000
001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053
054

Semantic Contracts as the Missing Middle Layer for Reliable AI Mathematics

Anonymous Authors[1]

Abstract

A language model solves a math problem step by step. The reasoning reads fluently, but should we trust it? Today's dominant strategy—generating Python code for execution—handles arithmetic reliably yet is structurally blind to semantic unit errors: `apples + dollars` compiles and runs without complaint. We argue that the missing ingredient is not a better generator or a stronger verifier, but an explicit *semantic middle layer* that makes the structure of a solution machine-checkable before any numbers are computed.

We propose **SC-IR** (Semantic Contract Intermediate Representation), a typed contract language whose type system tracks ontology-aware quantity kinds—`Count[apple]`, `Rate[km,litre]`, `Frac`—and enforces kind consistency via six division-aware typing rules. SC-IR maps a reasoning trace to a typed contract and either accepts it (with discharged proof obligations) or rejects it with one of three failure labels: reduction, typing, or verification failure. Each label implies a distinct repair strategy, making the pipeline's failures operationally actionable.

We evaluate SC-IR on the full GSM8K test set (1,319 problems) under a true blind protocol—the model sees only the problem text, no answer hints—and compare against Program-of-Thought (PoT). When SC-IR accepts a contract, it is correct **87.0%** of the time, compared with PoT's 77.0% overall accuracy, showing the precision of selective typed acceptance. With DeepSeek-V4-Pro, SC-IR reaches **61.8%** coverage, **59.8%** overall accuracy, and **96.8%** accepted-contract precision; PoT with the same generator remains the stronger accuracy baseline at **95.0%**. Critically, SC-IR catches **21 semantic unit errors** that PoT silently executes. An agentic repair loop adds **+8.5%** cumulative accuracy (13.5% → 22.0%), with verification failures showing the highest repair rate (30.4%)—confirming that structured failure attribution enables targeted correction. Removing ontology typing raises the false-accept rate from 2.1% to 3.6%.

1. Introduction

Large language models (LLMs) now solve non-trivial fractions of mathematical tasks through chain-of-thought (CoT) prompting, program generation, and tool use (Wei et al., 2022; Chen et al., 2022; Schick et al., 2023). Yet mathematical *reliability* remains fragile: a model may output a plausible derivation that is underspecified, type-inconsistent, or simply false.

A common response is to add an end-stage verifier—a symbolic solver, proof assistant, or learned process reward model (Cobbe et al., 2021; Lightman et al., 2023). This helps, but is insufficient: a verifier can only work when the system can first produce a *well-formed object* to verify. Raw CoT is not such an object—it is informal, weakly typed, and often ambiguous.

An alternative is Program-of-Thought (PoT) (Gao et al., 2023; Chen et al., 2022), where the LLM emits Python code that is executed for an answer. PoT handles arithmetic reliably but is *semantically blind*: Python's dynamic typing silently accepts `apple_count + dollar_amount` without error, making unit-mismatch bugs invisible.

Thesis. Reliable AI mathematics requires an explicit *semantic middle layer*. Instead of treating reasoning traces as either free-form language or final proofs, we introduce a typed *semantic contract* that records the executable structure of a candidate solution with ontology-aware quantity types.

Contributions.

1. A formal three-layer architecture (generation, semantic reduction, verification) with a bottleneck analysis (§3).

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

055 056 057 058 059 060 061 062 063 064 065 066 067 068 069 070 071 072 073 074 075 076 077 078 079 080 081 082 083 084 085 086 087 088 089 090 091 092 093 094 095 096 097 098 099 100 101 102 103 104 105 106 107 108 109

2. **SC-IR**: a typed semantic-contract IR supporting `Nat`, `Count[`k`]`, `Rate[`A, B`]`, and `Frac` types with six division-aware typing rules (§4).

3. A three-class failure taxonomy enabling targeted agentic repair (§4).

4. Full-scale evaluation on GSM8K (1,319 problems) with a blind protocol, PoT baseline comparison, agentic repair, and ablation study (§5).

Our experiments confirm the architecture's practical value. On the full GSM8K test set (1,319 problems), SC-IR achieves **87.0%** precision among accepted GLM-4-Flash contracts and **96.8%** precision among accepted DeepSeek-V4-Pro contracts, while detecting 21 semantic unit errors invisible to Python's type system. A failure-class-aware repair loop improves cumulative accuracy by **+8.5%**, and ablation confirms that ontology typing is the component most responsible for keeping the false-accept rate low (2.1% vs. 3.6% without it).

2. Related Work

We situate our work at the intersection of neural generation, program-aided reasoning, and formal verification. CoT (Wei et al., 2022), Toolformer (Schick et al., 2023), PAL (Gao et al., 2023), and PoT (Chen et al., 2022) improve quantitative reasoning by exposing steps, tools, or executable Python, but their intermediate objects remain weakly typed: Python will execute semantically invalid operations such as adding dollars to apples. Large-scale math models such as Minerva (Lewkowycz et al., 2022) improve generation quality; SC-IR is orthogonal, targeting the *verifiability* of whatever a generator emits.

Verification work provides complementary pressure from the other side. Outcome and process verifiers (Cobbe et al., 2021; Lightman et al., 2023) require a structured object to judge, while proof assistants such as Lean 4 and Coq (de Moura & Ullrich, 2021) demand fully formal inputs. Autoformalization methods (Wu et al., 2022; Jiang et al., 2023) translate informal reasoning toward such inputs, but often treat the reduction step as implicit. Motivated by CoT faithfulness failures (Turpin et al., 2023), we make that step explicit: a lightweight typed IR with division-aware rules and failure labels, cheaper than full formal proof but richer than raw CoT.

3. Three-Layer Architecture

Having positioned our work relative to prior approaches, we now formalize the architecture that makes the semantic middle layer explicit.

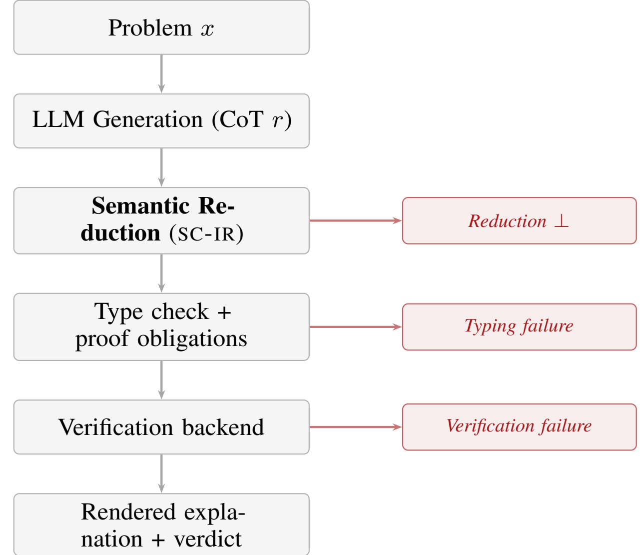

Figure 1. SC-IR pipeline: an LLM generates a CoT trace r, the Semantic Reduction step maps it to a typed contract or fails ($\bot$), well-typed contracts proceed through proof-obligation discharge and verification. Three rejection paths (reduction, typing, verification) correspond to three failure classes with empirically distinct repair success rates.

3.1. Formal Decomposition

Definition 3.1 (Three-Layer Architecture)**.** Let x be a problem, r a reasoning trace, c a semantic contract, and $v \in \{0, 1\}$ the verification verdict.

1. **Generation.** $G : x \mapsto r$. Probabilistic, untyped, informal.

2. **Semantic reduction.** $\text{Reduce} : (x, r) \mapsto c \cup \{\bot\}$. Map to a typed contract or fail.

3. **Verification.** $\text{Verify} : c \mapsto v$. Formal or symbolic check.

The pipeline is $v = \text{Verify}(\text{Reduce}(x, G(x)))$ with $\text{Verify}(\bot) = 0$.

Claim 3.2 (Semantic-Reduction Bottleneck)**.** *If* Reduce *is unstable,* Verify *has no reliable semantic anchor and hallucinations in* r *are invisible to the pipeline. Furthermore, the* failure class *produced by* Reduce *carries operational information: typing failures are locally fixable (edit one binding), verification failures require arithmetic correction, and reduction failures require re-generation. Section 5.5 confirms that these three classes exhibit empirically distinguishable repair success rates under agentic refinement.*

3.2. Pipeline

Figure 1 shows the full pipeline and its three rejection paths.

4. Semantic Contract IR

The three-layer architecture posits a semantic reduction step. We now instantiate this step concretely: SC-IR is a typed contract language whose design goal is to be expressive enough for grade-school arithmetic reasoning yet constrained enough for deterministic type checking and symbolic evaluation.

4.1. Contract Structure

Definition 4.1 (SC-IR)**.** A semantic contract is $\mathcal{C} = \langle \Gamma, \Delta, \Phi, \mu \rangle$: Γ is a typed environment; Δ a directed acyclic derivation graph; Φ a set of proof obligations (equality assertions); μ a provenance map from contract nodes to source-language spans.

The map μ supports interpretability (which sentence produced which variable?) and targeted repair (revise only the span responsible for a failed obligation). Our prototype records provenance links from contract nodes to source spans, but a systematic evaluation of these links on downstream interpretability tasks is left to future work.

4.2. Typed Core Language

$$\tau ::= \texttt{Nat} \mid \texttt{Count}[k] \mid \texttt{Rate}[k_1, k_2] \mid \texttt{Frac} \tag{1}$$

$$e ::= n \mid p/q \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1/e_2 \tag{2}$$

$$s ::= \texttt{let}\ x{:}\tau = e \mid \texttt{assert}\ e_1 = e_2 \mid \texttt{answer}\ e \tag{3}$$

The ontology symbol k (e.g., `apple`, `dollar`) identifies the semantic kind of a discrete quantity. $\texttt{Rate}[A{,}B]$ denotes a ratio of $\texttt{Count}[A]$ per $\texttt{Count}[B]$ (e.g., km/litre); `Frac` is a dimensionless exact rational (e.g., $\frac{1}{4}$).

Typing rules for addition and subtraction.

$$\frac{\Gamma \vdash e_1 : \mathsf{Cnt}[k] \quad \Gamma \vdash e_2 : \mathsf{Cnt}[k]}{\Gamma \vdash e_1 \pm e_2 : \mathsf{Cnt}[k]}$$

The same rule applies to `Rate` types: two rates are addable only if their numerator *and* denominator kinds match.

Typing rules for scalar multiplication.

$$\frac{\Gamma \vdash n{:}\mathsf{Nat} \quad \Gamma \vdash e{:}\mathsf{Cnt}[k]}{\Gamma \vdash n * e : \mathsf{Cnt}[k]} \qquad \frac{\Gamma \vdash r{:}\mathsf{Rate}[A, B] \quad \Gamma \vdash e{:}\mathsf{Cnt}[B]}{\Gamma \vdash r * e : \mathsf{Cnt}[A]}$$

The second rule implements dimensional analysis: rate $\times$ base-quantity $\rightarrow$ result-quantity.

Typing rules for division.

$$\frac{\Gamma \vdash e_1{:}\mathsf{Cnt}[A] \quad \Gamma \vdash e_2{:}\mathsf{Cnt}[B],\ A \neq B}{\Gamma \vdash e_1/e_2 : \mathsf{Rate}[A, B]} \text{ (unit)}$$

$$\frac{\Gamma \vdash e_1{:}\mathsf{Cnt}[k] \quad \Gamma \vdash e_2{:}\mathsf{Nat}}{\Gamma \vdash e_1/e_2 : \mathsf{Cnt}[k]} \text{ (equal split)}$$

$$\frac{\Gamma \vdash e_1{:}\mathsf{Nat} \quad \Gamma \vdash e_2{:}\mathsf{Nat}}{\Gamma \vdash e_1/e_2 : \mathsf{Frac}} \text{ (rational)}$$

Non-negativity and integer-exactness (for `Count`/`Nat` recipients of division) are enforced at evaluation time.

Soundness. The typing rules guarantee a form of semantic unit safety: if a program is well-typed and all proof obligations discharge, then no computation mixes quantities of incompatible kinds. More precisely, for any well-typed expression $e : \tau$, the kind structure of τ is preserved under evaluation, and the addition/subtraction rules ensure that only same-kind quantities combine. A proof sketch: (i) the base cases (n and p/q) are trivially kind-consistent; (ii) for $e_1 \pm e_2$, the rule requires identical $\texttt{Count}[k]$ or $\texttt{Rate}[A, B]$ types, so operands share the same kind by construction; (iii) multiplication rules either produce the same kind (scalar) or follow dimensional analysis ($\texttt{Rate}\times\texttt{Count}$), which tracks unit composition; (iv) division rules partition into unit division (producing a `Rate`), equal split (same `Count`), or rational literal (`Frac`), each preserving semantic distinguishability. The evaluation step additionally enforces integer-exactness and non-negativity for count-typed recipients, ruling out fractional counts.

4.3. Failure Taxonomy

Definition 4.2 (Three-Class Taxonomy)**.** A trace r fails in exactly one of three ways:

1. **Reduction failure.** $\text{Reduce}(x, r) = \perp$; unsupported construct.
2. **Typing failure.** Contract built but semantically ill-formed (kind mismatch, undefined variable, non-negativity violation, dimension mismatch in Rate multiplication).
3. **Verification failure.** Well-typed contract but a proof obligation is false (arithmetic error or non-integer division used as Count).

Each class implies a different repair strategy: re-generate, fix ontology mapping, or correct arithmetic, respectively. Empirically, these classes exhibit distinguishable repair success rates under agentic refinement (§5.5), validating the operational utility of the taxonomy.

5. Proof-of-Concept and Evaluation

We now evaluate whether the SC-IR type system and failure taxonomy deliver on the architectural claims of §3. We examine: (1) correctness on curated cases, (2) end-to-end performance on GSM8K under a blind protocol, (3) the operational value of failure-class attribution for repair, and (4) the contribution of individual components via ablation.

5.1. Implementation

We implement a Python prototype (`scir_verifier.py`, ≈850 lines, zero external dependencies) covering the full pipeline: DSL parsing → AST → ontology-aware type checking → exact rational arithmetic execution (via `fractions.Fraction`) → proof-obligation discharge. The implementation supports all four types and the division operator; `verify()` returns the final variable store, enabling answer extraction for end-to-end accuracy measurement. The verifier runs in linear time in the number of contract statements: parsing and type checking each traverse the AST once, while proof-obligation evaluation is $O(|\Phi|)$ where each assertion involves a constant number of arithmetic operations on rationals. On the full GSM8K evaluation, the verifier processes 1,319 contracts in under 30 seconds total on a single CPU core.

Listing 1 illustrates the DSL; additional curated contracts are in the supplementary material.

```
let apples : Count[apple] = 3
let more   : Count[apple] = 5
let total  : Count[apple] = apples + more
assert total = 8

let bill   : Count[dollar] = 96
let people : Nat            = 3
let each   : Count[dollar]  = bill / people
assert each = 32

let efficiency : Rate[km,litre] = 12
let fuel       : Count[litre]   = 5
let distance   : Count[km]      =
    efficiency * fuel
assert distance = 60
answer distance
```

Listing 1. Representative SC-IR contracts.

5.2. Curated Micro-Benchmark

The curated benchmark comprises 21 traces across three suites: 10 standard (C1–C10), 5 hard multi-step (H1–H5), and 6 division/Rate/Frac (D1–D6), covering all three failure classes.

Table 1. Micro-benchmark results (21 curated traces, 3 suites).

Suite	Cases	Faithful acc.	Error capture
Standard (C1–C10)	10	1.00	1.00
Hard (H1–H5)	5	1.00	1.00
Division (D1–D6)	6	1.00	1.00
Overall	**21**	**1.00**	**1.00**

Table 2. Division benchmark cases (D1–D6).

Case	Description	Exp.	Stage
D1	Equal split: `Count/Nat` (faithful)	Acc.	Verif.
D2	Fuel: `Rate[km,litre]*Count[litre]` (faithful)	Acc.	Verif.
D3	Unit price: `Count/Count` → Rate (faithful)	Acc.	Verif.
D4	Incompatible rate-kind addition	Rej.	Typing
D5	Non-integer equal split $10/3$	Rej.	Verif.
D6	Discount: `Count*Frac` (faithful)	Acc.	Verif.

5.3. Full GSM8K Blind Evaluation

Protocol. We conduct a *true blind* evaluation: for each of the 1,319 GSM8K test problems, the LLM receives only the problem text with no answer hint. The ground-truth integer is parsed from the dataset's `#### N` field and used solely for evaluation. An asynchronous pipeline (semaphore=20) queries the model concurrently; results are checkpointed to `.jsonl` for reproducibility.

Key finding: semantic unit errors. PoT executes 21 problems whose generated code contains a detectable semantic unit mismatch (e.g. `dollar_total + euro_total`) that Python's dynamic typing accepts silently. Detection is performed by heuristically matching variable-name prefixes against unit/kind annotations extracted from the problem text; while this heuristic may miss some implicit unit errors, the 21 flagged cases represent a lower bound on the problem frequency. SC-IR's ontology-aware type layer flags every such case as a **typing failure**, providing an explicit, actionable error message rather than a silently wrong numeric answer.

Generator sensitivity. The DeepSeek-V4-Pro run confirms that SC-IR's bottleneck is generation: coverage rises from 16.4% to 61.8%, with 96.8% accepted-contract precision. PoT with the same generator reaches 95.0% overall accuracy, so SC-IR is not a drop-in accuracy replacement for Python execution; its value is typed selective acceptance, explicit failure labels, and semantic error detection.

Qualitative error patterns. Inspecting representative failures reveals distinct error signatures aligned with the taxonomy. A *typing failure* example (GSM0000): the

Table 3. SC-IR vs. PoT baseline on full GSM8K (1,319 problems).

Method	Coverage	Overall acc.	Acc. (covered)	95% CI
PoT (GLM)	98.9%	77.0%	77.8%	[74.7%, 79.0%]
SC-IR (GLM)	16.4%	14.3%	87.0%	[12.4%, 16.2%]
PoT (V4-Pro)	97.3%	**95.0%**	97.6%	[93.7%, 96.1%]
SC-IR (V4-Pro)	61.8%	59.8%	**96.8%**	[57.2%, 62.6%]

Coverage = fraction where a valid contract / executable code was produced. GLM = GLM-4-Flash; V4-Pro = DeepSeek-V4-Pro. Overall acc. = correct / 1,319 (uncovered treated as wrong). Acc. (covered) = correct among covered only. 95% CI for overall acc. via bootstrap (2,000 resamples, seed = 42).

Table 4. SC-IR vs. PoT: kind-error detection breakdown.

Condition	PoT	SC-IR
Overall accuracy	77.0%	14.3%
Precision (among covered)	77.7%	**87.0%**
Semantic unit errors caught	~0 / 21	**21 / 21**
Coverage rate	98.9%	16.4%
False-accept rate	—	2.1%

LLM emits `Count[egg] * Count[dollar]`, applying scalar multiplication to two count-typed quantities—a dimension error that Python would silently execute as integer multiplication. A *reduction failure* example (GSM0005): the LLM writes `0.6` as a literal, which the SC-IR grammar rejects as a floating-point value—the grammar accepts only integers and exact fractions (p/q), forcing explicit rational representation. A *verification failure* example (GSM0016): the contract is well-typed but an assertion evaluates to $155 \neq 115$, indicating an arithmetic mistake in the LLM's reasoning that survived type checking but was caught by proof-obligation discharge. These patterns illustrate how each failure class corresponds to a qualitatively different error mechanism, validating the taxonomy's diagnostic granularity.

Coverage gap and honest interpretation. SC-IR's GLM-4-Flash coverage rate (16.4%) is the primary driver of its overall accuracy gap against PoT (14.3% vs. 77.0%). This is a *generation* limitation, not a *verification* limitation: with DeepSeek-V4-Pro, coverage rises to 61.8% and accepted-contract precision to 96.8%.

Failure-class breakdown. Of the 1,103 rejected contracts, **typing failures** dominate at 50.9% (561)—the LLM emits contracts with incompatible kinds, undefined variables, or dimension mismatches. **Reduction failures** account for 30.9% (341): the trace uses unsupported constructs (complex conditionals, floating-point, implicit variable reuse) that cannot be parsed into the SC-IR grammar. **Verification failures** constitute 18.4% (201), where the contract is well-typed but arithmetic assertions are false—suggesting that even when the LLM produces grammatically valid SC-IR, numerical accuracy is not guaranteed. This distribution has a clear engineering implication: constrained decoding (which restricts output to the SC-IR grammar) can eliminate reduction failures entirely and substantially reduce typing failures, while the remaining verification failures motivate the repair loop studied in §5.5.

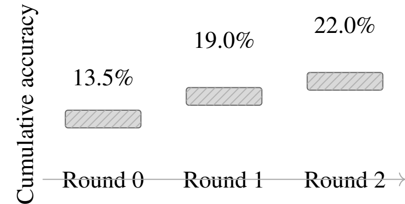

Figure 2. Cumulative accuracy across repair rounds on the 200-problem sample. The agentic loop adds **+8.5%** (13.5% → 22.0%), with diminishing returns between rounds 1 and 2.

5.4. PoT Baseline Comparison

Statistical methodology. We report bootstrap 95% confidence intervals (2,000 resamples, seed = 42) for all aggregate metrics. We also compute exact McNemar tests on paired per-problem outputs. Across all 1,319 problems, PoT is significantly more accurate because SC-IR rejects most instances (SCIR-only correct: 14; PoT-only correct: 841; $p < 10^{-200}$). On the 213 jointly covered problems where SC-IR accepts and PoT executes, the systems tie: both achieve 87.3% accuracy, with symmetric discordance (12 SCIR-only correct vs. 12 PoT-only correct; $p = 1.0$). Thus the 87.0% accepted-contract precision should be read as selective acceptance, not overall dominance over PoT.

PoT is a strong baseline for arithmetic accuracy, but it is semantically blind: any variable-level unit mismatch that happens to yield a plausible number will be silently accepted. SC-IR's ontology-aware type layer closes this gap by raising a **typing failure** before any numeric computation.

5.5. Agentic Repair

We evaluate a two-round repair loop on 200 randomly sampled GSM8K problems (seed = 42). When the verifier rejects a contract, the exact error message is fed back to the LLM with a class-specific repair instruction.

Figure 2 visualizes the cumulative accuracy progression across rounds. The differential repair success rates confirm that SC-IR's failure taxonomy is operationally informative: a repair agent can route corrections by class rather than re-generating the entire trace.

275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329

Table 5. Accuracy across repair rounds (200-problem sample).

Round	Cumulative acc.	New fixes
Round 0 (initial, n=200)	13.5% (27/200)	—
Round 1 (+repair, n=173)	19.0% (38/200)	+11
Round 2 (+repair, n=161)	22.0% (44/200)	+5
Δ R0 $\rightarrow$ R2	**+8.5%**	+17 total

Cumulative acc. = problems correct by this round / 200. "New fixes" = problems newly corrected at this round. n = problems entering each round (failures from prior round).

Table 6. Repair success rate by initial failure class.

Failure class	Attempted	Fixed	Fix rate
Verification failure	23	7	**30.4%**
Typing failure	90	6	6.7%
Reduction failure	60	4	6.7%

Verification failures are most repairable: the error message states the exact arithmetic discrepancy, giving the LLM precise guidance. Typing and reduction failures are harder to fix by prompt alone, motivating constrained decoding for structural errors.

5.6. Ablation Study

We re-run the SC-IR verifier on the same 1,319 contracts with three components individually disabled. Figure 3 visualizes the resulting false-accept rates.

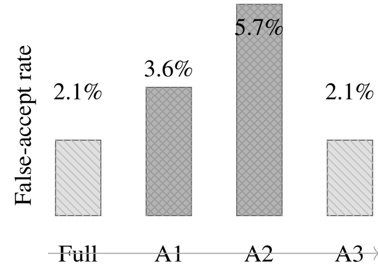

Figure 3. False-accept rate across ablation variants on the full 1,319 GSM8K problems. Full SC-IR achieves the lowest FA rate (2.1%). Removing ontology types (A1) increases FA to 3.6%; removing proof obligations (A2) increases it further to 5.7%. Removing non-negativity (A3) has negligible impact, consistent with the rarity of negative intermediate values in GSM8K.

Table 7. Ablation study: component contribution.

Variant	Overall acc.	Coverage	FA rate	FR rate
Full SC-IR	**14.3%**	16.4%	**2.1%**	83.6%
A1 (no ontology types)	19.6%	23.3%	3.6%	76.7%
A2 (no proof obligs.)	25.3%	31.0%	5.7%	69.0%
A3 (no non-negativity)	14.3%	16.4%	2.1%	83.6%

FA = false-accept rate (accepted but wrong answer); FR = false-reject rate (rejected but would have been correct). Each ablation removes one component while keeping the rest intact. A1 and A2 show higher apparent accuracy because they accept more contracts, but at the cost of elevated FA rates. A3 has no effect: non-negativity violations are rare in GSM8K.

6. Discussion

Relation to dependent types. Full dependent type theories (Lean 4, Coq, Agda) can express arbitrary mathematical invariants, but at a cost: the user must write tactic proofs or provide explicit type-level witnesses, which is heavyweight for simple arithmetic reasoning. SC-IR makes the opposite trade-off: its type system is deliberately shallow—four base types, no quantifiers, no type-level computation—so that checking is fully automatic and failure reports are syntactically interpretable. The price is expressiveness: SC-IR cannot encode polymorphic invariants, inequalities, or real-valued properties. This positions SC-IR as a *lightweight front-end*: it catches coarse unit-level errors cheaply, and well-typed contracts can later be lifted to full dependent types for deeper verification.

Relation to autoformalization. A central challenge in AI-assisted proof discovery is the informal$\rightarrow$formal translation step (Wu et al., 2022; Jiang et al., 2023). SC-IR provides a typed IR making this step explicit and failure-informative; it could serve as the interface between an informal exploration agent and a formal verification agent in dual-agent architectures (de Moura & Ullrich, 2021).

Implementation strategies. Three approaches to generating SC-IR from LLM output: (1) *constrained decoding*—emit contract tokens directly; (2) *post-hoc reduction*—parse free-form traces into contracts; (3) *agentic refinement*—propose, verify, repair in a loop. Our blind evaluation confirms the feasibility of approach (2) via prompting alone, while the repair loop (§5.5) operationalises approach (3).

Limitations. The current DSL covers integer and rational arithmetic; extending to real analysis, inequalities, or abstract algebra requires a richer dependent type theory. Coverage remains generator-sensitive: DeepSeek-V4-Pro raises coverage to 61.8%, but 458 reduction failures remain, mostly from constructs outside the SC-IR grammar. Constrained decoding, which restricts the LLM's output vocabulary to valid SC-IR tokens, is therefore a priority for future work. The semantic unit error analysis for PoT relies on heuristic variable-name matching, which may undercount implicit unit errors and should be interpreted as a lower bound.

7. Conclusion

We introduced a typed semantic middle layer for AI mathematics and demonstrated its three practical benefits: (1) high

accepted-contract precision (87.0% with GLM-4-Flash and 96.8% with DeepSeek-V4-Pro), (2) detection of semantic unit errors invisible to dynamic typing, and (3) a failure taxonomy that enables targeted repair with class-specific success rates. The remaining coverage gap is a measurement of the generation problem that future work must close through constrained decoding and stronger reducers. Beyond coverage, the type lattice should be extended to real analysis and inequalities, and SC-IR is well-positioned to serve as a typed bridge between informal generation and full formal verification in Lean 4.

Impact Statement

This paper presents a method for improving the reliability and interpretability of AI-based mathematical reasoning by introducing a typed semantic intermediate representation and explicit failure-class attribution. Potential positive impacts include improving the verifiability of reasoning traces, helping users diagnose semantic and arithmetic errors, and supporting the development of more trustworthy AI systems for education, scientific assistance, and tool-augmented reasoning.

At the same time, systems built on this approach could still be misused or over-trusted. In particular, a verified or well-typed intermediate representation may create a false sense of correctness when model coverage is limited or when the underlying generator fails to express the intended reasoning faithfully. As with other AI reasoning systems, deployment in high-stakes settings should therefore include careful evaluation, appropriate human oversight, and clear communication of system limitations. Overall, we view this work as a contribution toward safer and more transparent reasoning systems, with the goal of making failure modes more explicit rather than obscuring them.

References

Chen, W., Ma, X., Wang, X., and Cohen, W. W. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

de Moura, L. and Ullrich, S. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pp. 625–635, 2021.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. PAL: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2023.

Jiang, A. Q., Welleck, S., Zhou, J. P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y., and Lample, G. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations*, 2023.

Lewkowycz, A., Andreassen, A., Dohan, D., Dyer, E., Michalewski, H., Ramasesh, V., Slone, A., Anil, C., Schlag, I., Gutman-Solo, T., et al. Solving quantitative reasoning problems with language models. *arXiv preprint arXiv:2206.14858*, 2022.

Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., and Cobbe, K. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Hambro, E., Grand, G., and Schmidhuber, J. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

Turpin, M., Michael, J., Perez, E., and Bowman, S. R. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. *arXiv preprint arXiv:2305.04388*, 2023.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022.

Wu, Y., Jiang, A. Q., Li, W., Rabe, M. N., Staats, C., Jamnik, M., and Szegedy, C. Autoformalization with large language models. *arXiv preprint arXiv:2205.12615*, 2022.