Generating Lean 4 Tactics from User Specifications: Strategies for LLM-Assisted Metaprogramming

Extended Abstract — Work in Progress

Silvère Gangloff[1], Jan Hula[1], and Miroslav Olšák[2]

[1] University of Ostrava, Czech Republic
[2] Czech Technical University in Prague

Motivation and context. Metaprogramming in Lean 4 allows for powerful automation through custom tactics, but it presents a steep learning curve. Developers must master the `TacticM` monad, Lean's internal `Expr` representation, and a vast, evolving API in *Mathlib*. While Large Language Models (LLMs) have shown proficiency in generating proof scripts (sequences of existing tactics), their ability to synthesize new *tactic procedures* remains largely unexplored. This work investigates structural strategies to make LLMs reliable partners in Lean 4 metaprogramming.

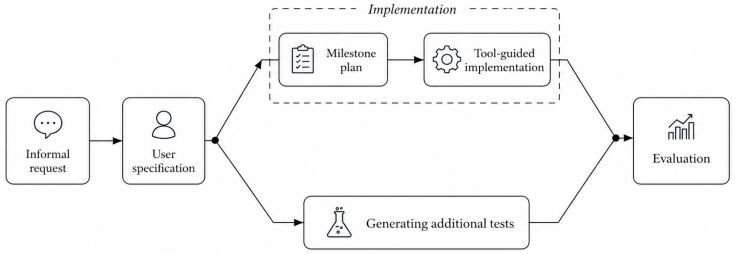
Challenges of metaprogramming synthesis. Our investigation began with a pilot study on generating a tactic for intuitionistic propositional logic (e.g., proving $A \Rightarrow ((A \Rightarrow B) \Rightarrow B)$). We observed that LLMs perform significantly better when prompted with a formal natural language description of the tactic's logic rather than attempting to generalize dynamically from few-shot examples. With the first method, one iteration was enough to generate a tactic proving all provable implications with at most 9 propositions, while it took several self-correcting iterations with dynamic few-shot prompting in order for the generated tactic to prove all provable implications with less than 6 propositions. This experiment revealed that tactic synthesis is qualitatively more difficult than standard code generation due to *API volatility*. Mathlib's naming conventions, while systematic, are subject to frequent renames and deprecations. This "knowledge cutoff" makes it difficult for models to generate correct imports and lemma references. While these errors can theoretically be fixed via a self-correction loop, such loops often fail to converge for complex tactics because a single API error can necessitate a cascading rewrite of the tactic's entire architecture.

Methodology - 2×2 experiment. We propose a pipeline that anchors the generation process in a mathematically rigorous interpretation of user intent. For a given request, we first generate a dual-part specification: a *user part* (functional requirements and test cases) and a *programming part* (implementation constraints). Using the user specification, we generate four versions of each tactic crossing two scaffolding variables:

- *Milestone planning:* Decomposing the specification into a sequential plan of incremental functionalities. This draws on *Least-to-Most Prompting* strategies to reduce the cognitive load of the generation.

– *Lean LSP-MCP Tools:* Providing the LLM with live tool access via the Model Context Protocol (MCP). We used open source tools found here. This includes real-time compiler diagnostics, goal state inspection, and semantic Mathlib search via *LeanSearch* or *Loogle*.

The main branch of this pipeline is illustrated on the following figure.



We use Claude code programmatically with Opus 4.7 for the generation of the tactics.

Evaluation framework A primary difficulty in tactic synthesis is the lack of objective metrics to ensure the output matches the user's intent. We evaluate the model using the following metrics:

1. *Test-case generalization:* We use the *user part* of the specification to generate a set of "held-out" mathematical examples that were not seen during the implementation phase.
2. *Convergence rate:* We measure the number of self-correction steps required for each condition to reach a state that passes the test suite.
3. *Heartbeats:* We measure the average number of heartbeats consumed during tactic execution over all the examples.

We evaluate our pipeline on user specifications generated out of tactic wishes found in this mathlib issue.

Discussion and future work. Preliminary observations suggest that live tool access significantly reduces the need for cascading self-correction by catching API errors at the source. The combination of **Informal Request → Formal Spec → Milestone Plan → Tool-Guided Implementation** provides a robust blueprint for LLM-assisted formal development.

Future work will refine the milestone process by incorporating *cumulative reasoning* paradigms. We also plan to expand our benchmark to other tactics, and collect public human feedback to further validate the utility of the generated tactics.