Formalizing Weak Memory Models in Lean with LLMs

Aresh Pourkavoos

December 5, 2025

1 Introduction

In this project, we use Lean 4 to formalize some prior results on weak memory models.[1] Specifically, we formally define the COM and GAM axiomatic memory model families and prove their equivalence. The proof (in the paper and in the Lean codebase) relies on two major lemmas: proving that the "Causality" and "SC-per-Location" COM axioms hold for GAM. Both of these require reasoning about paths and cycles in graphs which are nontrivial to implement in Lean.

2 Software

The Lean code was written by two LLMs: Claude 4.5 Sonnet and Claude 4.5 Opus. The formalization began with Sonnet, then continued with Opus, a more capable model, after its release on Nov 24, 2025. Both were operating inside Claude Code, a harness which provides the model with basic tools like editing files, running Bash commands, and spawning more agents to perform subtasks and avoid context clutter. It also allows the user to connect "plugins" via the Model Context Protocol (MCP), which allows the model to communicate with a server via JSON requests, as well as "skills," which include instructions for specific tasks. I used the Lean 4 Theorem Proving skill set developed by Cameron Freer [2], which includes instructions for Lean tasks like proof repair, golfing, and `sorry` resolution.

3 Process

3.1 Context management

In the first few sessions, I had Claude read the compiled paper PDF at the beginning. However, this filled up context quickly and contained a significant amount of irrelevant text, such as the bibliography. As such, I decided to use the LaTeX source instead, dedicating a session to reading it and indexing it in CLAUDE.md, a file in the project root directory which is loaded into context at the beginning of each session. Additionally, to ensure coherence across multiple sessions, I created a PLAN.md file, meant to store a summarized state of the current code and next steps. I added instructions to CLAUDE.md for Claude to execute at the beginning of each section unless instructed otherwise (paraphrased):

- Read PLAN.md for the current state
- Read the relevant LaTeX source file(s) given the index
- Continue with the next steps as planned
- At the end of the session, modify PLAN.md with the current state and next steps

[1] Sizhuo Zhang et al. "Weak Memory Models with Matching Axiomatic and Operational Definitions". In: *CoRR* abs/1710.04259 (2017). arXiv: `1710.04259`. URL: `http://arxiv.org/abs/1710.04259`.

[2] github.com/cameronfreer/lean4-skills

The Claude Code harness already contains some to-do list functionality, so this was a simple way to allow the state to persist in a way Claude was familiar with. Claude Code also has an "auto-compact" feature, which summarizes the entire current context once it is nearly full and begins a new session with that summary as the initial prompt. However, in the past, I have had bad experiences with auto-compact as it would lose relevant details; better to start anew and ensure that things like the detailed source material of the paper are not summarized.

3.2 Ongoing guidance

On its own, Claude is quite adept at identifying and working around unexpected issues in the environment, but without further guidance, it has to rediscover these issues every session, which wastes tokens. I supervised Claude as it wrote, and When I noticed recurring problems, I added notes about them to CLAUDE.md. For example, the `use` tactic is part of Lean 4 mathlib, which this project does not rely on. Claude's Lean training data likely contains mathlib and dependent projects, so by default, it attempts `use` where it may be relevant. It was instructed to avoid this tactic.

Additionally, Claude attempted to "simplify" paper definitions on a few occasions, which would have compromised the fidelity of the formalization. For instance, the paper's definition of memory instructions allowed register values to be used in addresses computations, but the initial Lean definition only allowed for static addresses. While this simplified some of the proofs by allowing locations to be directly compared, I had to emphasize that the paper's definitions must be followed in this case, which necessitated the addition of a runtime address assignment `aa`.

Finally, for both of the major lemmas, Claude had its own ideas about how to modify the proof for ease of implementation. Unfortunately, neither of these worked and led to a significant waste of tokens and time. This problem is similar in nature to the previous one, and given that the model was able to complete the proof with more explicit guidance to stick to the paper, I think it's fair to say that these were a result of Claude's personality or overestimation of its own abilities. Other frontier models, such as GPT-5.1, are more known for taking instructions very literally; these differences likely reflect the RL enviroments in which they were trained, and in future work, I would be curious to see how other models fare on this task when provided with the same Claude Code agent harness and plugins.

4 Lean Project Structure

The project contains 16 Lean files totaling over 5000 lines of code.

- `GamI2e/Types.lean`: Core type definitions including instructions, addresses, registers, fence kinds, and address/data operands. Also defines address assignments for handling runtime-computed addresses.
- `GamI2e/Relations.lean`: Fundamental binary relations including program order, memory order, read-from, coherence order, from-reads, and their compositions. Also defines acyclicity and transitive/reflexive closures.
- `GamI2e/Ordering.lean`: The `Ordered` function parameterizing GAM, with concrete instantiations for TSO, RMO, SC, WMM, ARM, and RISC-V memory models or model families. Includes I2E compatibility proofs.
- `GamI2e/PreservedPO.lean`: Preserved program order (ppo) definition as the transitive closure of preserved memory/fence order, preserved dependency order, and preserved same-address order.
- `GamI2e/Axiomatic.lean`: GAM axiomatic model with execution witnesses, the Inst-Order axiom (ppo implies mo), and the Load-Value axiom (loads read from mo-maximal visible stores).
- `GamI2e/Operational.lean`: GAM operational model with ROB-based abstract machine, processor states, execution rules for fetch/execute/store operations, and address assignment extraction.
- `GamI2e/COM.lean`: Module aggregator re-exporting all COM submodules.

- `GamI2e/COM/Definitions.lean`: COM axiom definitions (SC-per-Location, Causality), the extended communication order (eco), and helper lemmas about loads/stores.
- `GamI2e/COM/CausalityHelpers.lean`: Utility lemmas for causality proofs including `com_contained_in_mo` showing rfe/co/fr/ppo are all contained in mo between memory instructions.
- `GamI2e/COM/Causality.lean`: Main causality proof infrastructure showing TransGen paths through the causality relation between memory instructions map to mo paths.
- `GamI2e/COM/Equivalence.lean`: Main equivalence theorems (`GAM_subset_COM`, `COM_subset_GAM`, `GAM_iff_COM`). Includes the axiomatized Szpilrajn extension theorem for constructing total orders.
- `GamI2e/COM/ECO.lean`: Module aggregator for ECO submodules.
- `GamI2e/COM/ECO/Base.lean`: Fundamental ECO lemmas including `eco_total_per_address` (eco is total over same-address accesses) and `eco_poloc` (poloc implies eco).
- `GamI2e/COM/ECO/SCRel.lean`: SC-per-Location relation definition (rf $\cup$ co $\cup$ fr $\cup$ poloc) and basic lemmas about edge types.
- `GamI2e/COM/ECO/SCRelPath.lean`: Path utilities for SC-per-Location including length-indexed paths (`SCRelLen`), path decomposition, and load/store chain analysis.
- `GamI2e/COM/ECO/SCPerLocation.lean`: Main SC-per-Location acyclicity proof showing store-to-store paths reduce to mo paths via cycle analysis.

5 Problems Encountered

The Lean LSP can take a moment to synchronize with a file after changes have been made, which occasionally leads to confusing tool call outputs where the proof state does not appear to change after an edit. This was circumvented by falling back to `lake build`, which did not take excessively long due to the project's relatively small size.

I ran this project on NixOS, which clears all environment variables by default when starting a new process in a shell. This included the `$CLAUDE_PLUGIN_ROOT` variable, which the Lean plugins relied on to locate essential scripts. This broke most of the Lean plugins, but once the problem was identified, it was an easy fix.

Near the end of the project, the custom subagent functionality was broken by an update. Due to time constraints, it was not debugged, and Claude used its usual subagents to perform tasks like filling sorries instead.

6 Other Notes

Claude 4.5 Sonnet's approach to interactive proofs was somewhat different to Opus's. Sonnet tended to use the LSP more like a human, generating a few tactics at a time and querying it for feedback. Opus, on the other hand, often generated dozens of lines of Lean at once, then tried to fix any errors. Despite the large amount of state involved at any given point in a Lean proof, Opus's approach worked surprisingly well, often only having to fix a few minor issues per block.

A few pieces of the code are incomplete and marked as such; they are not used in the COM = GAM proof but would need to be filled in if the other results were to be formalized. The project also defines the Szpilrajn extension theorem and takes it as a classical axiom in the otherwise constructive proof. It could potentially be avoided by explicitly requiring all structures to be finite.

References

Zhang, Sizhuo et al. "Weak Memory Models with Matching Axiomatic and Operational Definitions". In: *CoRR* abs/1710.04259 (2017). arXiv: 1710.04259. URL: http://arxiv.org/abs/1710.04259.