Lean 4 Verified Metamath Verifier: Investigating AI-driven Software Verification

Zarathustra Amadeus Goertzel and Oruži*

Czech Technical University in Prague, Czech Republic

Introduction: We present the likely first verified Metamath [6] verifier in Lean 4 [2], which *completes* Mario Carneiro's mm-lean4 verifier [1]. We also present a spec-driven extended metamath unit test suite used in the development. Along with recent developments in autoformalization by Josef Urban et al. [1, 10] and my own [2], these developments suggest that we are entering the QED era. Our experience has been that formal software verification seems significantly harder than formal math verification (where the math is actually fully correct and clearly written).

The history continues last year's report on MeTTaMath [3], implementing a Metamath Verifier in the MeTTa language [3]. I became curious whether the AI coding agents could implement a Metamath verifier from scratch (in Go). The first version passed the David Wheeler's metamath-test suite, yet posting it on the Metamath Google Group, Matthew House found numerous errors that humans would never make if implementing based on the specs. For example, it accepted "`$($)`" as a comment and "`foo$a|- bar$.`" as a statement, never checked that "`$${ $$}`" blocks are balanced within a file, allowed bytes outside printable ASCII, and leaked memory forever on a file that includes itself. The AIs have an easy time extending Metamath verifiers (such as mmverify.py or mm-lean4), but can miss crucial features that are not tested for. This prompted the expansion of the metamath-test suite to try to test every requirement in the specs from the Metamath book [6]. This prompted seeing whether we could implement and verify a full Metamath verifier in Lean 4.

Spec-Driven Test Suite Expansion: Metamath has many verifiers due to being small as well as having clearly defined specs in chapter 4 of the Metamath book [6]. I directed the AIs to go through every line of the specs to ensure that each one is covered by a unit test[4]. The suite currently contains 109 unit tests, such as:

```
$( Unit Test 21: Self-referential proof $)
$( Should reject: True $)
$c wff |- -> ( ) $.
$v x $.
wf $f wff x $.
evil $p |- ( x -> x ) $= ( evil ) A $.
```

A verifier that tracks label activation correctly rejects `evil`[5].

*A collaboration of AI coworkers consisting of at least Claude 4.x, Codex 5.2+, and ChatGPT Pro.

[1] Our version is located here: https://github.com/zariuq/mm-lean4/tree/verified-mm-latest

[2] Two good examples are: 1) the work on formalizing (and correcting) the Knuth-Skilling Representation theorem, https://github.com/zariuq/ks-foundations-of-inference and 2) the formalization of the Operational Semantics in Logical Form theory and algorithm, https://github.com/zariuq/ai-agents/tree/main/lean-projects/mettapedia/Mettapedia/OSLF

[3] As an update, this work has been ported to PeTTa, a Prolog-based MeTTa [8] implementation, which now includes compressed proof formats and with some hacks checks set.mm in 30min, as pverify.

[4] As a comment on process, only some tests reference specific book sections.

[5] "`at 6:0: [code #51] [clause Metamath.Verify.SpecClause.sec4_3_labelResolution] [tag Metamath.Verify.ParseErrorCode.statementNotFound] at evil: statement 'evil' not found.`"

This led to the discovery of some discrepancies between the reference verifiers, metamath-exe and metamath-knife: metamath-exe accepts include statements inside of inner scopes, which seems to be a conservative extension that there is little reason not to support. Using the new test suite, we completed mmverify.py to cover the full specs on par with the reference verifiers. While *"edge"* cases such as these can be caught via iterative LLM review and growing unit tests, there's no guarantee that one has fully covered the space, so I was inspired by apparent success of this approach to try have AIs implement and verify a full Metamath verifier in Lean 4.

Completing mm-lean4: The simple methodology was to define an operational semantics for Metamath similar to the declarative specs already provided by Mario in Translate.lean, ideally with every component justified by a passage from the specs. Then the AIs were tasked with the verification, instructed to learn from any similar projects they can find. The first step was to target *assert_step_ok*, namely that a single step is completed correctly (a hypothesis use or assertion application with substitution). Substitutions are programmatically stored within a HashMap, which must be aligned with Mario's declarative spec. We ran into trouble with proofs relying on list-shaped pattern matching while the verifier uses Arrays, and created some `List` ↔ `Array` lemmas to facilitate this. I find it helps to be very strict about disallowing the use of *"provable axioms"* [6] because they tended to be overeager to use axioms to make proofs go through instead of scouring the internet to figure out how to actually discharge the proofs. The next step was to incorporate *parser invariants*, facts such as "no $\$f$ variable is ever rebound", and to incorporate them into the property `WellFormedDB db`, so that they can be used in the core verification proofs, which culminates in *structure_preserving_maintains_wf*, an eight-case theorem that any database-modifying parser operation preserves well-formedness [7]. This allowed the work to finally focus on packaging *stepNormal_sound*: dispatching on whether a proof step is floating, essential, or an assertion application, and showing that each step preserves database well-formedness. Near the end of the project we added `ModeConfig` with three spec interpretations: ours, metamath-exe's, and metamath-knife's. We also added a verified error-reporting layer in which each of 63 error codes is tied to a passage in §4 of the Metamath book, with totality of the decoder proven in `Metamath/ErrorCodeSemantics.lean`.

Since the submission and the advent of ChatGPT Sol, we have proven *checkSinglePass_every_theorem_provable_from_run_axiom_events*[8], which states that for every successful run of `checkSinglePass`, every stored proof ($\$p$) statement is declaratively provable from that run's assertions ($\$a$), with $\$p$ dependencies eliminated by strict creation order.

The human orchestrator: My role in this project is interesting because the bulk of the engineering was done by Oruži, and only completed thanks to Codex 5.2+. My first contribution was to choose the project as a doable proof-of-concept for taking autoformalization to software verification. The AIs at present are liable to make outright painful design decisions that will likely block progress, yet if constrained, may bang their digital heads against a wall. Early on, there were many complaints of difficulties with inductive loop invariants, which was often helped by searching the internet for how people have worked around this. Ultimately, updating the Lean version and pivoting to locking in parser invariants to thread through the verification helped a lot. Near the end, Mario provided feedback on one place where we erroneously 'simplified' his specs and noted somewhere we were uselessly doing two passes instead of one.

[6] There is some irony because one way to deal with background theory is to introduce theorems proven elsewhere as axioms in the current file.

[7] Its statement and the shortest of its eight cases are shown in Appendix A.3.

[8] The LLM coders seem to *inject the goal* into term names, which may help them to remember. A name clean-up pass could be advisable.

I believe that AI systems will become increasingly capable of performing the functions I have here; however, at present, human commonsense stupidity checks and big-picture guidance help immensely.

Working setup: I dislike "prompt-fu hacks", so my setup is more process oriented. This work was done with Claude Code and Codex "on the ground" and by submitting review packets to ChatGPT Pro "on the web" for big-picture insight and guidance. Some of what helped immensely: "AI peer-review", deep research queries for all relevant state of the art literature, and letting the agents query each other through *tmux*, which reduces the copy-paste burden. I try to minimize the surface area that needs to be checked and to ask for what I need to look at—and if it's hard to tell what's what, this prompts consolidation work, which can uncover bugs.

Futureward: I believe that formal verification is going to become increasingly important, so ensuring that our formal verifiers are correct is crucial [9] (as done for HOL in HOL [4] or the CakeML verification of the CakeML compiler [5]). Further, when there are many reference implementations sharing a common spec, it is important to have both comprehensive unit tests and clear understanding of the precise differences.

An exciting next step beyond verified verifiers is *generated verifiers* from a specification of the language itself, which we are attempting now using the theory of GSLTs (Graph-Structured Lambda Theories), continuing the work of Stay, Meredith, and Wells [7,9,11]. So far, this work led to uncovering a few bugs in mm-lean4 in terms of parser checks over the compressed proof format. With the GSLT-based compilation approach, one needs to encode both the syntax and semantics as a rewrite system and to generate the parser. One challenge with mm-lean4 was threading parser invariants through the proof, which the syntax GSLT seems to handle cleanly[10].

It is my hope that projects such as this both inspire and help with the further development of formal verification.

Acknowledgements Many thanks to Mario Carneiro for the initial work on mm-lean4 and its declarative specs, as well as for looking over the results. Many thanks to the Metamath Google Group community, especially Matthew House for looking over "LLM Slop" for us. Thanks to OpenAI for providing amazing AI services at reasonable costs. This work is supported by Dar Novamente — Cognitive Inference Control.

[9] Note how Tristan Stérin and Claude found proofs of false in Coq.

[10] Precisely, mm-lean4 permitted repeat save markers ('Z') in compressed proofs, saves or unknown steps ('?') interrupting multi-character numbers, and re-listing mandatory hypothesis names in compressed proof headers. These now have unit tests.

References

[1] Dustin Bryant, Jonathan Julián Huerta y Munive, Cezary Kaliszyk, and Josef Urban. Munkres' general topology autoformalized in isabelle/hol, 2026.

[2] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings. Ed.: A. Platzer*, volume 12699 of *Lecture Notes in Computer Science (LNCS)*, pages 625–635. Springer International Publishing, 2021.

[3] Zarathustra Amadeus Goertzel. Mettamath: Integrating formal verification into an agi cognitive architecture via the metta language. 2025.

[4] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-formalisation of higher-order logic. *Journal of Automated Reasoning*, 56(3):221–259, Mar 2016.

[5] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: A verified implementation of ml. *ACM SIGPLAN Notices*, 49(1):179 – 191, 2014. Cited by: 73.

[6] Norman D. Megill and David A. Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press, Morrisville, North Carolina, 2019. `http://us.metamath.org/downloads/metamath.pdf`.

[7] L. Gregory Meredith. Graph-structured lambda theories: A reference for the working developer. https://github.com/F1R3FLY-io/publications/blob/main/GSLT-intro/omnibus.pdf, 2026. Draft.

[8] Lucius Gregory Meredith, Ben Goertzel, Jonathan Warrell, and Adam Vandervorst. Meta-metta: an operational semantics for metta, 2023.

[9] Michael Stay, L. Gregory Meredith, and Christian Wells. Generating hypercubes of type systems. https://github.com/F1R3FLY-io/publications/blob/main/drafts/Hypercube/main.pdf, 2026. Draft.

[10] Josef Urban. 130k lines of formal topology in two weeks: Simple and cheap autoformalization for everyone?, 2026.

[11] Christian Williams and Michael Stay. Native type theory. In Kohei Kishida, editor, *Proceedings of the Fourth International Conference on Applied Category Theory, ACT 2021, Cambridge, United Kingdom, 12-16th July 2021*, volume 372 of *EPTCS*, pages 116–132, 2021.

Appendix: Representative proofs

Here are three proofs partially chosen by Claude Code Opus 5 that should show what the proofs involved in verifying a verifier look like.

A.1 The declarative/operational bridge. The important top-level theorem connecting the declarative and operational semantics of Metamath: `Metamath/Spec/Equivalence.lean:3469`. One of the primary lemmas, however, is 723 lines, of which the ax (assumption) case is 603 lines:

```
theorem operational_iff_semantic {Γ : Database} {consts : ConstSet}
    {fr : Frame} {e : Expr}
    (h_wf : WellFormedDatabaseStrong Γ consts)
    (h_fr_nodup : FloatVarNoDup fr)
    (h_fr_disjoint : Spec.FrameVarsDisjointConsts consts fr)
    (h_supported : SemanticFrameSupported Γ fr) :
    Provable Γ fr e ↔
    Semantic.Provable (dbToAxioms Γ) (frameToContext fr)
      (exprToFormula (varMapOfFrame fr) e) := by
  constructor
  · -- Forward: Operational → Semantic
    exact operational_to_semantic h_wf h_fr_disjoint
  · -- Backward: Semantic → Operational
    exact semantic_to_proofValid h_wf h_fr_nodup h_fr_disjoint h_supported
```

One of the primary lemmas, however, is 723 lines, of which the ax (assumption) case is 603 lines:

```
theorem mario_to_proofValid_aux {Γ : Database} {consts : ConstSet} {fr : Frame}
    (h_wf_strong : WellFormedDatabaseStrong Γ consts)
    (h_fr_nodup : FloatVarNoDup fr)
    (h_fr_disjoint : Spec.FrameVarsDisjointConsts consts fr)
    {fmla : Semantic.Formula}
    (h_mario : SupportedProvable Γ fr fmla)
    {e : Expr}
    (h_eq : fmla = exprToFormula (varMapOfFrame fr) e) :
    Provable Γ fr e := by
  have h_wf : Spec.WellFormedDatabase Γ consts := h_wf_strong.1
  induction h_mario generalizing e with
  | hyp h h_in =>
      -- h_in : the declarative formula is a hypothesis of the frame's context.
      -- Recover the Hyp that produced it, then rebuild a one-step stack proof.
      rw [h_eq] at h_in
      obtain ⟨hyp, hyp_in, hyp_eq⟩ := hyps_correspondence h_in
      cases hyp with
      | essential e' =>
          simp only [hypToMarioFormula_essential] at hyp_eq
          -- Formula equality is injective on a fixed frame, so e' = e.
          have h_eq_expr := exprToFormula_eq_of_eq_frame hyp_eq.symm
          rw [← h_eq_expr]
          exact ⟨[ProofStep.useHyp (Hyp.essential e')], [e'],
                 ProofValid.useEssential fr [] [] e' hyp_in (ProofValid.nil fr), rfl⟩
      | floating c v => ...          -- 60 lines
  | var v => ...                     -- 67 lines
  | ax σ h_axs h_dj h_hyps h_hyps_var h_hyps_wf ih_h ih_var =>
      ...                            -- 603 lines: substitution, disjoint-variable
                                     -- side conditions, and the induction hypotheses
                                     -- for every mandatory hypothesis
```

"The `essential` branch above is the whole shape in miniature: recover the object the declarative side is talking about, prove the two representations agree, then exhibit the operational witness — here a one-step stack proof. Every other branch is that same obligation against a harder correspondence" (Opus 5).

A.2 Induction over a proof-validity derivation. *ProofValidFrom.append_suffix* proves that if a valid run of proof steps changes the stack from $\texttt{stk}_1$ to $\texttt{stk}_2$, then the same steps do the same with any suffix appended to both. This is what allows the verifier to run step-by-step, checking each step without knowing what sits further down the stack (Metamath/Spec/Operational.lean:154).

```
theorem ProofValidFrom.append_suffix
    {Γ : Database} {fr : Frame} {stk₁ stk₂ : List Expr} {steps : List ProofStep}
    (h : ProofValidFrom Γ fr stk₁ stk₂ steps) (suffix : List Expr) :
  ProofValidFrom Γ fr (stk₁ ++ suffix) (stk₂ ++ suffix) steps := by
  induction h with
  | nil =>
      simpa using (ProofValidFrom.nil fr (stk₁ ++ suffix))
  | useEssential stack steps e h_in _ ih =>
      simpa using (ProofValidFrom.useEssential fr (stk₁ ++ suffix)
        (stack ++ suffix) steps e h_in ih)
  | useFloating stack steps c v h_in _ ih =>
      simpa using (ProofValidFrom.useFloating fr (stk₁ ++ suffix)
        (stack ++ suffix) steps c v h_in ih)
  | useAxiom stack steps l fr' e σ h_find h_dv h_typed _ needed h_needed
      remaining h_stack ih =>
      -- Adjust the remaining suffix
      have h_stack' : stack ++ suffix = needed.reverse ++ (remaining ++ suffix) := by
        simpa [List.append_assoc] using congrArg (fun s => s ++ suffix) h_stack
      exact ProofValidFrom.useAxiom fr (stk₁ ++ suffix) (stack ++ suffix) steps
        l fr' e σ h_find h_dv h_typed ih needed h_needed (remaining ++ suffix) h_stack'
```

A.3 Any parser operation preserves database well-formedness. This is one of the most important theroems: every parser operation that modifies the growing database (and does not raise an error) maps well-formed databases to well-formed databases, providing the needed invarients to the verifier. It takes 691 lines and covers eight cases (`insert` of a constant, variable, hypothesis, or assertion; `pushScope`; `popScope`; `withFrame`; `id`), of which the constant case below is the shortest (Metamath/ParserCorrectness.lean:1139).

```
theorem structure_preserving_maintains_wf
    {op : DB → DB} (db : DB)
    (h_struct : StructurePreservingOp db op)
    (h_wf : WellFormedDB db)
    (h_no_err_before : db.error? = none)
    (h_no_err_after : (op db).error? = none) :
    WellFormedDB (op db) := by
  rcases h_wf with ⟨h_frame_wf, h_objs_wf⟩
  cases h_struct with
  | insert pos label obj h_validated h_obj_var_names_match h_fresh_db
      h_fresh_label h_fresh_in_asserts =>
      cases h_obj : obj label with
      | const c =>
          change WellFormedDB (db.insert pos label obj)
          change (db.insert pos label obj).error? = none at h_no_err_after
          constructor
          · -- Frame well-formedness is preserved.
            rw [insert_frame_unchanged]
            -- Freshness of the label rules out overwriting an existing variable.
            have h_not_var_dup : ¬(∃ v_dup, obj label = .var v_dup ∧
                db.find? label = some (.var v_dup)) := by
              intro ⟨v_dup, _, h_find_old⟩
              rw [h_find_old] at h_fresh_db
              cases h_fresh_db
            -- Every variable already in the database is named by its own label.
            have h_var_inv : ∀ lbl v, db.find? lbl = some (.var v) → v = lbl := by
              intro lbl v_old h_find
              exact var_label_eq_name_of_db ⟨h_frame_wf, h_objs_wf⟩ h_find
            exact insert_preserves_frame_wf db pos label obj db.frame
              h_frame_wf h_fresh_label h_no_err_before h_no_err_after
              h_not_var_dup h_var_inv h_obj_var_names_match
          · -- Every stored object is still well-formed.
            ...                          -- 60 lines: new object, then the old ones
      | var v => ...                     -- 132 lines
      | hyp ess f name => ...            -- 209 lines
      | assert fmla fr lbl => ...        -- 260 lines
  | pushScope => ...
  | popScope pos => ...
  | withFrame f h_preserves => ...
  | id => ...
```

This lemma delivers us the DBExecution theorem on preserving wellformedness.

```
theorem DBExecution.preserves_wellformedness {db₁ db₂ : DB} :
    DBExecution db₁ db₂ →
    db₁.error? = none →
    db₂.error? = none →
    WF.WellFormedDB db₁ →
    WF.WellFormedDB db₂ := by
  intro h_exec h_no_err1 h_no_err2 h_wf
  induction h_exec with
  | refl => exact h_wf
  | step db₁ db₂ db₃ h_step h_exec ih =>
      cases h_step with
      | op op_fn h_struct h_no_err_before h_no_err_after =>
          have h_wf2 : WF.WellFormedDB (op_fn db₁) :=
            structure_preserving_maintains_wf db₁ h_struct h_wf
              h_no_err_before h_no_err_after
          exact ih h_no_err_after h_no_err2 h_wf2
```