MATHTYPES: TOWARDS DETERMINISTIC STEP-LEVEL CHECKING FOR REASONING IN MATH EXAMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Mathematical reasoning with Large Language Models (LLMs) mostly remains final-answer verifiable, with existing step-level checks being either probabilistic (LLM judges, process reward models), unstructured (pairing a model with a CAS), or too heavyweight for exam-style reasoning (proof assistants). We present MathTypes, a typed language in which reasoning steps are calls into a registry of SymPy-backed operations carrying a claimed result that is evaluated deterministically, and a compile–verify–correct evaluator loop assay built on it. Our method makes models select from a fixed type registry instead of authoring verification in free-text, with unverifiable steps left to escape hatches, thereby providing step-level feedback on natural-language mathematical reasoning akin to interpreted programming languages, while ensuring reasoning chains are more readable than proof assistants. On difficulty-filtered problems from six competitive exams, MathTypes is the most accurate method for all seven models tested, beating the strongest of six baselines by 9.1–17.6 points and extending the accuracy-cost Pareto frontier. Detailed ablations show that mostly the verifier's step-level diagnostics, not registry knowledge alone, drive the gain; that most of it lands within two repair passes; and that the required type registry can be induced cheaply. Through MathTypes, we establish the promise of readable math reasoning whose typed steps are machine-checked. Our code is made public: `https://anonymous.4open.science/r/mathtypes-A65C`

1 INTRODUCTION

LLMs now approach saturation on many mathematics benchmarks (Cobbe et al., 2021; Lightman et al., 2024; Pyeon et al., 2025) and have reached gold-medal-level performance at the International Mathematical Olympiad (Huang & Yang, 2025). However, models reach correct answers through flawed derivations (Petrov et al., 2025), write chains of thought that do not reflect what actually drove their predictions (Turpin et al., 2023), and degrade when superficial details of a problem change (Mirzadeh et al., 2025). Verifying the process besides the outcome is therefore central both to measuring mathematical reasoning and to improving it. Existing work has approached this problem with a few standard methods: handing computation to an interpreter or a computer algebra system (Gao et al., 2023; Chen et al., 2023; Bagheri Nezhad et al., 2026), and using a language model to check each step (Lightman et al., 2024; Wang et al., 2024; Huang & Yang, 2025; Feng et al., 2026). However, Python programs check isolated steps with nothing about the reasoning connecting them, and LLM-based verifiers are probabilistic, expensive, and open to reward hacking (Cheng et al., 2025). In our reproduction, we noted that frontier models such as Gemini 3.1 Pro as a PRM returned the same verdict on all five repeats for only 82.7% of solutions, with its acceptance of a solution contingent on whether the solution agreed with the answer Gemini would have given itself. Proof assistants are, therefore, increasingly being used to solve these issues as languages such as Lean (de Moura et al., 2015) and Coq (Coquand & Huet, 1988) are deterministically sound. However, they require decomposing every step into atomic inferences (Mahinpei et al., 2025; Rossel et al., 2026), a formalization cost that school- and university-level mathematics rarely needs (Tran Minh et al., 2025; Karsten et al., 2023).

Motivated by ideas in type systems and these shortcomings in existing methods, we introduce **MathTypes**. In a strongly typed language, a program is built from declared operations whose inputs and outputs are checked. MathTypes applies the same discipline to mathematical reasoning.

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.

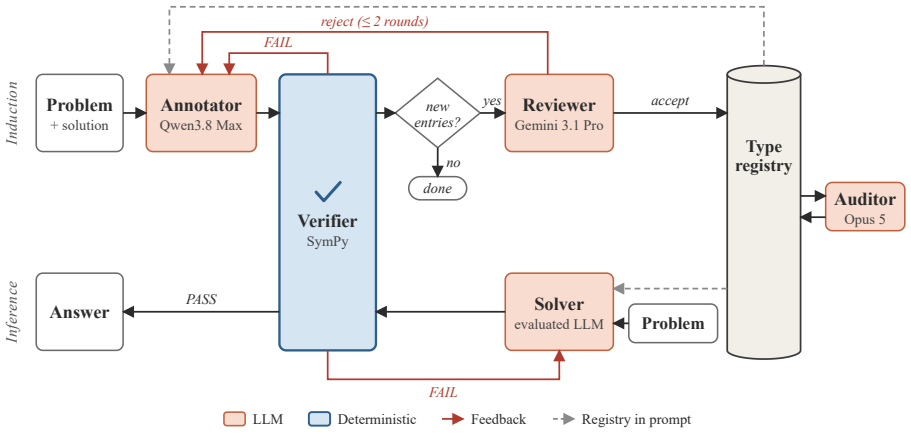

Figure 1: An overview of the type generation and evaluation pipeline used in MathTypes. In an automated system, an annotator model translates expert human-written solutions to MathTypes `.mt` code, which a problem-level and registry-level auditor LLM reviews to ensure mathematical validity, usability, and avoid duplicates. At inference time, the type registry is input to the solver, which writes a `.mt` program to be verified. All LLM choices are based on experimental results (Section 4.1).

A solution is a program in which each step is a call to a registered type, interleaved with free-form natural-language blocks. The model ends each call with a claimed result that a deterministic SymPy-backed verifier recomputes. The verifier's step-level verdicts, which never see the reference answer, go back to the model as compiler diagnostics. Solving a mathematics problem becomes a compile–verify–repair loop, much like programming (Figure 1). A type system is only as useful as its vocabulary, and writing out all of mathematics by hand is infeasible. Creating Lean libraries, for instance, has taken decades of community effort. We instead induce exam-specific registries automatically with a three-stage process from the published solutions of past papers, starting from a small seed to show that our method can scale like PRMs. Specifically, our contributions are:

1. **MathTypes**, a typed language for natural-language mathematical reasoning with a five-stage compiler and a deterministic verifier (Section 3.1).
2. An automated registry induction pipeline that grows type and theorem vocabularies from worked solutions to automatically scale MathTypes (Sections 3.2 and 4.1).
3. A difficulty-filtered evaluation set from six selective mathematics examinations to better elucidate capability gains with modern frontier LLMs (Section 3.2).
4. Analyses motivating our design choices, and reliability of deterministic vs. LLM-based process verification (Section 4.3).

Evaluation on the hard set of 171 problems spanning high-school to doctoral level admissions shows that MathTypes leads across all seven tested models in accuracy, beating the best prompting and program-based baselines by 9.1 to 17.6 points in question-solving accuracy, while machine-checking the steps expressible in its registry. Our method also extends the accuracy–completion-token Pareto frontier as the correction loop is cheap (68% of the gain comes from the first correction, and about 90% by the second). Controls show that giving CoT the registry as plain text does not reproduce the gain, and that replacing the verifier's diagnostics with a generic error message loses part of it. Unlike a PRM, our verifier returns the same verdict every time a program is re-verified. Further, we show that a registry induced for just $1.29 already reaches saturated accuracy and coverage on JEE.

2 Related Work

2.1 Process Verification for LLM Reasoning

Process reasoning was introduced by Chain-of-Thought (Wei et al., 2022), which has since been standardized across multiple domains, including mathematics. Programmatic verification of such resulting processes has also been studied in great depth, with some of the earliest work including

Program-aided Language Models (Gao et al., 2023), which proposed generating Python programs with the language's interpreter acting as the *solution* step. Program of Thoughts (PoT) further improved upon this idea (Chen et al., 2023), which interleaved text and Python code in the model's generation. Recent approaches such as SymCode have improved upon this by using specialized Python libraries such as SymPy (Bagheri Nezhad et al., 2026). However, programmatic verification of mathematical reasoning has been shown to have little impact on LLMs' representational capabilities (Nath et al., 2026; Asperti et al., 2026). Other lines of work include plan generation before solving to improve readability and performance (Wang et al., 2023); grounding LLM reasoning in pedagogical guidelines (Jiang et al., 2024); or explicit self-check mechanisms (Miao et al., 2024).

Process verification has also been attempted with LLMs themselves. Lightman et al. (2024) introduced the first human-labeled dataset for training probabilistic Process Reward Models (PRMs). Utilizing this, Wang et al. (2024) introduced step-level scoring. LLM evaluators have been heavily used in recent work on solving international mathematics and physics olympiad questions (Huang & Yang, 2025; Yu et al., 2026), especially as part of multi-model workflows. Aletheia (Feng et al., 2026), for instance, includes Gemini 3.1 Pro Deep Think in three roles: a solution generator, verifier, and reviser. However, PRMs fail to offer deterministic feedback, are expensive to run, and can be reward-hacked (Cheng et al., 2025). Motivated by the representational brittleness of programming language-based methods and the probabilistic nature of PRMs, we introduce a typed language designed for natural language LLM mathematical reasoning, MathTypes.

2.2 Mathematical Verification

Modern computational mathematics verification systems trace their origin to Type theory (Russell, 1908), where a type is defined as a specification of what an expression is allowed to be. The promise of computational interpretation of mathematics was established as early as 1940 (Church, 1940). The Curry-Howard correspondence (Curry, 1934; Howard, 1980) connected proofs and programs, laying the groundwork for automatic verification of mathematical processes. Further, the Martin-Löf dependent type theory (Martin-Löf, 1975) led to the development of proof assistants such as Coq (Coquand & Huet, 1988), Lean (de Moura et al., 2015), Agda (Norell, 2007), and Idris (Brady, 2011). Previous work has utilized proof assistants to improve LLM performance in mathematical tasks (Yang et al., 2023; Tsoukalas et al., 2026; Trinh et al., 2024). However, these systems act as standalone programming languages, requiring atomic decomposition to verify each mathematical operation, which makes them verbose, syntax-heavy, and less readable than pen-and-paper proofs (Mahinpei et al., 2025; Rossel et al., 2026). Further, HERMES (Ospanov et al., 2026) interleaves informal reasoning with Lean-verified steps and feeds verification signals back to the model. MathTypes keeps this loop but replaces per-step Lean autoformalization and proving with calls into a fixed, automatically induced type registry. Moreover, proof assistants see limited application to other domains of mathematics, such as high school and college-level material, where over-formalization is often unnecessary (Mahinpei et al., 2025; Tran Minh et al., 2025; Karsten et al., 2023). Therefore, we study useful deterministic verification while retaining a natural-language reasoning interface.

2.3 Mathematical Evaluation Benchmarks and Exam-Based Reasoning

Mathematics benchmarks are well-represented among evaluation corpora (Cobbe et al., 2021; Lightman et al., 2024). Recent advances in LLMs have ensured they achieve near-saturation accuracy on most mathematics exams without the need for sufficient formalization (Pyeon et al., 2025; Huang & Yang, 2025). Yet, the latest benchmarks on some of the most selective exams in the world, such as the Iranian Konkoor (Dabiriaghdam et al., 2026), the Brazilian ITA/IME (Godoy, 2025) and the Indian Joint Entrance Examination (JEE) Advanced (Mukherjee & Ghosh, 2025), show the scope for improvement. We specifically choose exams with verifiable outputs and non-saturated difficulty.

3 Methods

3.1 MathTypes

We design MathTypes as a typed language for mathematical reasoning with a fully-formed five-stage compiler (Appendix A) and elements specifically suited such that LLMs can write it fluently

without major training. A deterministic compiler reduces the model-written code to an intermediate representation (IR) checkable through computer algebra systems. Its design target is the middle ground between proof assistants and process reward models, as it's heavier than pairing a model with a calculator (which verifies isolated computations but nothing about the reasoning connecting them), while being lighter than Lean or Coq (which focuses on atomic reduction and pays a formalization cost that school- and university-level mathematics does not need) and process reward models (which itself is a language model inheriting the unreliability we intend to check).

Programs: A MathTypes program is a `.mt` source file, which interleaves a sequence of typed calls implemented as functions with natural-language reasoning delimited by `<thought>` tags. We reuse HTML-style tags and function-call syntax, which are heavily represented in LLM training data, so no new notation has to be taught. Unlike native tool calling, a call does not ask a tool to compute: it asserts a result for the verifier to check, and a whole solution is checked as one program rather than one round trip per step. To write `.mt` code, every call must carry a valid `type_id` in the final registry, bind declared inputs, and pass a `claimed_result` which models the LLM's expected outcome that our verifier recomputes independently and compares. Theorems take a name, constraints, conditions, and claimed conclusion as input. While compiling, all `<thought>` traces are discarded as code comments. Here's an example:

```
<thought>Solve the first equation for x.</thought>
solve_linear(equation: "4*x - 20", solve_for: "x", claimed_result: "5")
<thought>I need to verify if the triangle is right-angled.</thought>
apply_theorem(theorem: "pythagorean_theorem",
  conditions: "{'leg1': '3', 'leg2': '4', 'hypotenuse': '5'}",
  conclusion_claimed: "right triangle", claimed_result: "right triangle")
```

Internally, a fixed table of SymPy-backed Python functions (Appendix I) executes an input, shifting the load of verification from the model: a solver cannot write a permissive check to bypass the verifier. Theorem bodies are the exception, as they are proposed during induction and admitted by review (Section 3.2); they are the main source of the verifier's false acceptances (Appendix H). For every step, MathTypes gives one of five verdicts: `PASS`, `FAIL`, `UNVERIFIED` (an escape hatch step that is not deterministically verifiable), `ERROR` (parse failures or timeouts), or `UNRESOLVED` (for a type/theorem proposed in the same program during grammar building). `UNRESOLVED` cases are reviewed before addition to the registry. We define two metrics to measure model behavior from this design: Coverage and Step Accuracy. For a MathTypes program with n steps:

$$\text{coverage} = \frac{n - n_{\text{UNVERIFIED}} - n_{\text{UNRESOLVED}}}{n}, \qquad \text{step accuracy} = \frac{n_{\text{PASS}}}{n \cdot \text{coverage}}$$

Further, to judge whether proposed theorems contain trivial assertions designed to inflate coverage and game the verifier, we measure the share of theorem verification bodies that read the step's claim (**"Binds claim"**) and the share whose verdict cannot depend on the claim at all (**"claim-blind"**). Both tests are syntactic, so the claim-blind rate is a lower bound (see Appendix F for the definitions).

Escape hatches: Not all mathematics in this domain is formally verifiable, and we include two escape hatches in MathTypes to model these steps. `translate_to_formal` covers the narrower case of rendering a natural-language problem statement into formal notation. This is important for math word problems, where linguistic interpretation follows any verifiable steps. `informal_step` is a general escape hatch void of verification for claims uncaptured by defined types.

An overuse threshold of 50% is used to prevent misuse based on empirical results (Appendix J), which voids a `.mt` solution if the model also proposed no new types or theorems. For further quantification, we propose a failure analysis metric, unfamiliar problem rate, which is defined as the share of a model's usable attempts on an exam that yield no machine-gradable final claim. High rates identify unfamiliar syllabus (types and theorems not in the registry) and model behavior, minimizing both of which are our optimization targets. For N gradable `.mt` programs, and $g(a) = 1$ if attempt a produced an extractable final claim, unfamiliar problem rate ($\mathcal{U}$) is defined as:

$$\mathcal{U} = 1 - \frac{1}{N} \sum_{a=1}^{N} g(a)$$

The evaluator loop assay: An evaluator loop assay is applied both for growing the grammar and for evaluations. A solver model receives the problem image and a registry-conditioned system prompt, and emits a .mt program. The program is compiled and verified. On any compile error

or verification failure, the compiler's diagnostic is appended to the conversation, and the model is asked to repair. Because MathTypes can provide step-level feedback, this essentially makes natural-language mathematical reasoning like programming. The compiler's diagnostic is fully blind to the ground-truth answer. Grading against official exam keys is done externally for all reported results.

3.2 GROWING THE GRAMMAR

Chosen exams: To test our method, we select six challenging mathematics exams spanning high-school to doctoral level, with predominantly natural-language reasoning problems and machine-verifiable answer formats (MCQs, numerical answers, True/False, and matching). These include the JEE Advanced, an engineering admissions test for India's premier engineering schools; the Oxford Test of Mathematics for University Admission (TMUA), a mathematics admissions examination used by the University of Oxford and other UK universities; the Chennai Mathematical Institute (CMI) Entrance Examination for MSc/PhD admissions; the Brazilian ITA and IME entrance examinations for admission to Brazil's premier engineering institutes; and the CSIR-UGC NET Mathematical Sciences examination, a national examination for postgraduate research and academic eligibility in India (Appendix K lists all eligible exams and our rationale for these choices).

Despite filtering for challenging exams, question-solving accuracy remains sufficiently high across modern frontier LLMs. To further control for difficulty and represent the merits of our method, we construct a "Hard" set for each exam. We keep problems that GPT-5 with CoT answered wrongly in a majority of 10 attempts. Resulting yield rates are exam-specific (Table 1). We report results on a pooled set of 171 problems. Registries are induced only from past papers disjoint from, and at least two years older than, this set, and are frozen before evaluation. Since every method uses the same solver models, memorization of these public papers would lift the baselines as much as MathTypes.

Table 1: Yield rates across the evaluated mathematics examinations.

Exam name	N	Yield rate
JEE Advanced	35	6.7% (35/526)
Oxford TMUA	15	7.5% (15/200)
CMI Entrance MSc/PhD	17	24.6% (17/69)
Brazil IME	32	21.6% (32/148)
Brazil ITA	32	27.4% (32/117)
CSIR-NET	40	8.3% (40/480)
Total	**171**	**11.1% (171/1540)**

Seed grammar and induction process: We note that building a type registry exhaustively on all of mathematics would be difficult. Systems like Lean have been made possible through decades of collective effort from experts from across the community. To promote automation in the process, we utilize a fully autonomous LLM-based seed grammar creation followed by additional specialized induction to form exam-specific type registries. The seed contains 17 types and 9 theorems. This small set occupies around 4,800 tokens (4,843 with `o200k_base` and 4,726 with `cl100k_base` as the tokenizer) and contains no exam-specific, topic-specific, or answer-shaped entry. A few types are hand-authored, while the remainder are generated from a collection of practice material covering American and Indian high-school mathematics (Appendix L prints it in full). We note this is a requirement for running the deterministic verifier from the first problem while defining common mathematics operations whose rediscovery can be avoided while building the per-exam type registry (sensitivity to the seed is tested by building a grammar from just the informal hatches in Section 4.3).

Each problem, along with its expert solution and the current registry, is input to the annotator model, which first transcribes the image and then writes a step-level typed annotation. Multiple frontier LLMs were evaluated for the purpose, and Qwen3.8 Max was finally chosen as it maximized the cost-performance Pareto (Section 4.1). The annotation is first validated and verified with the existing grammar (seed corpus for the first problem) to discard any immediate failures. For correct problems, if no proposals exist in the `.mt` program, this concludes the induction process for the specific problem. Those with proposals go through a two-step process: first, the annotator LLM writes a justification for the proposed types and theorems. The proposals, justifications, existing type registry, problem context, and preliminary test results (such as duplicacy warnings through Jaccard overlap) are then passed to a reviewer LLM with an evaluation rubric that asks it to mechanically verify whether the proposal is similar to a method in the current list and judge the level of generality and truth of the proposed hypotheses. The reviewer returns one of three decisions per proposal: `ACCEPT`, `ACCEPT_WITH_CHANGES`, or `REJECT`. We define an annotator's **proposal precision** as the share of `ACCEPT` among its `ACCEPT` and `REJECT` judgments, dropping `ACCEPT_WITH_CHANGES` from the denominator, as this is the reviewer's work in part.

We let the reviewer take decisions autonomously over 2 rounds of correction and verification. The round budget is motivated by cost, as each additional automated loop linearly increases the amount of spent credits. Proposals that do not reach consensus within this budget are not merged. Accepted proposals merge into the registry, except that a proposal that correctly declares that no existing verification strategy fits is staged for engineering review instead (Appendix I). A second LLM, the auditor, runs periodically across the entire registry to merge similar entries, generalize over-specific ones, and perform provenance audits. We introduce it specifically to combat Qwen's behavioral quirks (Section 4.1). Both reviewers mechanically verify types by testing with a wide range of inputs. Theorems must satisfy seven tests: that it is a theorem rather than a definition or an operation; that it is not a one-line corollary of a registered entry; that it is not half of a result whose other half is also being proposed; that its hypotheses are complete; that its verification body can actually fail against a false claim; that it is stated at the generality of the parent fact rather than the instance the problem needed; and that two proposals from a problem are not two steps of it.

3.3 Experiments

For building the type grammar, we utilize Alibaba's Qwen3.8 Max (0902) as the annotator. Gemini 3.1 Pro acts as the validator LLM, with Claude Opus 5 on Claude Code as the auditor. These choices are motivated by experiments in Section 4.1. All evaluations are carried out on 171 problems across six exams as shown in Table 1. The temperature and thinking are left at the API provider's defaults, as best settings are pre-tuned. We test Gemma 4 31B, Qwen3.6 27B, Qwen3.8 27B, and Nemotron 3 Nano Omni 30B among open mid-tier models, with GLM 5.3 Flash, Muse Spark 1.3, and Gemini 3.8 Flash as frontiers. As baselines, we compare MathTypes against Chain of Thought (CoT), Program of Thoughts (PoT), Plan-and-Solve (PS), Self-Check (SC), Pedagogical CoT (PedCoT), and SymCode. Ablations include a seed sensitivity test (bootstrapping one family from the escape hatches only); vocabulary size test; registry quality pre- versus post-audit; varying correction budget between 0 and 3; and isolating the type registry from the verification loop.

4 Results

4.1 Choosing the Annotator and Reviewer LLMs

Table 2: Analysis of MathTypes annotation capabilities across seven frontier models on N=30 problems from the seed corpus. The LLM reviewer used for proposal precision is Gemini 3.1 Pro. For % Claim-blind, cost/problem, mean latency, and informal step rate, lower is better.

Annotator	Proposal Precision (N)	Proposals/ Problem	% Claim-blind (N)	Binds Claim	Cost/ Problem	Mean Latency	Coverage	Informal Step Rate
Claude Opus 5	**39.1%** (64)	3.03	62% (90)	0%	$0.837	184s	99.3%	0.7%
Qwen3.8 Max	35.8% (53)	3.00	51% (88)	44%	$0.381	816s	96.9%	3.1%
Kimi K3	21.2% (66)	3.50	59% (105)	37%	$0.675	747s	98.9%	1.1%
Gemini 3.1 Pro	14.3% (35)	1.93	**24%** (58)	**76%**	$0.640	271s	90.5%	9.5%
Muse Spark 1.2	3.3% (30)	1.57	68% (47)	32%	**$0.152**	148s	87.9%	12.1%
Gemini 3.6 Flash	2.8% (36)	1.70	82% (50)	8%	$0.248	**98s**	99.5%	**0.5%**
GPT-5.6 Sol	2.7% (73)	3.20	66% (96)	32%	$0.819	237s	95.2%	4.8%

A pilot analysis (Table 2) over 30 problems from the seed grammar creation corpus (disjoint from the evaluation set) and seven candidate annotator models was used to identify the ideal combination. The task can be demanding, and we only utilize frontier LLMs for this purpose. Here, proposal precision is the primary selection criterion, as it answers how often a model proposes registry vocabulary that is usable as written. We note that Claude Opus 5 and Qwen3.8 Max are the best LLMs by this metric, outperforming the third-best model, Kimi K3, by a wide margin (39.1% and 35.8% against 21.2%). A qualitative analysis showed that Gemini 3.6 Flash and Muse under-propose and sometimes evade formalizing critical steps. This is reflected in their high claim-blind rates and Muse Spark's 12.1% informal step usage. Moreover, none of Opus 5's 90 theorems read the injected claim, scoring the worst "binds claim" result. Qualitatively, it writes correct mathematics, but fails to contextualize them to the problem, reducing its effectiveness. Finally, we choose Qwen3.8 Max given a combination of the second-best proposal precision, claim-blindness, and bounded claim rates; an available open-weights alternative (Qwen3.8 2.4T); and a low cost.

Among the reviewer LLMs, we test Gemini 3.1 Pro and Claude Opus 5. Other models were excluded if they failed to propose usable vocabulary (GPT-5.6 Sol, Muse Spark 1.2) or had high latency (Kimi K3, Qwen3.8 Max). The two judges agree at $\kappa = 0.285$ on the binary accept/reject with 63.2% raw agreement, which is fair. Of 196 binary disagreements, Opus 5 was the lenient party in 152 (78%). It accepts its own proposals 65.5% (self-preference of $+0.358$) against Gemini 3.1 Pro's 14.3% (self-preference of -0.050). For three reasons, we select Gemini 3.1 Pro: (i) its low claim-blind tendency (which, in turn, reduces Qwen3.8 Max's claim-blind rate); (ii) its negative self-preference (-0.050); and (iii) it is cheaper among the two. One reviewer, however, is not structurally sufficient, as a problem-level judge only operates on the proposals of a single `.mt` program. For example, 51% of Qwen3.8 Max's raw theorem proposals could not reject a false claim. After Gemini review, 8 of 55 (15%) still could not reject a false claim. Therefore, we utilize Claude Opus 5 as a theorem-level auditor in a plan-and-solve harness (Claude Code), specifically because it is more lenient and therefore preserves more registry entries: strictness is enforced per problem by the reviewer, whereas the auditor generalizes across the whole registry, where a strict judge would delete capability.

4.2 Mathematical reasoning with MathTypes

Table 3: Results on the pooled set of math exam evaluations (6 exams: JEE Adv, TMUA, ITA, IME, CMI MSc/PhD, CSIR-NET Math). N=171 over 5 runs each. All iterative methods are capped at four passes. "MathTypes cumulative" grades answers independently at every correction step and considers a question correct if it produced the right answer at any step, as an accuracy ceiling. Accuracy (%) $\pm$ one standard error from a question-level cluster bootstrap over 10k samples.

Method	Gemma 4 31B	Qwen 3.6 27B	Qwen 3.8 27B	Nemotron 3 Nano Omni 30B	GLM 5.3 Flash	Muse Spark 1.3	Gemini 3.8 Flash
CoT (Wei et al., 2022)	48.1±3.8	27.1±3.3	36.0±3.5	19.7±2.6	41.6±3.3	57.4±3.7	60.7±3.7
PoT (Chen et al., 2023)	25.1±3.1	23.1±3.0	27.6±3.4	22.3±3.1	22.8±2.9	35.1±3.2	50.0±3.6
PS (Wang et al., 2023)	45.7±3.6	27.8±3.2	32.5±3.6	22.3±3.1	44.7±3.6	58.1±3.6	60.2±3.8
PedCoT (Jiang et al., 2024)	44.1±3.6	27.7±3.3	38.5±3.7	24.0±3.2	40.6±3.6	56.5±3.7	58.4±3.7
SelfCheck (Miao et al., 2024)	45.5±3.5	24.3±3.0	31.5±3.5	21.6±2.9	43.3±3.7	57.1±3.7	58.4±3.7
SymCode (Bagheri Nezhad et al., 2026)	48.3±3.6	40.9±3.7	46.0±3.4	21.6±2.5	34.9±3.2	54.8±3.6	53.8±3.8
MathTypes (Ours)	58.7±3.4	58.5±3.2	58.5±3.3	37.1±3.8	60.3±3.7	69.6±3.4	69.8±3.5
MathTypes cumulative	**62.0±3.5**	**60.1±3.2**	**60.8±3.4**	**45.4±4.1**	**60.4±3.7**	**69.8±3.4**	**70.1±3.4**

Our main results on the pooled set of difficult math reasoning problems (N=171) are presented in Table 3. MathTypes is the most accurate method for all seven models, ahead of the strongest baseline in each column by +9.1 (Gemini 3.8 Flash) to +17.6 points (Qwen 3.6); paired over shared questions, the margin exceeds two standard errors for all seven. We note that PoT, the baseline requiring generic Python verification, falls below CoT for six of the seven models, and SymCode, which adds a debugging loop, beats CoT for four but trails MathTypes for all seven. Besides performance gains, our method also yields reasoning chains whose typed steps are machine-checked. Across all models and methods, a MathTypes configuration produces the single best result on three of the six exams (JEE Advanced, CMI, IME) and ties for the best on two (TMUA, CSIR-NET). It is outscored only on ITA (41.7% vs. PedCoT with Muse Spark 1.3's 51.9%). Further, plotting pooled accuracy vs. completion tokens shows that MathTypes extends the current Pareto frontier (Figure 2). Our method places two models on the frontier: Muse Spark 1.3 (69.6% at 6.7k

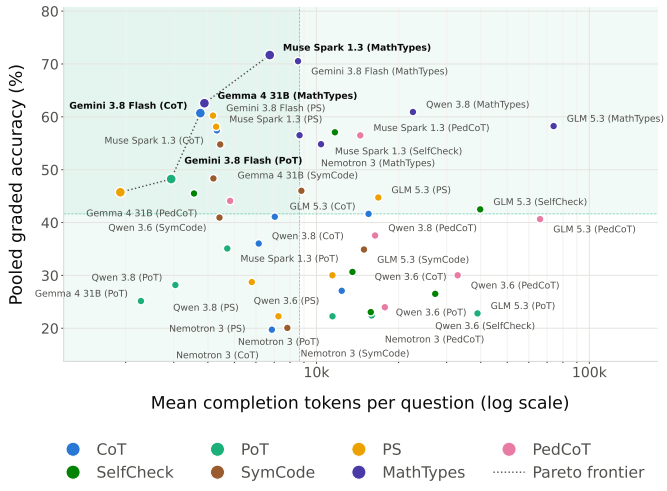

Figure 2: Pooled accuracy vs. completion tokens. MathTypes extends the Pareto frontier and holds two models on it, the most of any method.

tokens per question) and Gemma 4 31B (58.7% at 3.9k), ahead of other methods which make one or no contributions.

4.3 Ablations and Analyses

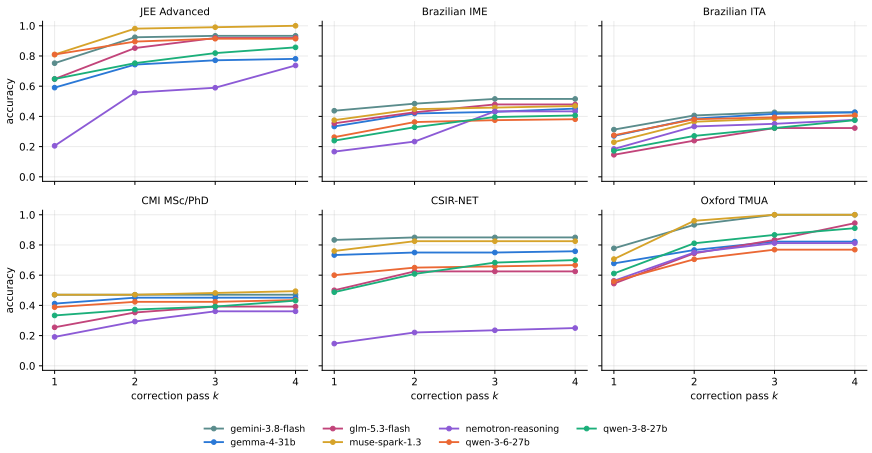

Figure 3: Cumulative question-solving accuracy per exam by correction pass, calculated as the share of correct answers by pass $\leq k$ as a fixed denominator.

We first analyze how the evaluator loop assay improves performance. Figure 3 plots the cumulative question-solving accuracy curve per pass, the share of all questions answered correctly at some pass $\leq k$. We note that 68% of the total gain lands in the first correction pass, a mean improvement of +10.9 points (more than 1 point in 40 of 42 panels). The second and third corrections add +3.6 and +1.5 points. About 90% of the achievable gain is therefore reached by pass 3, showing that MathTypes does not require significant test-time scaling, explaining the observed trends in Figure 2.

To isolate the source of the gain, we run two controls on Gemma 4 31B and Muse Spark 1.3 (Table 4; prompts in Appendix M). C1 gives CoT the exam's registry as plain-text context, with no typed syntax or verifier. It changes CoT's accuracy by at most 1.5 points, and MathTypes beats it by 11.3–13.7 points, so registry knowledge alone does not cause the gain. C2 keeps the loop but, after a verifier rejection, tells the LLM only that its program contains an error. Despite using more passes, it loses 4.1–5.8 points against MathTypes, so the verifier's step-level diagnostics carry part of the gain.

Table 4: Paired accuracy differences (points, ± 1 SE over questions) for two controls. C1: CoT with the exam registry as plain-text context. C2: MathTypes whose correction passes are told only that the program contains an error.

Comparison	Gemma 4 31B	Muse Spark 1.3
C1 − CoT	-0.7 ± 2.8	-1.5 ± 1.2
MathTypes − C1	$+11.3 \pm 3.1$	$+13.7 \pm 3.3$
C2 − MathTypes	-5.8 ± 2.1	-4.1 ± 1.2
Passes/question (C2 vs. MathTypes)	2.65 vs. 2.22	1.87 vs. 1.70

Further, to determine the effectiveness of the second auditor LLM, which is supposed to remove over-specific and redundant vocabulary without removing capability, we run a pre- and post-audit question-solving accuracy test. On JEE Advanced, it edited the registry from 22 types and 84 theorems to 24 types and 75 theorems. Results on GLM 5.3 Flash and Muse Spark 1.3 are presented in Table 5. While GLM shows a measurable gain in capability, we note that Muse, which was already at the performance ceiling, remains untouched. We achieve this while reducing the prompt block from 10,477 tokens to 6,716, reducing input costs by 35.89%. Further, the post-audit registry required fewer correction passes (Muse 1.71 against 2.52 pre-audit; GLM 2.62 against 2.69). Output tokens per question fell accordingly (Muse 6,023 against 8,583; GLM 48,411 against 54,980). This reduces per-question cost for GLM by 8.0% and Muse by 20.9%, motivating the use of a second auditor.

Table 5: Accuracy on JEE Advanced (N=35) with MathTypes registries before and after the second auditor LLM's edits. We report means over 3 runs.

Model	Pre-audit	Post-audit	Δ
GLM 5.3 Flash	84.8	91.0	+6.2
Muse Spark 1.3	100.0	100.0	+0.0

We also test the impact of using the 17-type and 9-theorem seed on performance by bootstrapping a type registry from the escape hatches alone. The same number of problems were annotated in both cases, with the induced registry containing 14 types and 35 theorems with the empty seed, against our proposed method's 26 types and 56 theorems. In the empty seed grammar, only seven of the 17 types in the seed grammar were reconstructed. None of the seed's theorems were found either by a lexical near-duplicate test or an LLM judge (Gemini 3.1 Pro). While dropping the seed made constructing the grammar cheaper ($36.53 against $46.07), the newly constructed grammar lost question-solving accuracy (Table 6) across two LLM backbones: Gemma 4 31B and Muse Spark 1.3, although the magnitude depends on the available ceiling. Notably, 46 of 142 passes with Gemma and 75 of 174 passes with Muse referenced an "unknown" type with the empty seed (32.4–43.1%), meaning the model accessed an undefined operation, against 0.0–6.3% with our proposed induction-based pipeline (Appendix C), motivating our two-stage pipeline.

Table 6: Accuracy on the induced registry versus the production registry on a single exam family, TMUA. Runs are repeated five times.

Model	Induced registry (14T / 35Th)	Production (26T / 56Th)	Δ
Gemma 4 31B	48.9 ± 9.8	68.9 ± 8.0	-17.0
Muse Spark 1.3	98.7 ± 1.3	100.0	-1.3

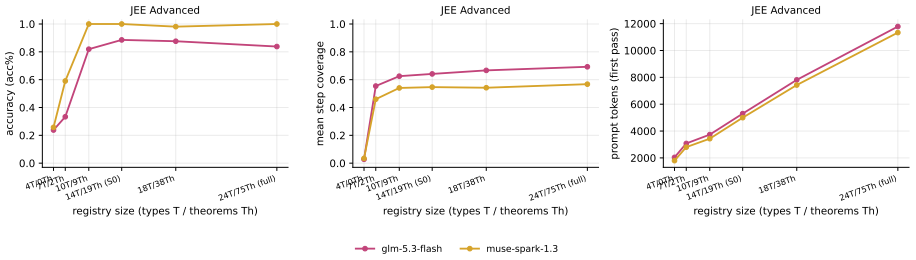

Figure 4: Sensitivity to MathTypes registry size across six configurations with increasing type and theorem counts. Left to right: accuracy, coverage, and total input tokens vs. registry size.

Figure 4 plots accuracy, coverage, and input prompt token count against the registry size across six configurations with increasing counts of types and theorems on JEE Advanced, included on a causal rate of their appearance. Both accuracy and formalization coverage saturate after the registry reaches 10 types and 9 theorems, suggesting that accuracy gains with MathTypes are not contingent on large-scale grammar creation and can be achieved cheaply (the 10T/9Th registry cost $1.29).

We also compare our verifier against Gemini 3.1 Pro used as a PRM with ProcessBench's template (Zheng et al., 2025) (Table 7). Judging 150 solutions five times each, the PRM returns the same verdict in every run for only 82.7% of them, and two calls disagree on whether a correction should fire 8.0% of the time. Re-verifying all 8,120 MathTypes programs five times changed no verdict. On 385 solutions stratified by correctness and by agreement with the PRM's own answer, agreement raises acceptance by about 24 points, but correctness by only 2–3, suggesting that the PRM largely compares final answers rather than checking steps.

Table 7: Gemini 3.1 Pro as a PRM vs. the MathTypes verifier. (a) Repeatability over 5 runs. (b) PRM acceptance rate (%) by answer correctness and agreement with the PRM's own answer.

(a) Repeatability	PRM	MathTypes
Same verdict in all runs	82.7%	100%
Correction decision flips	8.0%	0%
(b) Acceptance	**Agrees**	**Disagrees**
Wrong solution	75.5	51.5
Right solution	77.9	54.8

5 Conclusion

We introduced MathTypes, a typed language that checks each step of mathematical reasoning. Automatically growing the type registries with an LLM-based pipeline, we created a grammar scoped to their syllabuses. Evaluations with our pipeline showed significant accuracy improvements across six examinations. Detailed ablations showed that MathTypes enables the use of such registries, but their size matters less than expected, and a seed-based induction pipeline is important. Through MathTypes, we show that much of natural-language exam reasoning can be checked step by step without significant compute. Future work should explore using MathTypes verdicts as training and reinforcement learning rewards and a hand-grown type registry covering all of mathematics.

AI USE STATEMENT

In this work, we used generative AI tools for the method itself: the annotator, reviewer, and auditor that grow the type registries are LLMs (Section 3.2), used deliberately to show that MathTypes can scale without manual grammar authoring. Every registry entry they produce is checked by the deterministic verifier, and all reported accuracies are graded against official answer keys. We also used generative AI tools to debug experiment code (Claude Code) and to retrieve relevant papers when double-checking our literature review (Allen AI's Asta). All AI-assisted code was tested by the authors, and every retrieved paper was read and verified before being cited. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work involves no human subjects and no personal data. All problems and expert solutions come from publicly released past examination papers and their official answer keys, used for research purposes only. Our evaluation set is filtered for difficulty and may not represent each examination's full syllabus, so results should not be read as a measure of general mathematical ability. MathTypes is a verification tool: a PASS verdict certifies only the steps expressed through registered types, and unverified escape-hatch steps are reported explicitly so that coverage is never overstated. We are not aware of conflicts of interest or sponsorship that bear on this work.

REPRODUCIBILITY STATEMENT

The MathTypes language, compiler, and verifier are described in Section 3.1, and the registry induction pipeline, including annotator and reviewer selection, in Sections 3.2 and 4.1. The construction of the difficulty-filtered evaluation set and its per-exam yield are given in Section 3.2 and Table 1, and the evaluated models and baselines are presented in the Experiments section. We will release the source code, the seed and per-exam type registries, the evaluation set manifest, prompts, and all stored model outputs and verifier reports as anonymized supplementary material. Because the verifier is deterministic, every reported verdict can be re-derived exactly from the stored programs; re-verifying all 8,120 stored programs five times reproduced every verdict (Table 7). Model versions and serving providers are recorded for every call.

REFERENCES

Andrea Asperti, Alberto Naibo, and Claudio Sacerdoti Coen. Thinking machines: Mathematical reasoning in the age of llms. *Big Data and Cognitive Computing*, 10(1), 2026. ISSN 2504-2289. doi: 10.3390/bdcc10010038. URL https://www.mdpi.com/2504-2289/10/1/38.

Sina Bagheri Nezhad, Yao Li, and Ameeta Agrawal. SymCode: A neurosymbolic approach to mathematical reasoning via verifiable code generation. In Vera Demberg, Kentaro Inui, and Lluís Marquez (eds.), *Findings of the Association for Computational Linguistics: EACL 2026*, pp. 1489–1503, Rabat, Morocco, March 2026. Association for Computational Linguistics. ISBN 979-8-89176-386-9. doi: 10.18653/v1/2026.findings-eacl.76. URL https://aclanthology.org/2026.findings-eacl.76/.

Edwin C. Brady. Idris —: systems programming meets full dependent types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*, PLPV '11, pp. 43–54, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304870. doi: 10.1145/1929529.1929536. URL https://doi.org/10.1145/1929529.1929536.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=YfZ4ZPt8zd.

Jie Cheng, Gang Xiong, Ruixi Qiao, Lijun Li, Chao Guo, Junle Wang, Yisheng Lv, and Fei-Yue Wang. Stop summation: Min-form credit assignment is all process reward model needs for rea-

soning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=3Sxby0hH1q.

Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2): 56–68, 1940. ISSN 00224812. URL http://www.jstor.org/stable/2266170.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2–3):95–120, February 1988. ISSN 0890-5401. doi: 10.1016/0890-5401(88)90005-3. URL https://doi.org/10.1016/0890-5401(88)90005-3.

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

Amirhossein Dabiriaghdam, Shayan Vassef, Mohammadreza Bakhtiari, Yasamin Medghalchi, Ilker Hacihaliloglu, Mesrob Ohannessian, Lele Wang, and Giuseppe Carenini. Vamps: Visual-assisted mathematical problem solving benchmark, 2026. URL https://arxiv.org/abs/2606.04244.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp (eds.), *Automated Deduction - CADE-25*, pp. 378–388, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21401-6.

Tony Feng, Trieu H. Trinh, Garrett Bingham, Dawsen Hwang, Yuri Chervonyi, Junehyuk Jung, Joonkyung Lee, Carlo Pagano, Sang hyun Kim, Federico Pasqualotto, Sergei Gukov, Jonathan N. Lee, Junsu Kim, Kaiying Hou, Golnaz Ghiasi, Yi Tay, YaGuang Li, Chenkai Kuang, Yuan Liu, Hanzhao Lin, Evan Zheran Liu, Nigamaa Nayakanti, Xiaomeng Yang, Heng-Tze Cheng, Demis Hassabis, Koray Kavukcuoglu, Quoc V. Le, and Thang Luong. Towards autonomous mathematics research, 2026. URL https://arxiv.org/abs/2602.10177.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

Henrique Godoy. Alvorada-bench: Can language models solve brazilian university entrance exams?, 2025. URL https://arxiv.org/abs/2508.15835.

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

Yichen Huang and Lin F. Yang. Winning gold at imo 2025 with a model-agnostic verification-and-refinement pipeline, 2025. URL https://arxiv.org/abs/2507.15855.

Zhuoxuan Jiang, Haoyuan Peng, Shanshan Feng, Fan Li, and Dongsheng Li. Llms can find mathematical reasoning mistakes by pedagogical chain-of-thought. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, IJCAI '24, 2024. ISBN 978-1-956792-04-1. doi: 10.24963/ijcai.2024/381. URL https://doi.org/10.24963/ijcai.2024/381.

Nadine Karsten, Frederik Krogsdal Jacobsen, Kim Jana Eiken, Uwe Nestmann, and Jørgen Villadsen. Proofbuddy: A proof assistant for learning and monitoring. *Electronic Proceedings in Theoretical Computer Science*, 382:1–21, August 2023. ISSN 2075-2180. doi: 10.4204/eptcs.382.1. URL http://dx.doi.org/10.4204/EPTCS.382.1.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=v8L0pN6EOi.

Chengwu Liu, Ye Yuan, Yichun Yin, Yan Xu, Xin Xu, Zaoyu Chen, Yasheng Wang, Lifeng Shang, Qun Liu, and Ming Zhang. Safe: Enhancing mathematical reasoning in large language models via retrospective step-aware formal verification. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12171–12186, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.594. URL https://aclanthology.org/2025.acl-long.594/.

Romina Mahinpei, Manoel Horta Ribeiro, and Mae Milano. Interactive theorem provers for proof education. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on SPLASH-E*, SPLASH-E '25, pp. 24–41, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400721427. doi: 10.1145/3758317.3759679. URL https://doi.org/10.1145/3758317.3759679.

Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson (eds.), *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pp. 73–118. Elsevier, 1975. doi: https://doi.org/10.1016/S0049-237X(08)71945-1. URL https://www.sciencedirect.com/science/article/pii/S0049237X08719451.

Ning Miao, Yee Whye Teh, and Tom Rainforth. Selfcheck: Using LLMs to zero-shot check their own step-by-step reasoning. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=pTHfApDakA.

Seyed Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. GSM-symbolic: Understanding the limitations of mathematical reasoning in large language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=AjXkRZIvjB.

Arka Mukherjee and Shreya Ghosh. mmJEE-eval: A bilingual multimodal benchmark for evaluating scientific reasoning in vision-language models. In Kentaro Inui, Sakriani Sakti, Haofen Wang, Derek F. Wong, Pushpak Bhattacharyya, Biplab Banerjee, Asif Ekbal, Tanmoy Chakraborty, and Dhirendra Pratap Singh (eds.), *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pp. 2268–2290, Mumbai, India, December 2025. The Asian Federation of Natural Language Processing and The Association for Computational Linguistics. ISBN 979-8-89176-303-6. doi: 10.18653/v1/2025.findings-ijcnlp.140. URL https://aclanthology.org/2025.findings-ijcnlp.140/.

Sagnik Nath, Edith Aurora Graf, Liang Zhang, and Diego Zapata-Rivera. Representation robustness under executable reasoning constraints in large language models for mathematical problem solving. In *Artificial Intelligence in HCI: 7th International Conference, AI-HCI 2026, Held as Part of the 28th HCI International Conference, HCII 2026, Montreal, QC, Canada, July 26–31, 2026, Proceedings, Part II*, pp. 155–171, Berlin, Heidelberg, 2026. Springer-Verlag. ISBN 978-3-032-30859-7. doi: 10.1007/978-3-032-30860-3_11. URL https://doi.org/10.1007/978-3-032-30860-3_11.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Azim Ospanov, Zijin Feng, Jiacheng Sun, Haoli Bai, SHEN XIN, and Farzan Farnia. HERMES: Towards efficient and verifiable mathematical reasoning in LLMs. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=w7BZcUc0fJ.

Ivo Petrov, Jasper Dekoninck, Lyuben Baltadzhiev, Maria Drencheva, Kristian Minchev, Mislav Balunovic, Nikola Jovanović, and Martin Vechev. Proof or bluff? evaluating LLMs on 2025 USA math olympiad. In *2nd AI for Math Workshop @ ICML 2025*, 2025. URL https://openreview.net/forum?id=3v650rMO5U.

Goun Pyeon, Inbum Heo, Jeesu Jung, Taewook Hwang, Hyuk Namgoong, Hyein Seo, Yerim Han, Eunbin Kim, Hyeonseok Kang, and Sangkeun Jung. Evaluating large language models on the 2026 korean csat mathematics exam: Measuring mathematical ability in a zero-data-leakage setting, 2025. URL https://arxiv.org/abs/2511.18649.

Marcus Rossel, Rudi Schneider, Thomas Kœhler, Michel Steuwer, and Andrés Goens. Towards pen-and-paper-style equational reasoning in interactive theorem provers by equality saturation. *Proc. ACM Program. Lang.*, 10(POPL), January 2026. doi: 10.1145/3776667. URL https://doi.org/10.1145/3776667.

Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262, 1908. doi: 10.2307/2272708.

Frédéric Tran Minh, Laure Gonnord, and Julien Narboux. Proof assistants for teaching: a survey. *Electronic Proceedings in Theoretical Computer Science*, 419:1–27, May 2025. ISSN 2075-2180. doi: 10.4204/eptcs.419.1. URL http://dx.doi.org/10.4204/EPTCS.419.1.

Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024. doi: 10.1038/s41586-023-06747-5.

George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Anja Surina, Moritz Firsching, Gergely Bérczi, Francisco J. R. Ruiz, Arun Suggala, Adam Zsolt Wagner, Eric Wieser, Lei Yu, Aja Huang, Miklós Z. Horváth, Andrew Ferraiuolo, Henryk Michalewski, Edward Lockhart, Codrut Grosu, Thomas Hubert, Matej Balog, Pushmeet Kohli, and Swarat Chaudhuri. Advancing mathematics research with ai-driven formal proof search, 2026. URL https://arxiv.org/abs/2605.22763.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel R. Bowman. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=bzs4uPLXvi.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2609–2634, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.147. URL https://aclanthology.org/2023.acl-long.147/.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce LLMs step-by-step without human annotations. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.510. URL https://aclanthology.org/2024.acl-long.510/.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

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. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Fangchen Yu, Junchi Yao, Ziyi Wang, Haiyuan Wan, Youling Huang, Bo Zhang, Shuyue Hu, Dongzhan Zhou, Ning Ding, Ganqu Cui, LEI BAI, Wanli Ouyang, and Peng Ye. Physicsminions: Winning gold medals in the latest physics olympiads with a coevolutionary multimodal multi-agent system, 2026. URL https://openreview.net/forum?id=kipQYpoZf1.

Chujie Zheng, Zhenru Zhang, Beichen Zhang, Runji Lin, Keming Lu, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. ProcessBench: Identifying process errors in mathematical reasoning. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1009–1024, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.50. URL https://aclanthology.org/2025.acl-long.50/.

Appendices

This supplementary material presents additional details on the following aspects:

- **Appendix A:** The MathTypes compiler
- **Appendix B:** MathTypes loop metrics
- **Appendix C:** Results by exam
- **Appendix D:** Accuracy by verification status
- **Appendix E:** Inference cost
- **Appendix F:** Proposal-quality metrics
- **Appendix G:** Comparison with Lean-based auto-formalization
- **Appendix H:** False acceptance under falsified claims
- **Appendix I:** Verification strategies
- **Appendix J:** The informal-step overuse threshold
- **Appendix K:** Exam selection
- **Appendix L:** Seed grammar
- **Appendix M:** Prompts

A THE MATHTYPES COMPILER

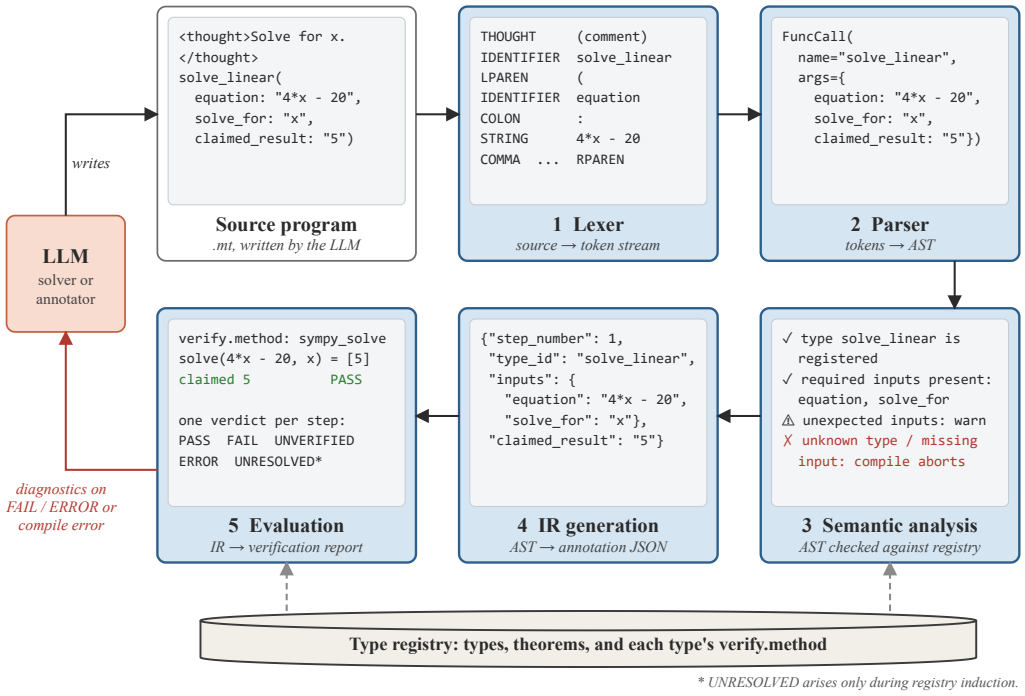

Figure 5: The five-stage MathTypes compiler on a one-step program. Stages 1–4 are structural; only stage 5 executes mathematics, through the fixed strategies of Appendix I. The type registry is read by semantic analysis and by evaluation. Compile errors and `FAIL`/`ERROR` verdicts return to the model as diagnostics.

A `.mt` program is compiled in five stages (Figure 5).

1. Lexer. The source is read line by line. `<thought>` blocks become comment tokens, any other HTML-style tag is stripped, and lines beginning with `#` are comments. Metadata lines of the form `@key "value"` become directive tokens. Every other line is split into identifiers, parentheses, colons, commas, and string literals. A brace-delimited dictionary literal, as used for theorem conditions, is kept as a single string token.

2. Parser. The token stream is parsed into an abstract syntax tree (AST): an ordered list of calls of the form `name(param: "value", ...)`, each recorded with its type name and a map from parameter names to string values. A malformed call is a parse error and ends compilation.

3. Semantic analysis. Each call is checked against the type registry. Its name must be a registered type, and it must supply every input that the type's signature declares. A signature may instead declare a choice between groups of inputs, in which case at least one group must be supplied in full. `claimed_result` is a keyword of the language and is exempt from the signature check. Unexpected inputs raise a warning. An unknown type or a missing input is an error, which aborts compilation and is returned to the model as a diagnostic.

4. IR generation. The validated AST is lowered to the intermediate representation: one JSON record per step holding its type, its inputs (dictionary literals are parsed into objects), and its claimed result, together with a program-level count of escape-hatch steps.

5. Evaluation. Each step is dispatched on its type's `verify.method` to one strategy from Appendix I, which recomputes the step from its inputs and compares the outcome with `claimed_result`. Every step runs in a separate worker process with a 10-second limit. The strategy's outcome determines the verdicts of Section 3.1: `PASS` or `FAIL` when the check runs; `UNVERIFIED` for the escape hatches; and `ERROR` when an input cannot be parsed, the check raises an exception, or the step times out. During registry induction, a step that uses a type or theorem proposed in the same program is verified against the proposed definition. If it passes, it is marked `UNRESOLVED` until the proposal has been reviewed; if it fails, it remains a `FAIL`.

Feedback. A compile error, or any step that is `FAIL` or `ERROR`, returns the diagnostic text to the model, which revises its program in the same conversation. The diagnostic never contains the reference answer.

B MATHTYPES LOOP METRICS

We define a few metrics to test the correction loop's effectiveness. **Assay-SP Confidence** is the fraction of problems whose first program passes verification, before any correction. **Assay-Final Confidence** is the fraction whose final program passes verification after the correction loop. **Recovery rate** is the fraction of problems whose first program fails verification but whose final program passes. **Compile-error rate** is the fraction of first programs that do not compile. **Single-pass accuracy** is the graded accuracy of the first program's answer. **Formalization gain** is the graded accuracy of MathTypes minus that of CoT, over the questions both methods answered, with each question weighted equally; a positive value means that writing the solution as a verified program gained accuracy. Coverage, step accuracy, and the unfamiliar problem rate are defined in Section 3.1.

Table 8 reports these metrics on exactly the cells behind Table 3, pooled over exams with the same weighting, so its first row reproduces the MathTypes row of that table. We note that the correction loop carries part of the margin, as single-pass accuracy is 9.1–20.8 points below final accuracy, and the loop recovers 66.7–99.4% of the first programs that fail verification. Without correction, MathTypes is above the strongest baseline in Table 3 for two of the seven models (Gemma 4 and Qwen 3.6), so the loop accounts for the rest of the margin. However, it would be incorrect to compare single-pass accuracy to CoT or other methods without verification loops, as for MathTypes, the model is required to write syntax in an unfamiliar language while solving difficult math problems simultaneously. This can be solved by fine-tuning or continued pre-training on MathTypes, which we leave to future work.

Strong models know how to write correct MathTypes syntax, as seen by high Assay-Final Confidence, even if their underlying reasoning is incorrect. This can be expected of any programming lan-

guage: a correctly compiled Python program does not carry a guarantee that its associated problem is solved. Despite this, formalization improves accuracy for every model. The formalization gain is positive for all seven models, from $+9.1$ (Gemini 3.8 Flash) to $+30.3$ points (Qwen 3.6). Pass-1 compile errors are common for the open mid-sized models (11–27%, and 47.5% for Nemotron), but almost every attempt ends with a machine-gradable answer: the unfamiliar problem rate is below 5% for six models and 16.9% for Nemotron.

Table 8: Loop metrics on the cells of Table 3 (all values in %, except passes). Accuracy, confidence, recovery, and compile-error rates are question-weighted as in Table 3; coverage is the mean per program; step accuracy is pooled over the steps that were checked. The formalization gain is paired over the questions both methods answered, ± 1 SE.

Metric	Gemma 4 31B	Qwen 3.6 27B	Qwen 3.8 27B	Nemotron 3 Nano Omni 30B	GLM 5.3 Flash	Muse Spark 1.3	Gemini 3.8 Flash
Accuracy (Table 3)	58.7	58.5	58.5	37.1	60.3	69.6	69.8
Single-pass accuracy	48.9	49.4	40.7	16.3	40.9	56.5	60.4
Assay-SP Confidence	39.4	45.8	30.4	37.4	16.1	52.8	56.5
Assay-Final Confidence	84.0	95.6	84.6	77.7	82.8	98.4	99.6
Recovery rate	76.6	93.3	80.5	66.7	81.0	97.5	99.4
Compile-error rate (pass 1)	13.6	10.8	26.9	47.5	20.9	9.3	6.2
Passes per problem	2.22	2.07	2.39	2.36	2.76	1.70	1.54
Coverage	58.5	47.6	59.6	72.0	71.6	49.9	49.0
Step accuracy	93.6	97.5	96.1	83.5	95.6	99.5	99.8
Unfamiliar problem rate	4.7	1.7	1.9	16.9	0.2	0.0	0.8
Formalization gain	$+13.9 \pm 4.0$	$+30.3 \pm 3.8$	$+21.3 \pm 3.6$	$+15.5 \pm 4.2$	$+22.5 \pm 3.6$	$+12.2 \pm 3.2$	$+9.1 \pm 2.6$

C RESULTS BY EXAM

Table 10 breaks Table 3 down by exam, with the same cells and the same MathTypes rows. MathTypes is the most accurate method for every model on JEE Advanced, and its advantage is smallest on the two Brazilian exams, where it is outscored on ITA. Per-exam intervals are wide, because each exam has 15–40 questions.

Table 9 breaks the same cells down by answer format instead. MathTypes is the most accurate method on single-select, multi-select, numeric, and Yes/No answers, with its largest margin on numeric answers (74.0% against at most 40.1%). On closed-form expressions, graded by symbolic equivalence, it is on par with the baselines (34.1% against 32.8–36.8%). JEE Advanced is mostly multi-select and numerical questions, explaining the large performance gain observed in Table 10. Expression-heavy exams, such as CMI, IME, and ITA, see more modest gains. Since optimizing for expressions requires careful adjustment of model behavior, we leave this as an open frontier for future work.

Table 9: Accuracy (%) by answer format, on the cells of Table 3: for each method, the mean over the seven models of each model's question-weighted accuracy on questions of that format. Single- and multi-select answers are option labels; Number is a numeric value; Expression is a closed-form answer; Yes/No is a verdict. N is the number of questions of each format.

Method	Single-select ($N = 31$)	Multi-select ($N = 44$)	Number ($N = 41$)	Expression ($N = 49$)	Yes/No ($N = 6$)
CoT	63.3	43.1	33.7	34.5	32.4
PoT	20.6	18.0	40.1	34.3	27.9
PS	63.2	42.5	31.4	**36.8**	39.4
PedCoT	62.1	41.8	34.6	35.3	29.8
SelfCheck	64.0	40.8	32.6	34.3	25.1
SymCode	69.3	49.0	30.5	32.8	28.6
MathTypes	**73.2**	**63.3**	**74.0**	34.1	**56.7**

We note that PoT's 2% result on CSIR-NET is an artifact of how the method parses multiple-choice questions. PoT answers these questions in two calls: the program computes a value, then a second "Closest Option" call picks the option label, using six AQuA example questions (which come from the original PoT repository and we didn't write or edit them). In 193 of Gemma's 200 CSIR-NET

Table 10: Accuracy (%) by exam, question-weighted as in Table 3, ± one standard error from a question-level bootstrap. Bold marks the best method for each model within an exam. Individual error margins are large as the number of questions ranges between 15–40 by exam.

Method	Gemma 4 31B	Qwen 3.6 27B	Qwen 3.8 27B	Nemotron 3 Nano Omni 30B	GLM 5.3 Flash	Muse Spark 1.3	Gemini 3.8 Flash
JEE Advanced							
CoT	25.7±7.5	22.3±7.0	26.9±7.4	10.9±4.9	24.6±6.3	33.7±8.0	53.1±8.2
PoT	38.9±8.1	40.0±7.8	33.3±7.8	28.6±6.8	38.6±7.0	45.1±8.3	51.4±8.5
PS	28.6±7.7	21.9±6.9	9.5±4.8	11.4±5.5	27.1±7.1	33.1±7.8	45.7±8.4
PedCoT	27.4±7.5	22.9±7.1	25.7±7.2	17.1±6.1	24.3±7.2	33.1±7.9	51.4±8.3
SelfCheck	26.3±7.2	21.9±6.9	18.1±5.5	10.0±4.4	31.4±7.9	29.1±7.4	52.9±8.3
SymCode	24.8±7.2	28.6±7.7	25.7±6.3	15.2±5.5	14.3±5.4	21.0±6.5	28.6±7.7
MathTypes	**74.3±6.4**	**89.5±3.5**	**81.0±5.9**	**61.7±7.6**	**91.0±4.5**	**100.0±0.0**	**93.3±3.7**
Oxford TMUA							
CoT	**82.7±8.8**	61.3±12.3	53.3±12.7	6.7±3.9	92.0±4.1	98.7±1.3	**100.0±0.0**
PoT	24.0±9.1	10.7±6.8	28.9±10.4	0.0±0.0	6.7±4.4	16.0±6.0	60.0±8.3
PS	77.3±9.5	65.3±10.4	53.3±13.0	25.0±10.2	86.7±8.8	98.7±1.3	**100.0±0.0**
PedCoT	74.7±10.0	68.0±9.6	60.0±12.6	16.7±9.6	90.0±7.0	98.7±1.3	97.8±2.1
SelfCheck	73.3±9.7	61.3±9.5	55.6±11.2	25.0±15.7	86.7±7.4	**100.0±0.0**	95.6±2.9
SymCode	71.1±10.8	73.3±10.7	**95.6±2.9**	26.7±7.1	80.0±8.8	**100.0±0.0**	**100.0±0.0**
MathTypes	68.9±8.0	**75.3±9.3**	86.7±6.9	**55.0±14.9**	**93.3±4.6**	**100.0±0.0**	**100.0±0.0**
CMI MSc/PhD							
CoT	35.3±11.6	14.1±7.8	**43.5±11.0**	16.5±5.6	12.9±7.2	45.9±11.2	**47.1±12.2**
PoT	35.3±10.3	29.4±9.1	35.3±11.6	14.7±8.1	17.6±9.3	23.5±8.0	38.2±9.8
PS	30.6±10.5	23.5±9.6	41.2±12.0	23.5±8.5	26.5±9.3	48.2±11.1	**47.1±12.2**
PedCoT	29.4±10.6	27.5±10.5	35.3±11.7	14.7±7.0	11.8±6.6	40.0±10.6	39.2±11.6
SelfCheck	29.4±10.0	13.7±6.8	23.5±10.3	29.4±10.2	8.8±4.6	44.7±11.2	41.2±12.1
SymCode	**47.1±11.1**	27.5±10.0	33.3±10.7	21.6±7.8	15.7±7.9	43.1±11.0	41.2±12.1
MathTypes	45.1±11.7	**41.2±10.5**	43.1±11.0	**30.6±9.1**	**39.2±11.2**	**49.4±11.1**	**47.1±12.2**
Brazil IME							
CoT	**43.8±8.8**	19.4±6.3	33.8±7.8	23.8±6.1	30.0±7.0	47.5±8.7	45.0±8.4
PoT	19.4±6.2	18.8±6.2	21.9±7.4	17.2±6.5	25.0±7.0	26.9±7.2	37.5±8.5
PS	36.9±8.0	20.8±6.5	28.1±7.9	26.6±6.6	40.6±8.4	48.1±8.3	46.9±8.8
PedCoT	29.4±7.1	16.7±6.4	**40.6±8.6**	28.1±7.3	31.2±7.9	47.5±8.5	45.3±8.6
SelfCheck	36.9±7.6	17.7±6.1	25.0±7.6	23.4±6.3	37.5±8.5	48.1±8.4	46.9±8.8
SymCode	38.5±7.3	30.2±7.9	28.1±7.0	24.0±5.7	25.0±6.4	**50.0±7.7**	37.5±8.5
MathTypes	39.6±7.8	**38.1±7.4**	37.5±8.0	**35.3±11.7**	**47.9±8.4**	46.9±8.4	**51.0±8.7**
Brazil ITA							
CoT	37.5±8.6	28.7±7.9	34.4±8.0	23.1±6.3	31.9±7.1	46.9±8.4	43.1±8.5
PoT	40.0±8.2	27.1±7.3	31.2±8.2	31.2±7.3	31.2±7.9	42.5±8.3	31.2±6.9
PS	35.6±7.8	29.2±7.5	37.5±8.6	29.7±7.6	**39.1±7.9**	48.1±8.3	**46.9±8.8**
PedCoT	36.2±7.7	28.1±7.7	34.4±8.4	**32.8±7.2**	23.4±7.0	**51.9±8.7**	43.8±8.5
SelfCheck	38.7±8.0	22.9±7.3	37.5±8.5	28.1±6.6	32.8±7.8	50.0±8.5	39.1±8.5
SymCode	37.5±8.2	**39.6±8.5**	**45.8±8.1**	21.9±5.4	26.0±6.2	46.9±8.5	**46.9±8.8**
MathTypes	**41.7±8.1**	38.1±7.4	36.5±7.5	32.2±9.3	33.3±7.4	39.6±8.3	41.7±8.7
CSIR-NET							
CoT	72.0±7.1	29.3±6.6	38.0±7.9	**24.0±4.9**	67.0±6.1	84.0±5.4	**85.0±5.6**
PoT	2.0±1.2	6.7±3.3	18.3±6.9	15.0±5.4	8.8±3.0	39.0±5.7	75.0±6.9
PS	70.5±6.5	24.4±6.9	43.3±9.1	21.7±6.6	60.0±6.7	**85.0±5.4**	**85.0±5.6**
PedCoT	71.5±6.5	24.4±6.6	46.7±9.1	23.3±6.5	**70.0±6.4**	79.0±5.9	80.0±6.4
SelfCheck	71.0±5.8	23.3±5.9	40.0±9.0	21.7±6.5	65.0±7.5	83.5±5.6	82.1±5.8
SymCode	**77.5±5.7**	55.0±7.7	65.0±5.6	23.3±5.7	59.2±5.9	82.5±5.6	82.5±6.0
MathTypes	75.8±6.0	**65.0±5.3**	**69.2±5.5**	17.6±5.5	62.5±9.9	82.5±5.6	**85.0±5.6**

cells, that second call returned prose or LaTeX instead of a label, as the AQuA examples use letters A–E. Further, Nemotron scored 0.0% on TMUA, as its programs crashed and PoT fell back to the CoT prompt, which gave a wrong or no answer. For the only program that ran, the mapping call returned prose instead of a label. Finally, SymCode falls behind CoT on JEE as it repeatedly returned SymPy objects (`540/7`, `12/5`, `4-sqrt(10)`) while the exam has strict formatting requirements. Notably, we already had appended any exam-specific requirements for all methods in the input prompt.

Table 11 breaks the unfamiliar problem rate ($\mathcal{U}$, Section 3.1) down by exam, on the same MathTypes cells as Table 10. Pooled over exams and weighted by questions. Broken out by exam, each model's rate depends on its individual coding capabilities, with Gemma 4 31B recording higher values up to 9.5%, while GLM 5.3 Flash, Muse Spark 1.3, and Gemini 3.8 Flash scoring 0.0% in most cases.

Table 11: Unfamiliar problem rate (%) by exam, on the MathTypes cells of Table 10. Pooled by exam and question-weighted.

Exam	Gemma 4 31B	Qwen 3.6 27B	Qwen 3.8 27B	Nemotron 3 Nano Omni 30B	GLM 5.3 Flash	Muse Spark 1.3	Gemini 3.8 Flash
JEE Advanced	3.8	1.0	0.0	0.4	0.0	0.0	0.0
Oxford TMUA	5.6	2.9	4.4	0.8	2.3	0.0	0.0
CMI MSc/PhD	3.9	3.5	0.0	2.8	0.0	0.0	0.0
Brazil IME	9.5	0.6	5.2	3.8	0.0	0.0	4.2
Brazil ITA	2.1	0.6	2.1	6.3	0.0	0.0	0.0
CSIR-NET	0.0	3.3	0.8	0.4	0.0	0.0	0.0

D ACCURACY BY VERIFICATION STATUS

A MathTypes solution can route part of its reasoning through `informal_step`, which is never checked. Table 12 measures how much of each model's accuracy depends on this. It uses the cells of Table 3. We note that the final answer is almost always carried by a verified step. If an answer stated only in an escape-hatch step earns no credit, accuracy falls by 0.1–4.1 points. The answer itself is therefore nearly always produced by a typed step that passed verification.

Importantly, the final answer rarely comes from the escape hatch. The share of solutions whose final answer is stated only inside an `informal_step`, rather than a verified typed step, is low: at most 1.6% for six of the seven models, and 5.7% for Nemotron.

Table 12: Accuracy by verification status (%), on the cells of Table 3. "Verified-step answers" credits a question only when its final answer comes from a typed step that passed verification. "Escape-hatch final answer" is the share of solutions whose final answer is stated only inside an UNVERIFIED `informal_step`; `translate_to_formal`, which transcribes the problem statement, is not counted as an escape hatch here.

Metric	Gemma 4 31B	Qwen 3.6 27B	Qwen 3.8 27B	Nemotron 3 Nano Omni 30B	GLM 5.3 Flash	Muse Spark 1.3	Gemini 3.8 Flash
Accuracy (Table 3)	58.7	58.5	58.5	37.1	60.3	69.6	69.8
Verified-step answers	57.9	55.3	57.5	33.0	57.7	69.5	68.8
Escape-hatch final answer	1.0	1.6	1.6	5.7	0.2	0.0	0.2

E INFERENCE COST

Every method is capped at four LLM calls per question for a *fair* comparison. Table 13 reports the calls, completion tokens, and dollar cost each method actually used, per question and pooled over exams, for the rows of Table 3. MathTypes averages 1.5–2.8 calls per question, fewer than the iterative methods, PedCoT (2.6–3.2) and SelfCheck (2.3–3.1).

Completion tokens measure what the model actually writes, and MathTypes writes fewer completion tokens than CoT for Gemma 4 and Qwen 3.6, and 1.6–2.3× as many for Nemotron, Muse Spark, and Gemini. The exceptions are GLM 5.3 Flash (4.6×) and Qwen 3.8 (3.7×). Dollar cost also counts input tokens, since the registry is re-sent on every pass. At one list price per model, MathTypes is not the costliest method for three of the six billed models (Qwen3.6 27B, Muse Spark 1.3, and Gemini 3.8 Flash). The registry block is *not* re-sent every pass but is served from a prompt cache, which helps keep costs manageable.

F PROPOSAL-QUALITY METRICS

For an annotator model m and the collection T_m of theorem verification bodies it proposes, a body *binds* the claim if it reads the step's claimed result or conclusion:

$$\text{binds_rate}(m) = \frac{|\{v \in T_m : \text{binds}(v)\}|}{|T_m|}$$

where $\text{binds}(v) = \texttt{"claimed_result"} \in v \ \vee \ \texttt{"conclusion_claimed"} \in v.$

Table 13: Mean LLM calls, completion tokens (thousands), and dollar cost (US cents) per question, pooled over exams. MathTypes rows are those of Table 3. Cost applies one list price per model to every method ($ per 1M input/output tokens: Gemma 4 31B 0.09/0.34, Qwen 3.6 0.30/2.00, Qwen 3.8 0.10/1.80, GLM 5.3 Flash 0.075/0.25, Muse Spark 1.3 0.10/0.20, Gemini 3.8 Flash 0.75/3.75); Nemotron has no public list price. Despite its heavier design, MathTypes is not the costliest method on three of the six tested models.

Method	Gemma 4 31B	Qwen 3.6 27B	Qwen 3.8 27B	Nemotron 3 Nano Omni 30B	GLM 5.3 Flash	Muse Spark 1.3	Gemini 3.8 Flash
Calls per question							
CoT	1.00	1.00	1.00	1.00	1.00	1.00	1.00
PoT	1.46	1.52	1.44	1.50	1.47	1.42	1.40
PS	1.00	1.20	1.06	1.08	1.04	1.00	1.00
PedCoT	3.16	3.02	3.16	2.57	3.03	3.03	3.02
SelfCheck	2.62	2.58	3.07	2.65	3.04	2.48	2.30
SymCode	1.13	1.25	1.34	1.50	1.42	1.07	1.06
MathTypes	2.22	2.07	2.39	2.36	2.76	1.70	1.54
Completion tokens per question (k)							
CoT	6.3	12.4	6.1	6.8	15.5	4.3	3.8
PoT	2.3	16.3	3.2	11.8	38.9	4.7	3.0
PS	1.9	11.8	6.2	7.6	16.8	4.3	4.2
PedCoT	4.8	33.9	17.4	18.2	65.9	14.5	9.6
SelfCheck	3.6	28.2	14.3	15.8	40.9	11.7	7.8
SymCode	4.2	4.4	8.8	8.1	14.9	4.4	3.0
MathTypes	3.9	8.3	22.6	10.9	71.0	7.1	8.6
Cost per question, registry served from cache (US cents)							
CoT	0.22	2.52	1.12	–	0.39	0.09	1.50
PoT	0.09	3.34	0.60	–	0.98	0.11	1.26
PS	0.07	2.51	1.15	–	0.47	0.09	1.66
PedCoT	0.23	7.42	3.32	–	1.83	0.35	4.03
SelfCheck	0.14	5.84	2.67	–	1.06	0.25	3.14
SymCode	0.18	0.98	1.61	–	0.50	0.10	1.26
MathTypes	0.36	2.23	4.35	–	2.04	0.18	3.60

A body that does not bind the claim may still depend on it through a condition: $\mathrm{cond}(v)$ holds when v matches any of `claimed_\w+`, `new_\w+`, or `target_\w+`. A body that does neither, or is empty, returns a verdict that does not depend on what the step claimed, and is *claim-blind*:

$$\mathrm{blind_rate}(m) = \frac{|\{v \in T_m : \mathrm{blind}(v)\}|}{|T_m|}, \qquad \mathrm{blind}(v) = \neg\mathrm{binds}(v) \wedge \neg\mathrm{cond}(v).$$

A high claim-blind rate suggests that a model proposes theorems ungrounded in the rest of its formalization. The ideal rate is not 0%, especially for problems with several parallel hypotheses, so the metric is relative. Both tests are syntactic, so they give only a lower bound on blindness.

G Comparison with Lean-based auto-formalization

Safe (Liu et al., 2025) auto-formalizes natural-language solution steps into Lean 4 with GPT-4o and a separate proof search stage. We report the analogous share of formalized and confirmed steps in our own results next to Safe's published numbers (Table 14) to give a rough sense of scale. This is not a controlled comparison, as the two systems run on different problems and models, "formally checked" is not defined identically across a fixed SymPy dispatch table and a Lean kernel, and Safe's numbers are taken from its paper rather than reproduced by us.

Table 14: Comparison of formalization outcomes for MathTypes and Safe. Safe's column is as reported in (Liu et al., 2025); see the disclaimer above before reading this as a head-to-head comparison.

	MathTypes	SAFE
Formally checked / formalized	58.2%	72.2%
… and confirmed	56.7% PASS	57.3% proved
Escape hatch / not formalized	37.4%	27.8%
Inside a program that never compiled	3.5%	–

Across checked and confirmed steps, both systems land near 56.7–57.3%, despite MathTypes filtering on some of the hardest problems from mathematics exams, with 27–31B open-weight LLMs writing the typed steps directly and no separate translation step or GPU-backed prover. Moreover, MathTypes averages roughly 2.2 calls per question against Safe's reported 40. We read this as evidence that a fixed, cheaper dispatch table can check a comparable share of steps to Lean autoformalization on this workload, not as evidence that the two provide equivalent guarantees. For example, a Lean proof is checked by a trusted kernel against all of dependent type theory, while a MathTypes `PASS` certifies only that one of a fixed, finite set of SymPy-backed strategies (Appendix I) accepted the step.

H FALSE ACCEPTANCE UNDER FALSIFIED CLAIMS

A verifier is only useful if it refuses false claims. We take the 2,768 steps that pass verification in the 333 accepted induction annotations (22 strategies, all six exam families), replace each step's `claimed_result` with a wrong value while leaving its inputs unchanged, and re-verify. Any `PASS` after corruption is a false acceptance. Wrong values come from three sources: code edits (negation, doubling, off-by-one, jittering the last digit, swapping operands, flipping a relation), and GLM 5.3 Flash and Muse Spark 1.3, each asked for a plausible but wrong value. Neither attacker wrote, reviewed, or audited any theorem body. Corruptions that remain equivalent to the original claim, or whose equivalence cannot be decided, are discarded before scoring; trials that end in `ERROR` stay in the denominator as not accepted.

Table 15 separates the two layers of the trusted base. Where the verifier runs a fixed SymPy strategy, false acceptance is 0.00–0.59% across the three sources, and most strategies never accept a wrong claim. `theorem_lookup`, which executes an induced theorem body, accepts 84.8–88.3% of wrong claims under every source and accounts for almost all of the pooled false acceptances. Of the 2,768 passing steps tested, 289 (10.4%) rely on it. The fixed dispatch table is therefore close to sound on this workload, while the theorem layer is not, and a `PASS` on a theorem step should be read as much weaker evidence than a `PASS` on a fixed strategy. Most of the remaining fixed-strategy acceptances are loosened inequalities or alternative roots that are still true of the step.

Table 15: False-acceptance rate (false accepts/scored trials) when each passing step's claimed result is replaced by a wrong value. Scored trials include `ERROR` outcomes (205, 1,017, and 1,519 for the three sources). For Muse, 1,500 of the 2,288 fixed-strategy trials ended in `ERROR` because no parseable value was returned.

Verification strategy	Code edits	GLM 5.3 Flash	Muse Spark 1.3
Fixed SymPy strategies, pooled	0.59% (46 / 7,808)	0.49% (12 / 2,447)	0.00% (0 / 2,288)
worst single strategy	`sympy_solve` 3.1%	`sympy_solve` 5.3%	none above 0
strategies with no false accepts	14 of 18	19 of 20	20 of 20
`theorem_lookup` (induced theorem bodies)	88.3% (690 / 781)	84.8% (139 / 164)	85.3% (116 / 136)
All strategies, pooled	8.78% (756 / 8,609)	5.77% (151 / 2,615)	4.78% (116 / 2,428)

I VERIFICATION STRATEGIES

Every registered type names one verification strategy in its `verify.method` field, and the verifier executes only the fixed strategies in Table 16. A solver cannot add a strategy or alter one. A type proposed during induction must name an existing strategy; a proposal that correctly declares that none fits is staged for engineering review instead of entering the registry (Section 3.2). Theorems are handled by a single strategy, `theorem_lookup`: it resolves the named theorem in the registry, binds the step's conditions and claimed conclusion, and executes the theorem's admitted verification body, which is SymPy code. Theorem bodies enter the registry only through induction and review, and they account for most of the false acceptances measured in Appendix H.

Table 16: The fixed verification strategies. "Registries" counts the five production registries (JEE Advanced, Oxford TMUA, CMI, CSIR-NET, Brazil ITA/IME) with at least one type using the strategy.

Strategy	What is recomputed and compared	Registries
Solving		
`sympy_solve`	Solves an equation for the named variable; compares the solution	5
`sympy_solve_inequality`	Solves an inequality in one variable; compares solution sets	5
`verify_integers_in_interval`	Enumerates the integers in an open or closed interval	5
`verify_filter_set`	Keeps the members of a finite set that satisfy a predicate	5
Substitution and evaluation		
`sympy_subs`	Substitutes a value for a variable in an expression and simplifies	5
`sympy_subs_eval`	Evaluates an expression under a complete assignment	5
`sympy_subs_equation`	Replaces a subexpression inside an equation	5
`verify_equation_substitution`	Substitutes values into both sides of an equation	5
`verify_relational_substitution`	Replaces a subexpression inside an inequality	1
Equivalence and rewriting		
`check_equivalence`	Checks that a rewritten expression or equation is equivalent	5
`sympy_equivalence`	Checks that a rearranged equation is equivalent (difference of sides)	5
`sympy_factor`	Factors a polynomial over the rationals	5
`verify_add_equations`	Adds two equations side by side	3
`verify_chain_equations`	Chains $a = b$ and $b = c$ into $a = c$	2
`verify_apply_to_both_sides`	Applies a function to both sides of an equation	2
`verify_relational_rewrite`	Rewrites an inequality by an order-preserving or order-reversing move	2
Extraction		
`sympy_poly_coeffs`	Extracts named coefficients of a polynomial in standard form	5
`sympy_coeff`	Extracts the coefficient of one variable	5
`verify_curve_features`	Reads slope, intercepts, centre, or radius off a line or circle	5
`verify_series_coefficient`	Extracts a coefficient of a series expansion about 0	1
Specialized		
`verify_resolve_absolute_value`	Resolves $\lvert e \rvert$ given the declared sign of e	1
`verify_condition_implication`	Decides whether one condition on a real variable implies another	2
`verify_extremum_from_bound`	Establishes a maximum or minimum from a bound and an attaining witness	1
`verify_counterexample`	Refutes a universal claim with a single witness	1
`verify_graph_transformation`	Applies a reflection, scaling, or translation to $y = g(x)$	1
`verify_contour_integral`	Evaluates a contour integral over an explicit closed contour	1
`verify_slope_and_intercept`	Reads slope and intercept off a line	0
Answer selection		
`dict_match`	Maps a computed result to a multiple-choice option	5
`verify_multiple_choice`	Maps computed results to a set of multiple-choice options	3
`verify_verdict`	Normalizes a Yes/No or True/False determination	1
Theorems and escape hatches		
`theorem_lookup`	Executes the admitted body of the named theorem	5
`none`	No check; the step is `UNVERIFIED` (`informal_step`, `translate_to_formal`)	5

J THE INFORMAL-STEP OVERUSE THRESHOLD

`informal_step` can never fail verification, which makes it a zero-risk choice for an annotator. During registry induction, an annotation in which at least 50% of the steps are `informal_step` and which proposes no new types or theorems is therefore not accepted. It is returned to the annotator with a diagnostic that lists the informal steps and asks for each to be expressed with an existing type or a proposed one. If the overuse persists after the correction budget, the problem is flagged and contributes nothing to the registry. Annotations that propose new vocabulary are exempt, because a missing type is the legitimate reason to fall back on the escape hatch.

The threshold was set against observed escape-hatch usage. In the pilot, the seven candidate annotators used `informal_step` for 0.5–12.1% of steps on average (Table 2). Across the 333 annotations accepted in the production induction runs, the median informal share is 0% (180 of 333 have none), the mean is 10.1%, and the 90th and 95th percentiles are 33.3% and 40.0%. A 50% threshold therefore sits above the 95th percentile of normal use and fires only on degenerate annotations, such as the motivating case of a practice problem annotated with 12 of 12 steps informal and no proposals. In the production runs it fired on 12 problems; 8 were repaired after the diagnostic and 4 were flagged. The 5 accepted annotations above 50% all propose new vocabulary.

K EXAM SELECTION

We surveyed 28 candidate mathematics examinations spanning five regions (Table 17) against four requirements: (i) final answers that can be checked mechanically against an official key; (ii) enough past papers with published solutions to induce a registry on years older than, and disjoint from, the evaluation years; (iii) headroom for current models; and (iv) accessible official papers. Proof-based examinations fail requirement (i), because their answers are rubric-graded arguments rather than checkable values. From the selected examinations we use only questions whose final answer is a choice, a number, or a closed-form expression. Difficulty is then controlled by the hard-set filter (Section 3.2).

Table 17: Candidate examinations and the decision for each. Region: SA South Asia, EA East Asia, EU Europe, AM Americas, ME Middle East and Central Asia.

Examination	Region	Decision and reason
JEE Advanced (India)	SA	**Selected.** MCQ, numerical, and matching answers only; large archive
CMI Entrance, MSc/PhD (India)	SA	**Selected.** Graduate level; official solutions; checkable questions only
CSIR-NET Mathematical Sciences (India)	SA	**Selected.** Graduate syllabus (analysis, algebra, topology); MCQ
Oxford TMUA (UK)	EU	**Selected.** Fully MCQ; UK coverage
Brazilian ITA	AM	**Selected.** Americas coverage; checkable questions only
Brazilian IME	AM	**Selected.** Americas coverage; checkable questions only
GATE Mathematics (India)	SA	Not selected; overlaps CSIR-NET in level and region
TIFR Graduate School Mathematics (India)	SA	Not selected; overlaps CSIR-NET in level and region
IIT-JAM Mathematics (India)	SA	Likely saturated
ISI Admission Test (India)	SA	No official answer key; proof-based second part
Chinese Gaokao	EA	Saturated; contamination risk; language
Korean CSAT (Suneung)	EA	Saturated
Japanese university entrance (Todai, Kyoto)	EA	Proof- and derivation-based
Taiwan GSAT	EA	Essay component without official answers; language
Taiwan AST	EA	Essay component without official answers; language
Cambridge STEP II and III	EU	Proof-based
Oxford MAT	EU	Discontinued; long answers are proof-based
French Concours, Polytechnique MP	EU	Proof-based; few problems per year
French Concours, Polytechnique MPI	EU	Proof-based; few problems per year
French Agrégation (written)	EU	Proof-based
Russian EGE Profile Mathematics	EU	Checkable part is easy; remainder rubric-graded
Italian Scuola Normale Superiore entrance	EU	Proof-based; primary sources unavailable
GRE Mathematics Subject Test	AM	Corpus too small for a hard subset (about 66 problems)
Society of Actuaries exams	AM	No official question papers
Casualty Actuarial Society exams	AM	No recent official papers
Iranian Konkoor	ME	Figure-dependent; IP-blocked outside of Iran
Turkish YKS / AYT	ME	Small corpus; redistribution restrictions; IP-blocked outside of Turkey
Israeli Bagrut	ME	Restricted access; open-ended answers

L SEED GRAMMAR

Tables 18 and 19 list the seed from which every exam family's registry is induced: 17 types and 9 theorems. Origin codes: **H**, hand-authored; **S**, derived from American high-school (SAT) practice problems; **W**, derived from an Indian high-school (JEE) practice book; **A**, introduced by a registry audit as a generalization of over-specific theorems previously induced from practice material or from training-split past papers. No seed entry derives from an evaluation problem. Per-family audits may later merge or rename seed entries.

M PROMPTS

This section gives the prompts of every evaluated method, rendered from the code that sent them. Text in braces is filled in per question.

CoT. The direct baseline, and the question prompt that the first call of PS, PedCoT, and SelfCheck builds on. Every method receives the question as an attached page image. The example below is for a single-answer multiple-choice question; the two format-dependent sentences for each answer format follow it.

Table 18: Seed types.

Type	Inputs	Strategy	Origin
solve_linear	equation, solve_for	sympy_solve	S
solve_inequality	inequality, variable	sympy_solve_inequality	A
substitute_value	expr, var, val	sympy_subs	S
substitute_into_equation	equation, assignments	verify_equation_substitution	S
evaluate_expression	expr, assignments	sympy_subs_eval	S
rewrite_expression	expr, target_form	check_equivalence	S
rearrange_equation	equation, target_form	sympy_equivalence	H
factor_polynomial	expr, var	sympy_factor	H
extract_coefficients	equation, form, var	sympy_poly_coeffs	S
extract_term_coefficient	equation, var	sympy_coeff	S
extract_curve_features	equation, feature	verify_curve_features	A
filter_set_by_predicate	values, predicate	verify_filter_set	A
integers_in_interval	lower, upper, lower_open, upper_open	verify_integers_in_interval	A
select_answer	result, choices	dict_match	S
apply_theorem	theorem, conditions, conclusion_claimed	theorem_lookup	S
translate_to_formal	natural_language, formal_expression	none (UNVERIFIED)	S
informal_step	claim, justification	none (UNVERIFIED)	H

Table 19: Seed theorems. Each carries a SymPy verification body executed by theorem_lookup.

Theorem	Conclusion	Origin
discriminant_sign_rule	The sign of $b^2 - 4ac$ ($a \neq 0$) decides whether a quadratic has two, one, or no real roots	S
two_line_system_solution_count	Two lines meet in one point, none, or infinitely many, according to their slopes and intercepts	S
theoretical_probability_formula	For equally likely outcomes, $P =$ favorable/total	S
translation_invariance_of_statistics	Adding k to every value shifts location statistics by k and leaves dispersion statistics unchanged	S
arithmetic_series_sum	$S_n = \frac{n}{2}(2a + (n-1)d)$	A
pythagorean_theorem	A triangle is right-angled exactly when $c^2 = a^2 + b^2$	W
law_of_sines	$a/\sin A = b/\sin B$	W
parabola_focal_distance_parametric	On $y^2 = 4ax$, the point $(at^2, 2at)$ has focal distance $\lvert a\rvert(1 + t^2)$	W
common_tangent_count_by_centre_distance	Two circles have 4, 3, 2, 1, or 0 common tangents, according to the distance between their centres	W

```
You are taking the {exam} ({subject}) exam. The question is shown in the attached image,
     including its answer choices.

Exactly ONE of the choices shown is correct.

Work through the problem rigorously, showing your reasoning. Then, on the very last line of
     your response, output your final answer in exactly this format, with nothing after it:

Final Answer: <your answer>

Give exactly one choice, in the exact form it is labeled in the image (e.g. "Final Answer: C
     " or "Final Answer: 2"). If none of the listed choices is correct, answer exactly "
     Final Answer: N/A".
```

```
[SingleLetter]
Exactly ONE of the choices shown is correct.
Give exactly one choice, in the exact form it is labeled in the image (e.g. "Final Answer: C
     " or "Final Answer: 2"). If none of the listed choices is correct, answer exactly "
     Final Answer: N/A".

[MultiLetter]
This question may have ONE OR MORE correct choices among those shown.
Give every choice you believe is correct, comma-separated, in the exact form the choices are
      labeled in the image (e.g. "Final Answer: A,C" or "Final Answer: 1,3,4"). No partial
     credit is given for an incomplete or over-inclusive set, so include all and only the
     choices you're confident in. If none of the listed choices is correct, answer exactly "
     Final Answer: N/A".

[Number]
This question expects a single numeric answer (no multiple-choice options).
Give a single numeric value -- a decimal or a simplified fraction, e.g. "Final Answer: 3.5"
     or "Final Answer: 7/2". Do not include units or extra text on that line.
```

```
[Expression]
This question expects a closed-form mathematical expression as the answer.
Give the exact final expression in plain math or LaTeX notation, e.g. "Final Answer: x^2 +
    1" or "Final Answer: \frac{\pi}{4}".
```

PoT. One call writes a program, which is executed. For multiple-choice questions, a second call maps the computed value to an option, using the six exemplars of the reference implementation verbatim. If the program does not run, one fallback call uses the CoT prompt.

```
Answer this question by writing a Python program.

```python
import math
import numpy as np
import sympy

# Question: the question shown in the attached image
# Answer this question by implementing a solver() function.
def solver():
    # Let's write a Python program step by step, and then return the answer
    # Firstly, we need define the following variable:
```

Complete the `solver()` function so that it returns the answer to the question in the
    attached image, then assign it with `ans = solver()`.
Output ONLY the Python program in a single ```python code block -- no explanation before or
    after it.
This is a multiple-choice question. Compute the underlying value; `ans` should be that value
    , not the option label.
```

```
Find the closest options based on the question and prediction.

Question: A company produces 420 units of a particular computer component every month, at a
    production cost to the company of $110 per component, and sells all of the components
    by the end of each month. What is the minimum selling price per component that will
    guarantee that the yearly profit (revenue from sales minus production costs) will be at
     least $626,400 ?
Options: ['A)226', 'B)230', 'C)240', 'D)260', 'E)280']
Prediction: 234.28571428571428
Closest Option: B

Question: In how many ways can the letters of the word "PROBLEC" be rearranged to make 7
    letter words such that none of the letters repeat?
Options: ['A)2!', 'B)3!', 'C)7!', 'D)8!', 'E)9!']
Prediction: 5040
Closest Option: C

Question: An exam is given in a certain class. The average (arithmetic mean) of the highest
    score and the lowest score is equal to x. If the average score for the entire class is
    equal to y and there are z students in the class, where z > 5, then in terms of x, y,
    and z, what is the average score for the class excluding the highest and lowest scorers
    ?
Options: ['A)(zy - 2x)/z', 'B)(zy - 2)/z', 'C)(zx - y)/(z - 2)', 'D)(zy - 2x)/(z -2)', 'E)(
    zy - x)/(z + 2)']
Prediction: (-2*x + y*z)/(z - 2)
Closest Option: D

Question: Find the total no. of distinct bike no.'s that can beformed using 2 letters
    followed by 2 no.'s. How many letters need to be distinct?
Options: ["A)74453", "B)64543", "C)74325", "D)65000", "E)97656"]
Prediction = 67600
Closest Option: D

Question: A wire in the shape of rectangle of length 27 cm and breadth 17 cm is rebent to
    form a square. What will be the measure of each side?
Options: ['A)9', 'B)11', 'C)22', 'D)25', 'E)31']
Prediction = [-21.42428528562855, 21.42428528562855]
Closest Option: C

Question: A point on the edge of a fan blade that is rotating in a plane 10 centimeters from
     the center of the fan. What is the distance traveled, in centimeters, by this point
    after 30 seconds when the fan runs at the rate of 300 revolutions per minutes?
Options: ['A)750pi', 'B)1500pi', 'C)1875pi', 'D)3000pi', 'E)7500pi']
Prediction: 9424.77
Closest Option: D

Question: the question shown in the attached image
```

```
Options: as labelled in the attached image
Prediction: {computed value}
Closest Option:
```

PS (Plan-and-Solve+). A second call is made only when the first reply states no answer.

```
Q: the question shown in the attached image

You are taking the {exam} ({subject}) exam. The question is shown in the attached image,
     including its answer choices.

Exactly ONE of the choices shown is correct.

Work through the problem rigorously, showing your reasoning. Then, on the very last line of
     your response, output your final answer in exactly this format, with nothing after it:

Final Answer: <your answer>

Give exactly one choice, in the exact form it is labeled in the image (e.g. "Final Answer: C
     " or "Final Answer: 2"). If none of the listed choices is correct, answer exactly "
     Final Answer: N/A".

A: Let's first understand the problem, extract relevant variables and their corresponding
     numerals, and devise a plan. Then, let's carry out the plan, calculate intermediate
     variables (pay attention to correct numeral calculation and commonsense), solve the
     problem step by step, and show the answer.
```

```
Therefore,the answer is ... Give it on the last line as 'Final Answer: <your answer>'.
```

PedCoT. A solve call, a blind Stage-1 call that regenerates the expected concepts, analyses, and calculations, a Stage-2 call that compares them with the solution's steps, and, if a step is labeled contradiction-found, one correction call.

```
You are taking the {exam} ({subject}) exam. The question is shown in the attached image,
     including its answer choices.

Exactly ONE of the choices shown is correct.

Work through the problem rigorously, showing your reasoning. Then, on the very last line of
     your response, output your final answer in exactly this format, with nothing after it:

Final Answer: <your answer>

Give exactly one choice, in the exact form it is labeled in the image (e.g. "Final Answer: C
     " or "Final Answer: 2"). If none of the listed choices is correct, answer exactly "
     Final Answer: N/A".

Set out your solution as explicitly numbered steps, starting each one on its own line as "(
     step 1)", "(step 2)", and so on.
```

```
You are given a math problem and several initial steps of the solution process.

Question: the question shown in the attached image

Initial steps:
(none yet -- this is the start of the solution)

Execute the following instructions sequentially.
1. List the mathematical concepts that should be applied in the next step for solving the
     problem.
2. List the key analyses that should be derived in the next step for solving the problem.
3. List the mathematical expressions that have not been fully computed yet in the question
     and the initial steps. Then carefully calculate their corresponding results step by
     step.
```

```
You are given a math problem and the full solution process proposed by a student.

Question: the question shown in the attached image

The student's answer steps are:
{numbered solution steps}

Execute the following instructions sequentially.
```

```
1. First extract the mathematical concepts employed in each actual step. Below is the
     correct mathematical concepts should be applied. Give a detailed comparison between the
      extracted concepts and the correct mathematical concepts.
{Stage-1 concepts}
Finally, for EACH step, output a label from the options (correct-and-aligned, reasonable-but
     -incomplete, nothing-extracted, and contradiction-found) to categorize the use of
     mathematical concepts for that step.

2. First extract the key analyses present in each actual step. Below is the correct analyses
      should be derived. Give a detailed comparison between the actual analyses and the
     correct analyses.
{Stage-1 analyses}
Finally, for EACH step, output a label from the options (correct-and-aligned, reasonable-but
     -incomplete, nothing-extracted, and contradiction-found) to categorize the problem
     solving approach for that step.

3. First extract the calculation results present in each actual step. Below is the correct
     calculation to be done. If any calculation results are extracted from the actual steps,
      list their corresponding results in the correct calculations, and give a detailed
     comparison between the extracted calculation results and correct results.
{Stage-1 calculations}
Finally, for EACH step, output a label from the options (correct-and-aligned, reasonable-but
     -incomplete, nothing-extracted, and contradiction-found) to categorize the correctness
     of calculations for that step.

End your reply with a summary block in exactly this form, one line per step:
LABELS step <n>: <concepts label> | <analyses label> | <calculations label>
```

```
Your solution to the problem in the attached image was reviewed and a mistake was identified
     .

The first incorrect step is (step {n}):
{text of step n}

The reviewer's assessment:
{Stage-2 reply}

Redo the solution. Keep the steps before (step {n}) if they are sound, fix (step {n}), and
     carry on from there to a corrected final answer.

{answer rule}
```

SelfCheck. Two solutions are sampled at temperature 1.0. If their answers agree, the first is graded; otherwise each is checked once, and the sample with the larger weight is graded.

```
Solve the following problem step by step. Please start each step with "Step :" and split
     sentences in each step with "\n\n".

You are taking the {exam} ({subject}) exam. The question is shown in the attached image,
     including its answer choices.

Exactly ONE of the choices shown is correct.

Work through the problem rigorously, showing your reasoning. Then, on the very last line of
     your response, output your final answer in exactly this format, with nothing after it:

Final Answer: <your answer>

Give exactly one choice, in the exact form it is labeled in the image (e.g. "Final Answer: C
     " or "Final Answer: 2"). If none of the listed choices is correct, answer exactly "
     Final Answer: N/A".
```

```
The following is a question and a proposed step-by-step solution.

Question: the question shown in the attached image

Solution:
{numbered solution steps}

Check the correctness of every step above. Please consider the information it relies on and
     check step by step. Please end your response with one line per step, in exactly this
     form:

CHECK step <n>: Correct
CHECK step <n>: Wrong
CHECK step <n>: Not Sure
```

SymCode. The model writes one SymPy script, which is executed. If it raises an error or times out, the model is asked to debug it, up to three times.

```
You are an expert mathematical reasoner. Your output must be ONLY a single Python code block
      fenced as ```python ... ``` with no prose before or after.
Inside that single Python script:
1. Import SymPy with `import sympy as sp`
2. Add explicit step-by-step reasoning as comments throughout your code
3. Document the problem setup:
   - Clearly identify variables, constraints, and goals in comments
   - Define symbols with appropriate assumptions (e.g., sp.symbols('x', positive=True,
     integer=True))
4. Include intermediate reasoning steps:
   - Each step should have a comment explaining the mathematical reasoning
   - Use meaningful variable names that reflect their purpose
   - Show the algebraic manipulations clearly
5. For verification:
   - Substitute solutions back into original equations
   - Check domain constraints (e.g., integer solutions, positive values)
   - Filter invalid solutions
6. Print ONLY the final answer in LaTeX boxed form:
   print(r"\boxed{{{}}}".format(final_answer))
# PROBLEM
The problem is the question shown in the attached image, including its answer choices.
Exactly ONE of the choices shown is correct.
The final answer is the label of the one correct choice, exactly as it is labeled in the
     image (for example C or 2).
# END PROBLEM
```

```
Debug the following code based on the provided error message.

```python
{script}
```

Error message:
```
{error message}
```
```

MathTypes. The solver's system prompt is generated from the exam's registry. Its fixed text is shown below, with the registry-dependent parts in braces; the types and theorems it lists are those of Appendices I and L and their per-exam extensions. The user message carries the page image and the question's answer format. Worked examples are simple boilerplates from the American SAT's public Practice Test 4 [1] problems to ensure zero contamination or overlap with the tested problems.

```
You are a mathematical reasoning agent. You solve math problems by writing step-by-step
     solutions in MathTypes, a typed language for verifiable mathematical reasoning.

## MathTypes Syntax

Each line is a function call with named parameters:
```
type_name(param1: "value1", param2: "value2", claimed_result: "your_answer")
```

Rules:
- Every function call MUST include `claimed_result` -- this is the answer you claim for that
      step.
- All values are strings in double quotes. Use SymPy syntax for math expressions:
  - Powers: `x**2` (not x^2)
  - Fractions: `Rational(1,7)` or `1/7` for numerical fractions
  - Functions: `sqrt(x)`, `sin(x)`, `Abs(x)`
  - Multiplication: `3*x` (explicit asterisk)
- Dict values use Python syntax inside quotes: `"{'x': 5, 'y': 8}"`
- Use `<thought>...</thought>` tags before each step to explain your reasoning in natural
     language. These are comments and do not affect compilation.
- Lines starting with `#` are comments.
- Start with `@problem_id` and `@source` directives.
{rules for the registry's terminal answer types}

## Available Types
```

[1] https://satsuite.collegeboard.org/media/pdf/sat-practice-test-4-digital.pdf

```
{type reference, generated from the exam's registry}

## Known Theorems (for use with apply_theorem)

{theorem reference, generated from the exam's registry}

## When to Use Each Type

{usage guidance for the registry's types}

## Important

- Break your solution into atomic steps. One operation per line.
- Do NOT combine multiple operations. "Substitute and simplify" is TWO steps.
- Your `claimed_result` is verified by a symbolic math engine -- but it can only check the
    steps you actually write. Reasoning you leave in comments or do in your head is never
    checked, so a wrong answer reached that way will pass silently. Put the work in the
    steps.
- Use `informal_step` ONLY for genuinely unverifiable reasoning (geometric intuition, visual
     extraction). Do NOT use it for arithmetic or algebra.

## Examples

{worked .mt examples}

## Your Task

Solve the given {exam name} problem by writing MathTypes code. Output ONLY the .mt code --
    no explanations before or after. Start with @directives, then solve step by step.
```

Each correction pass sends the program and the compiler's output:

```
Your MathTypes code had verification errors. Here is your code and the compiler output:

## Your Code
```
{program}
```

## Compiler Output
```
{compiler and verifier output}
```

Fix the errors in your MathTypes code. Output ONLY the corrected .mt code -- no explanations
    .
```

Controls (Section 4.3). C1 sends the CoT prompt as the user message, with the exam's type and theorem references as a system message:

```
Below is a reference sheet of mathematical operations and theorems that are useful for
    problems from this exam. You may use any of them in your reasoning.

## Operations

{type reference}

## Known Theorems

{theorem reference}
```

C2 runs MathTypes unchanged, except that after a verifier rejection the correction pass sends:

```
Your MathTypes code contains at least one error. Here is your code:

## Your Code
```
{program}
```

Re-check every step of your solution and fix any errors. Output ONLY the corrected .mt code
    -- no explanations.
```