute. Experiments on miniF2F show that verifier-based SFT and RL improve the generation of verifier-accepted CNL proofs, while iterative refinement further benefits from structured verifier feedback. These results suggest that CNL offers a practical middle ground between informal mathematical reasoning and fully formal proof languages, and that lightweight verification can serve as a useful training and interaction layer for LLM-based mathematical reasoning.

It is also important to acknowledge several limitations: (i) the verification framework is not yet fully certified: the translation of CNL statements into logical formulas is not formally certified, and there are still tactic modules that do not produce kernel derivations or solver certificates, so part of the reported acceptances are driver-accepted rather than theorem-backed and fall outside the scope of Theorem 4.1; (ii) the verifier has incomplete tactic coverage, so some valid proofs may be rejected simply because the required reasoning step is unsupported; and (iii) the current system still relies on manually constructed CNL prefixes and is primarily evaluated on competition-style problems, leaving broader mathematical domains for future work.

AI USE STATEMENT

In this work, we used generative AI tools for paper drafting and polishing, coding, literature review, proving theoretical claims, and generating candidate CNL proofs. LLM-generated code was verified and tested for correctness. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Ayush Agrawal, Siddhartha Gadgil, Navin Goyal, Ashvni Narayanan, and Anand Tadipatri. Towards a mathematics formalisation assistant using large language models, 2022. URL https://arxiv.org/abs/2211.07524.

Merlin Carl, Hinrich Lorenzen, and Michael Schmitz. Natural language proof checking in introduction to proof classes – first experiences with diproche. *Electronic Proceedings in Theoretical Computer Science*, 354:59–70, February 2022. ISSN 2075-2180. doi: 10.4204/eptcs.354.5. URL http://dx.doi.org/10.4204/EPTCS.354.5.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. URL https://arxiv.org/abs/2304.05128.

Marcos Cramer, Bernhard Fisseni, Peter Koepke, Daniel Kühlwein, Bernhard Schröder, and Jip Veldman. The naproche project: Controlled natural language proof checking of mathematical texts. In *Controlled Natural Language*, 2009. URL https://api.semanticscholar.org/CorpusID:8516552.

Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International conference on automated deduction*, pp. 378–388. Springer, 2015.

Maximilian Doré and Krysia Broda. The elfe system - verifying mathematical proofs of undergraduate students, 2018. URL https://arxiv.org/abs/1801.10513.

Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning*, 3, 01 2010.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, Wolfgang Macherey, Arnaud Doucet, Orhan Firat, and Nando de Freitas. Reinforced self-training (rest) for language modeling, 2023. URL https://arxiv.org/abs/2308.08998.

Petr Hajek and Pavel Pudlak. *Metamathematics of First-Order Arithmetic*. Springer Verlag, 1998.

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models, 2022. URL https://arxiv.org/abs/2102.06203.

John Harrison. Hol light: An overview. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pp. 60–66, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 9783642033582. doi: 10.1007/978-3-642-03359-9_4. URL https://doi.org/10.1007/978-3-642-03359-9_4.

Petra Hozzová, Laura Kovács, and Andrei Voronkov. Integer induction in saturation. In *CADE*, 2021. URL https://api.semanticscholar.org/CorpusID:235800783.

Muhammad Humayoun and Christophe Raffalli. Mathnat- mathematical text in a controlled natural language. 07 2010.

Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs, 2023. URL https://arxiv.org/abs/2210.12283.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan O Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training LLMs to reason and leverage search engines with reinforcement learning. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Rwhi91ideu.

Matt Kaufmann, Panagiotis Manolios, and Jstrother Moore. *Computer-Aided Reasoning: An Approach*, volume 3. 06 2000. ISBN 978-1-4613-7003-1. doi: 10.1007/978-1-4615-4449-4.

Richard Kaye. *Models of Peano Arithmetic*. Clarendon Press, 1991.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023. URL https://arxiv.org/abs/2305.20050.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia LI, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover: A frontier model for open-source automated theorem proving. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=x2y9i2HDjD.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=j4C0nALrgK.

Mingshan Liu, Shi Bo, and Jialing Fang. Enhancing mathematical reasoning in large language models with self-consistency-based hallucination detection. *ArXiv*, abs/2504.09440, 2025. URL https://api.semanticscholar.org/CorpusID:277781064.

Wenjie Ma, Andrei Cojocaru, Neel Kolhe, Haihan Zhang, Vincent Zhuang, Matei Zaharia, and Sewon Min. Reliable fine-grained evaluation of natural language math proofs. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=ky5iqwZSXI.

Patrick Massot. Teaching mathematics using lean and controlled natural language. In *International Conference on Interactive Theorem Proving*, 2024. URL https://api.semanticscholar.org/CorpusID:272330159.

Shashank Pathak. Gflean: An autoformalisation framework for lean via gf, 2024. URL https://arxiv.org/abs/2404.01234.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020. URL https://arxiv.org/abs/2009.03393.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Jun-Mei Song, Mingchuan Zhang, Y. K. Li, Yu Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *ArXiv*, abs/2402.03300, 2024. URL https://api.semanticscholar.org/CorpusID:267412607.

Andreas Stephan, Dawei Zhu, Matthias Aßenmacher, Xiaoyu Shen, and Benjamin Roth. From calculation to adjudication: Examining LLM judges on mathematical reasoning tasks. In Ofir Arviv, Miruna Clinciu, Kaustubh Dhole, Rotem Dror, Sebastian Gehrmann, Eliya Habba, Itay Itzhak, Simon Mille, Yotam Perlitz, Enrico Santus, João Sedoc, Michal Shmueli Scheuer, Gabriel Stanovsky, and Oyvind Tafjord (eds.), *Proceedings of the Fourth Workshop on Generation, Evaluation and Metrics (GEM²)*, pp. 759–773, Vienna, Austria and virtual meeting, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-261-9. URL https://aclanthology.org/2025.gem-1.65/.

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, 2025a. URL https://arxiv.org/abs/2504.11354.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023. URL https://arxiv.org/abs/2203.11171.

Yiping Wang, Qing Yang, Zhiyuan Zeng, Liliang Ren, Liyuan Liu, Baolin Peng, Hao Cheng, Xuehai He, Kuan Wang, Jianfeng Gao, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and yelong shen. Reinforcement learning for reasoning in large language models with one training example. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025b. URL https://openreview.net/forum?id=IBrRNLr6JA.

Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow. The isabelle framework. In *International Conference on Theorem Proving in Higher Order Logics*, pp. 33–38. Springer, 2008.

Markus Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry (eds.), *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, Lecture Notes in Computer Science, pp. 167–184. Springer, 1999. doi: 10.1007/3-540-48256-3_12. URL https://doi.org/10.1007/3-540-48256-3_12.

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models, 2022. URL https://arxiv.org/abs/2205.12615.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data, 2024a. URL https://arxiv.org/abs/2405.14333.

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search, 2024b. URL https://arxiv.org/abs/2408.08152.

Bowen Yang, Yi Yuan, Chenyi Li, Ziyu Wang, Liangqi Li, Bo Zhang, Zhe Li, and Zaiwen Wen. Construction-verification: A benchmark for applied mathematics in lean 4, 2026. URL https://arxiv.org/abs/2602.01291.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models, 2023. URL https://arxiv.org/abs/2306.15626.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Zheng Yuan, Dahua Lin, and Kai Chen. Lean workbook: A large-scale lean problem set formalized from natural language math problems, 2025. URL https://arxiv.org/abs/2406.03847.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling relationship on learning mathematical reasoning with large language models, 2023. URL https://arxiv.org/abs/2308.01825.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning, 2022. URL https://arxiv.org/abs/2203.14465.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. The lessons of developing process reward models in mathematical reasoning. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 10495–10516, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.547. URL https://aclanthology.org/2025.findings-acl.547/.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=uccHPGDlao.

A AST TO TERM

A.1 ATOMIC FORMULAS AND RELATIONS

Definition A.1 (Relation mapping)**.** *Let s, t denote terms.*

CNL syntax	*FOL / internal predicate*
Order and equality	
`s = t`	$s = t$ *(arguments of sort* `SET` *become* `eq_set`*)*
`s \neq t`	$s \neq t$
`s < t`, `s > t`	$s < t$, $s > t$
`s \leq t`, `s \le t`	$s \leq t$
`s \geq t`, `s \ge t`	$s \geq t$
Set membership and inclusion	
`s \in T`	$s \in T$
`s \notin T`	$s \notin T$
`s \subseteq T`, `s \subset T`	$s \subseteq T$*; both surface forms parse to* `subset` *(see remark)*
`s \supseteq T`, `s \supset T`	$s \supseteq T$*; both parse to* `superset`*; RHS is a* `domain` *nonterminal*
Number theory	
`s \mid t`	$s \mid t$
`s \nmid t`	$s \nmid t$
`s \equiv t \pmod n`	$s \equiv t \pmod{n}$ *as* $\texttt{cong}(s, t, n)$
`t \text{is prime}`	$\mathsf{is_prime}(t)$ *(predicate* `is_prime`*)*

Remark (`\pmod` in CNL). *In CNL input,* `\pmod` *is* not *used by itself: it appears only as part of the congruence surface form $s \equiv t \pmod{n}$, built from* `\equiv` *together with* `\pmod`*, and maps to* $\texttt{cong}(s, t, n)$ *(table above). Integer remainder as a* term *is written with* `\bmod` *(Definition A.2), which maps to* `mod`*. The parser also accepts* `expr \pmod{...}` *as a binary* `mod` *term for alignment with LaTeX; typical CNL statements use* `\bmod` *for remainder and* `\equiv ... \pmod` *for congruence. When* `ast_to_term` *processes* `\equiv` *without an explicit* `\pmod` *child, it may still recover a modulus from a* `mod` *subterm inside the left/right expressions.*

Chained relations. *CNL admits mixed chains of equalities and inequalities. Examples (all under* `prop_atom` *in* `parser/gramma.py`*):*

$[\![a < b \leq c]\!] = (a < b) \wedge (b \leq c)$ *(rule* `chained_lt_le`*)*
$[\![a \leq b < c]\!] = (a \leq b) \wedge (b < c)$ *(*`chained_le_lt`*)*
$[\![a = b = c]\!] = (a = b) \wedge (b = c)$ *(*`chained_eq_eq`*; longer chains use* `chained_eq_eq_4`*)*
$[\![a = b \geq c]\!] = (a = b) \wedge (b \geq c)$ *(*`chained_eq_ge`*)*

Chained congruence. *Nested* `a \equiv b \equiv c \pmod{m}` *is split into* `and(`$\texttt{cong}(a, b, m)$, $\texttt{cong}(b, c, m)$*) when the parser emits a nested* `cong` *AST.*

A.2 Terms and Expressions

Definition A.2 (Term mapping)**.**

CNL expression	*FOL term*
Variable x	x *(sort from* Γ*)*
Integer/rational literal n	*constant* n
$s + t$	$+(s, t)$
$s - t$	$-(s, t)$
$s \cdot t$ *(or juxtaposition)*	$\cdot(s, t)$
`s / t`	$/(s, t)$
`s^n`	$\mathsf{pow}(s, n)$
`s \bmod n`	$\mathsf{mod}(s, n)$ *(binary remainder* term*;* `FuncSymbol.name mod`*)*
`\sqrt{s}`	$\mathsf{sqrt}(s)$
`\gcd(s,t)`	$\mathsf{gcd}(s, t)$
`\lcm(s,t)`	$\mathsf{lcm}(s, t)$
`\lfloor s \rfloor`	$\lfloor s \rfloor$
`\lceil s \rceil`	$\lceil s \rceil$
`f(t_1,...,t_n)`	$f(t_1, \ldots, t_n)$

A.3 Set Comprehension

Definition A.3 (Set comprehension)**.**

$$[\![\{x \in D \mid \varphi\}]\!]_\Gamma \;=\; \lambda x{:}[\![D]\!].\, x \in [\![D]\!] \wedge [\![\varphi]\!]_{\Gamma, x:[\![D]\!]}$$

This is realised as the characteristic predicate of the set. In FOL, membership in the comprehension is characterised by

$$t \in \{x \in D \mid \varphi\} \iff t \in D \wedge \varphi[x \mapsto t].$$

The system encodes a definition `eq(S, {x ∈ D | φ})` *as*

```
Forall([x], Iff(member_of(x,S), Conjunction(member_of(x,D), φ)))
```

where `D` *is the domain constant, derived from the type node in the comprehension, and then uses* $\forall$*-elimination for membership queries.*

A.4 Compositionality of the Translation

The mapping $[\![\cdot]\!]_\Gamma$ is *compositional* in the sense appropriate to a context-indexed semantics: for each AST node, the translated term is determined by the node's grammar rule, the recursive translation of its children, and the current environment Γ (*binders* extend Γ in the usual way). Some branches are *pattern* clauses over a whole subtree (e.g. set definitions as in Definition A.3), rather than a single generic rule that first translates each child in isolation and then applies a fixed combinator; the specification is nevertheless structural recursion on the Lark parse tree.

B TACTIC LIBRARY DETAILS

This appendix documents the tactic library used by the ProofKernel, including its taxonomy, scheduler triggers, input/output goal shapes, common failure modes, and representative examples. The purpose is to make the verifier's trusted boundary explicit and the step-level behavior reproducible.

B.1 OVERVIEW AND TRUST BOUNDARY

At a high level, the 57 tactic modules are grouped for exposition into: (1) algebra/arithmetic tactics, (2) logical-structure tactics, (3) number-theory tactics, (4) set-theory tactics, and (5) meta-composition tactics (e.g., `auto_complete`). The registry itself uses eight domain categories (arithmetic, logic, set theory, number theory, inequality, quantifier, transformation, meta) plus combinators; a module may expose more than one named tactic, which is why "57 modules" and "$\sim$65 tactics" differ. Tactics live *outside* the trusted kernel: they search for a derivation or request an endorsement, but they cannot construct a `Thm`. The kernel is responsible for proof-tree correctness (case completeness, branch closure, context discipline) and for the twenty-two inference rules of Table 2. Thus, an accepted proof backed by a final kernel theorem is sound relative to: (i) the CNL-to-FOL translation, (ii) the kernel rules and registered axioms, and (iii) the labeled endorsements used in its derivation.

B.2 TACTIC CATALOG

Tactic	Category	Typical goal shape
`linarith`	algebra/arithmetic	linear (in)equality over $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$
`nlinarith`	algebra/arithmetic	polynomial (in)equality (low degree)
`ring`	algebra/arithmetic	polynomial identity/equality
`field_simp`	algebra/arithmetic	rational expression equality
`forall_instantiate`	logic	$\forall x, \phi(x) \Rightarrow \phi(t)$
`existence`	logic	$\exists x, \phi(x)$
`disjunction_elimination`	logic	goals under case split
`divisibility`	number theory	$a \mid b$, $a \nmid b$
`mod_simp`	number theory	congruence / mod terms
`prime_properties`	number theory	primality lemmas
`set_theory`	set theory	$\in, \subseteq, \cup, \cap, \setminus$ relations
`set_cardinality`	set theory	finite set cardinal constraints
`auto_complete`	meta	mixed local goals

Table 6: Representative tactics dispatched by the scheduler, using the names in `tactic/tactic_scheduler.py`.

B.3 SCHEDULER POLICY

The scheduler follows a deterministic priority policy:

1. **Direct assumption/identity check** (goal already in context).
2. **Goal decomposition** (e.g., conjunction splitting).
3. **Specialized tactics first** (by detected syntactic/typed pattern).
4. **Meta-composition fallback** (`auto_complete`).
5. **Failure return** with phase-localized error code.

In implementation terms, tactic selection is syntax- and type-driven: a candidate tactic is attempted only when its precondition pattern matches the goal and context. This keeps error attribution stable and reproducible.

B.4 FAILURE MODES AND ERROR-CODE MAPPING

Failure mode	Typical error code	Suggested repair hint
No tactic matches goal pattern	`E_KERNEL_GOAL_UNRESOLVED`	add intermediate lemma / rewrite goal
Premises insufficient for closure	`E_KERNEL_GOAL_UNRESOLVED`	derive missing fact via `We have` step
Type mismatch in tactic inputs	`E_TYPE_FUNC_ARG_MISMATCH`	fix domain/sort declarations
Undeclared variable	`E_TYPE_VAR_UNDECLARED`	add a `Let` declaration
Non-exhaustive case split	`E_KERNEL_CASE_INCOMPLETE`	add missing case branch
Goal used as a case header	`E_KERNEL_CIRCULAR_CASE_GOAL`	split on a genuine case condition
Illegal `Let`-rebinding	`E_KERNEL_LET_REBINDING`	do not reassign a preamble variable
Bare restatement of the goal	`E_KERNEL_BARE_GOAL_RESTATEMENT`	derive the goal from premises
Unparseable line	`E_PARSE_NO_AST` / `E_PARSE_NO_OPENER`	start the line with an allowed opener

Table 7: Common tactic/kernel interaction failures and repair directions, using the project's `VerifierError` codes.

B.5 REPRESENTATIVE STEP-LEVEL EXAMPLES

Example A (algebraic closure with `linarith`). **Context:** $y \in \mathbb{Z}, 0 < y, y < x, x+y+xy = 80$
Goal: $1 + y \geq 2$
Kernel action: forwards $y \in \mathbb{Z} \wedge 0 < y$ into tactic state; dispatches `linarith`.
Outcome: goal discharged. (Without $y \in \mathbb{Z}$, $0 < y$ over $\mathbb{R}$ does not imply $y \geq 1$.) `We have` $y \in \mathbb{Z} \wedge 0 < y$`.` then `Then` $y \geq 1$`.` then `Then` $1 + y \geq 2$`.`

Example B (ring normalization with `ring`). **Goal:** $(1 + x)(1 + y) = 1 + x + y + xy$
Kernel action: sends polynomial identity to `ring`.
Outcome: normalization equality proved.

Example C (case branch closure). **Branch assumption:** $1 + x = 1$
Local derivation: $x = 0,\ y = 80,\ y < x \Rightarrow 80 < 0$
Kernel/tactic interaction: arithmetic contradiction discharged; branch marked complete.

C Natural deduction notation

This appendix records the natural-deduction reading of the trusted kernel in Table 2. The rules below are implemented as side-effect-free functions on the abstract type `Thm`; they are not merely documentation of tactic traces. For theorem-backed acceptance the gateway holds a `Thm` whose conclusion is α-equal to the goal and whose hypotheses are a subset of the current context; Theorem 4.1 then accounts for every leaf of that `Thm`.

C.1 Sequents

A sequent $\Gamma \vdash \varphi$ means that φ is derivable from the finite set of hypotheses Γ. Kernel rules construct a new `Thm` from zero or more premise `Thm`s after a shape check. The two extension points `new_axiom` and `oracle` are *not* ND rules; they populate $\mathcal{A}$ and $\mathcal{E}(P)$ in Theorem 4.1.

C.2 Core inference rules

Code names match Table 2.

C.2.1 Structural

$$\frac{}{\varphi \vdash \varphi}\ \textsf{assume} \qquad \frac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi}\ \textsf{weaken}$$

C.2.2 Conjunction

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ \textsf{conj_intro} \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi \text{ or } \Gamma \vdash \psi}\ \textsf{conj_elim}$$

C.2.3 Disjunction

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\ \textsf{disj_intro_left} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}\ \textsf{disj_intro_right}$$

$$\frac{\Gamma \vdash \varphi \vee \psi \quad \Gamma, \varphi \vdash \chi \quad \Gamma, \psi \vdash \chi}{\Gamma \vdash \chi}\ \textsf{disj_elim}$$

C.2.4 Implication

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ \textsf{impl_intro} \qquad \frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ \textsf{impl_elim}$$

C.2.5 Biconditional

$$\frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \psi \to \varphi}{\Gamma \vdash \varphi \leftrightarrow \psi}\ \textsf{iff_intro}$$

$$\frac{\Gamma \vdash \varphi \leftrightarrow \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ \textsf{iff_elim_left} \qquad \frac{\Gamma \vdash \varphi \leftrightarrow \psi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi}\ \textsf{iff_elim_right}$$

C.2.6 Negation and falsity

$$\frac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ \textsf{neg_intro} \qquad \frac{\Gamma \vdash \neg\varphi \quad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\ \textsf{neg_elim}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ \textsf{falsum_elim} \qquad \frac{\Gamma, \neg\varphi \vdash \bot}{\Gamma \vdash \varphi}\ \textsf{by_contradiction}$$

C.2.7 Universal quantification

$$\frac{\Gamma \vdash \varphi[x \mapsto y]}{\Gamma \vdash \forall x.\, \varphi}\ \textsf{forall_intro}\ (y \text{ fresh in } \Gamma) \qquad \frac{\Gamma \vdash \forall x.\, \varphi}{\Gamma \vdash \varphi[x \mapsto t]}\ \textsf{forall_elim}$$

C.2.8 EXISTENTIAL QUANTIFICATION

$$\frac{\Gamma \vdash \varphi[x \mapsto t]}{\Gamma \vdash \exists x.\, \varphi}\ \textsf{exists_intro} \qquad \frac{\Gamma \vdash \exists x.\, \varphi \quad \Gamma, \varphi[x \mapsto y] \vdash \chi}{\Gamma \vdash \chi}\ \textsf{exists_elim}\ (y \text{ fresh in } \Gamma, \chi)$$

There is no kernel rule for $\exists!$. As in Definition 3.2, uniqueness is sugar for $\exists \wedge \forall$; uniqueness reasoning is tactic- or oracle-side.

C.2.9 EQUALITY

$$\frac{}{\Gamma \vdash t = t}\ \textsf{refl} \qquad \frac{\Gamma \vdash s = t \quad \Gamma \vdash \varphi[x \mapsto s]}{\Gamma \vdash \varphi[x \mapsto t]}\ \textsf{eq_subst}$$

Symmetry and transitivity are derived rules in the gateway, assembled from `refl` and `eq_subst`.

C.3 INDUCTION IS A SCHEMA INSTANCE, NOT A KERNEL RULE

Induction is not among the twenty-two rules. A completed induction block is assembled by the gateway: the base and step branches become

$$\forall n.\, (n = b \to \varphi(n)) \quad \text{and} \quad \forall n.\, (n > b \to (\varphi(n-1) \to \varphi(n))),$$

a per-block instance of the schema

$$\big(\forall n.\, n = b \to \varphi(n)\big) \wedge \big(\forall n.\, n > b \to (\varphi(n-1) \to \varphi(n))\big) \to \forall n.\, (n \geq b \to \varphi(n))$$

is registered with `new_axiom`, and the block is closed by modus ponens against a completeness theorem $n = b \vee n > b$. The instance is an element of $\mathcal{A}$ in Theorem 4.1.

C.4 TACTIC COMBINATORS

The `TacticScheduler` parses tactic scripts into a small combinator language. Combinators search; they do not construct `Thm`s.

Definition C.1 (Tactic combinators)**.**

FIRST : *try alternatives separated by* `|` *until one succeeds.*
t_1 `;` t_2 : *run* t_1 *then continue on the resulting goal(s) with* t_2.
DONE : *succeed only if there is no remaining goal; otherwise fail.*
`fail "msg"` : *signal failure with a user-visible message.*

A composed script can produce a theorem only if some atomic tactic it invokes produces one, either by a genuine derivation or by a labelled endorsement. Combinators add no trust beyond Table 2.

C.5 CASE ANALYSIS AND INDUCTION AS PROOF TREES

Definition C.2 (Proof tree)**.** *A* proof tree *is a tree of* `ProofNode` *objects. Each node holds a* `ProofState` (Γ, G)*, pointers to its parent and children, and Boolean flags including* `is_directly_solved`*,* `is_branch_complete`*, and* `is_proved_by_sorry`*. Those flags are a reporting device. For theorem-backed acceptance, a node additionally requires a gateway theorem whose conclusion is* α*-equal to* G *and whose hypotheses are a subset of* Γ.

Case analysis is kernel $\vee$E (`disj_elim`):

$$\frac{\Gamma \vdash A \vee B \qquad (\Gamma, A \vdash G) \qquad (\Gamma, B \vdash G)}{\Gamma \vdash G}\ \textsf{disj_elim}$$

Mathematical induction is the schema instance above, not a kernel inference rule.

D WORKED EXAMPLE IN FULL

```
Let $x\in\mathbb{Z}$.
Let $y\in\mathbb{Z}$.
Assume $0 < y$.
Assume $y < x$.
Assume $x+y+xy=80$.
The goal is to prove $x=26$.
We have $1+x+y+xy=81$.
Then $(1+x)(1+y)=81$.
We have $y \in \mathbb{Z} \land 0 < y$.
Then $y \ge 1$.
Then $1+y \ge 2$.
We prove $x=26$ by working through different cases :
\begin{itemize}
\item Case $1+x=1$.
Then $x=0$. We have $1+y=81$. Then $y=80$.
We have $y < x$. Then $80 < 0$. Contradiction.
\item Case $1+x=3$.
Then $x=2$. We have $1+y=27$. Then $y=26$.
We have $y < x$. Then $26 < 2$. Contradiction.
\item Case $1+x=9$.
Then $x=8$. We have $1+y=9$. Then $y=8$.
We have $y < x$. Then $8 < 8$. Contradiction.
\item Case $1+x=27$.
Then $x=26$.
\item Case $1+x=81$.
Then $x=80$. We have $1+y=1$. Then $y=0$.
We have $0 < y$. Then $0 < 0$. Contradiction.
\end{itemize}
```