Proposal Prospectus:
Structured Semantic Context for Programming Processes

Andrew Blinn, University of Michigan

December 7, 2025

1 Thesis Statement

Programming is decision-making, and decisions require context. When programmers read and write syntax, derived semantic information informing their edits, including types, values, and control flow, is often disconnected or distant from the code itself. Bridging this gap requires mental correlation or manual collation—cognitive costs that compound as codebases grow. Modern language servers provide a wealth of such information, but too much information, or irrelevant information, leads to poor decisions. This is especially true for LLM-driven agents, which are prone to over-index on whatever is immediately in front of them. This thesis therefore defends the following claim:

Effective semantic contextualization of programming processes requires iteratively reducing indirection between syntax and derived semantic information, surfacing what is relevant without overwhelming or misleading the decision-maker.

2 Introduction

Hi! This is not yet a complete thesis proposal, but represents a close approximation of my research direction for the purposes of committee selection. I'm still tightening up the specifics; if you're interested in serving on my committee, please let me know any particular points you'd want addressed or elaborated on.

My research focuses on user interfaces that expose structured semantic context to inform programming processes for both human and AI programmers. In particular, I am investigating using types and runtime data to collate and annotate code with relevant, actionable semantic information, including suggestions for contextual actions that support structured code authoring and debugging.

This work divides into three phases:

- **Finished work: Static Contextualization (Blinn *et al.*, 2024):** Here we used type-directed methods to provide large language models with targeted contextual information during program sketching, driven by typed holes in the Hazel editor. **Evaluation:** We evaluated the system with a custom benchmark suite, MVUBench, intended to address gaps in the coding benchmark landscape as described in Section 2. We found significant gains in performance in cases which are context-starved or subject to conflation involving name reuse.
- **Current work: Dynamic Contextualization:** My current focus is Live Probes, a system for programmers to selectively surface intermediate runtime values alongside their code, extending the Hazel live programming features from (Omar *et al.*, 2019). Probes augment live values with other dynamic information like the environment and call stack in a lightweight way to aid comprehension and debugging while mitigating information overload. The implementation is largely complete. **Evaluation:** We plan to assess the design using the framework of Information Foraging Theory, and the implementation via a user study.
- **Future work: Semantic Context Engineering:** My cumulative project focuses on applying concepts from structured and semantics-driven editing to context engineering for LLM-driven coding agents. We plan to incorporate the extant static and dynamic contextualization implementations as tools to expose to the agent, as well as new tools for structured editing and strategic scaffolding for programming workflows (e.g. type and test-driven development). **Evaluation:** We will be evaluating coding model performance ablated across combinations of query, editing, and strategy tools.

3 Static Contextualization

In (Blinn *et al.*, 2024), we tackled the issue of LLM-based code completion systems hallucinating broken code due to lack of appropriate code context, particularly when working with definitions that are neither in the training data nor near the cursor. We showed that tighter integration with the type and binding structure of the language in use, as exposed by its language server, can address the contextualization problem in a token-efficient manner.

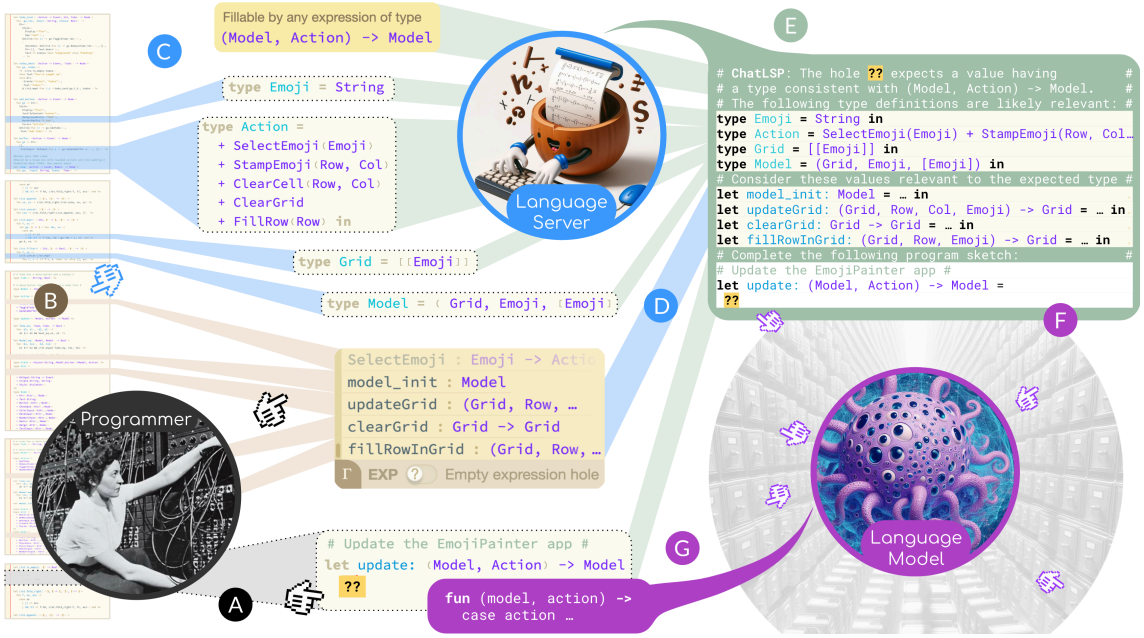

Figure 1: A programmer requests a hole filling (A) by typing **??**, either intentionally or in a fit of frustration. The Hazel Language Server provides codebase-wide (B) semantic information relevant to the hole, collecting types based on the expected type (C) and selecting type-relevant headers from the context (D). These are combined into a contextualized text prompt (E) which is sent (F) to the LLM resulting in hole filling (G).

Implementation: In particular, we integrated LLM code generation into the Hazel live program sketching environment. The Hazel Language Server is able to identify the type and typing context of the hole that the programmer is filling, with Hazel's total syntax and type error correction ensuring that a meaningful program sketch is available whenever the developer requests a completion. This allows the system to prompt the LLM with codebase-wide contextual information that is not lexically local to the cursor, nor necessarily in the same file, but that is likely to be semantically local to the developer's goal. Completions synthesized by the LLM are then iteratively refined via further dialog with the language server, which provides error localization and messages.

Evaluation: To evaluate these techniques, we introduced MVUBench, a dataset of model-view-update (MVU) web applications with accompanying unit tests that have been written from scratch to avoid data contamination, and that can easily be ported to new languages because they do not have large external library dependencies. These applications serve as challenge problems due to their extensive reliance on application-specific data structures.

Through an ablation study, we examined the impact of contextualization with type definitions, function headers, and error messages, individually and in combination. We found that contextualization with type definitions is particularly impactful. After introducing our ideas in the context of Hazel, a low-resource language, we duplicated our techniques and ported MVUBench to TypeScript in order to validate the applicability of these methods to higher-resource mainstream languages. We also outlined ChatLSP, a conservative extension to the Language Server Protocol (LSP) that language servers can implement to expose capabilities that AI code completion systems of various designs can use to incorporate static context when generating prompts for an LLM.

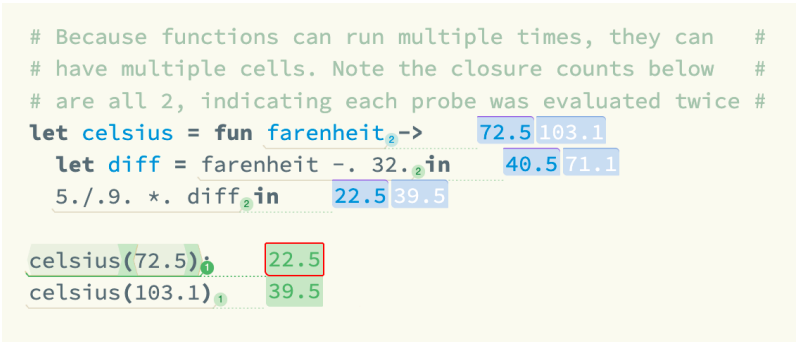

Figure 2: Basic Probe UI. Here there is a probed expression on each line, indicated by the colored underlines. A small number indicates the number of captured samples; 2 each for the three probes within the functions, and 1 for each call site. The sample for the first call site is selected, which results in the samples corresponding to that call in the function literal being highlighted.

4 Dynamic Contextualization with Live Probes

Programmers routinely step away from code to external consoles and debuggers, incurring indirection costs to correlate behavior with the source. While modern debuggers can render expression values inline, and inline evaluation in editors like Emacs can selectively render values on-demand, when expressions occur inside function literals, both of these depend on using indirect tools (stepwise debuggers) to navigate linearly through a program trace before any values can be seen. Conversely, time-travel or "omniscient" debugging approaches present an entire programming trace, but typically do so in a way that is entirely removed from the main syntax editor. Ever-popular print statement debugging, on the other hand, presents a very simple interaction model, but still requires a sometimes-cognitively-costly effort to correlate the program trace with the instrumented expressions.

Implementation: Live Probes are the result of our attempt to develop a progressive approach to providing dynamic context to programmers. A probe can be placed on any expression (or pattern) via a context menu or keyboard shortcut, resulting in a sample appearing to the right of the line, presenting a value taken on by that expression during execution; see Figure 2. If there is more than one such value, arrow keys or on-screen affordances can be used to navigate between them. Alternatively, a keyboard shortcut or GUI toggle can be used to instead show all collected samples in a row. Samples consist not just of values, but also the environment, evaluation context, and call stack when that sample was recorded. We are refining ways to progressively expose this information to users without overwhelming them with unnecessary detail.

The most similar existing work is likely Sorin Lerner's projection boxes (Lerner, 2020). In comparison, we put particular emphasis on the connection between selected samples across different probes with respect to position in the call stack to give a more global view of how information flows through the program.

We also have an alternate mode, **autoprobing**, suitable for debugging and exploratory programming. When enabled, probes are automatically and dynamically placed on heuristically selected expressions of the current top-level definition, one per line, as seen in Figure 3. This mode enables the user to flexibly steer where probes are placed without the need for an additional control surface, demonstrating intent via line break placement. This mode, in particular the steering via line breaks, was inspired by REPL-driven development. Here though both the expressions whose values are surfaced and their values are provided as you type, updating asynchronously. Autoprobes are intended as a supporting and exploration-oriented feature, suitable as an alternate to using a REPL to try out unfamiliar functions or compose a data transformation pipeline. Using autoprobes, a user can start with tests (TDD) or use sites in live code, and use the resulting live values to inform their writing process without waiting until their implementation is complete.

We have also implemented a variety of more advanced features to help with code comprehension and debugging. Figure 4 demonstrates **call pinning**, used to restrict the number of samples presented.

Figure 5 illustrates the **dynamic cursor**, which allows the user to point to a step in execution, as represented by a given sample, resulting in other samples being styled in ways illustrating their relation to the indicated sample; such as being part of the same function call, or above or below

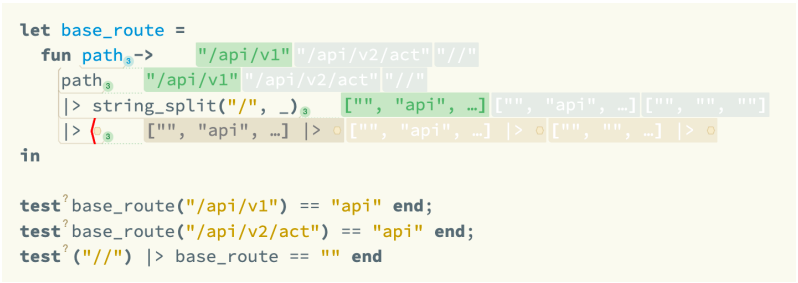

Figure 3: Here a user is implementing a function to extract the base element of a URL. They have already written tests, whose values drive the samples shown. In this example, the string-split function takes two arguments which are both strings; type-based hinting alone won't reveal which is which, but the live values provide clearer indication, possibly preventing an error which might be harder to track down by the time the implementation is actually complete.

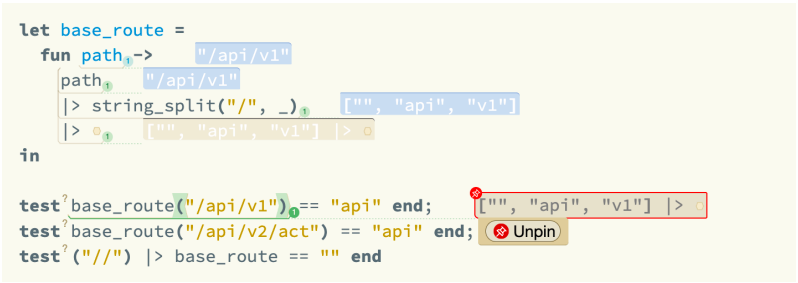

Figure 4: Users can narrow the range of samples shown by selecting the 'pin' option on samples for function calls, which restricts the displayed samples to those downstream of that call.

each other in the call stack. We are currently evaluating which such properties are actually useful to expose to users via formative testing.

Evaluation: We plan a design evaluation leveraging Information Foraging Theory and a user study tracking performance and qualitative impressions during debugging and authoring tasks.

In the IFT framework (Fleming *et al.*, 2013), more recently employed specifically in the live programming domain by (Rein *et al.*, 2025), a user (termed "predator") seeks out information in an environment consisting of a network of information patches linked by traversable edges, each with an associated traversal cost. We intend to use IFT as a lens into the cognitive cost of indirection in programming tasks which typically interleave between examining syntax and runtime data. Foraging here might consist of tracing an unexpected result back to its origin, traversing the program's data dependency graph, reading code fragments to identify both dependencies and suspect logic, and exposing values to form or validate hypotheses. In particular, we offer the ability to navigate across 'horizontal' information links between samples of an expression originating at different points in a program trace, a costly movement in a traditional debugger.

For our user study, the goal is to explore whether inline probes reduce indirection and cognitive load relative to print statements, and assess how auto-probing affects errors during program writing. This will likely be a small-N study (N = 9–15 total participants) with an emphasis on think-aloud and post-task interviews.

For the debugging portion we intend to do a between-subjects study comparing Hazel with simple print statement support (our baseline) to Hazel with probes. We are considering an ablation covering a variety of probe features beyond their basic application. The tasks will be debugging programs with failing tests, requiring corrections on the order of 2–4 lines of code in programs ranging from dozens to the low hundreds of lines. For the program writing task, we will be providing a series of small programming tasks featuring provided tests.

5 Semantic Context Engineering for AI Coding Agents

As the programming abilities of LLMs improve, programmers are increasingly using these models as the core of long-running processes intended to accomplish large-scale programming tasks. For

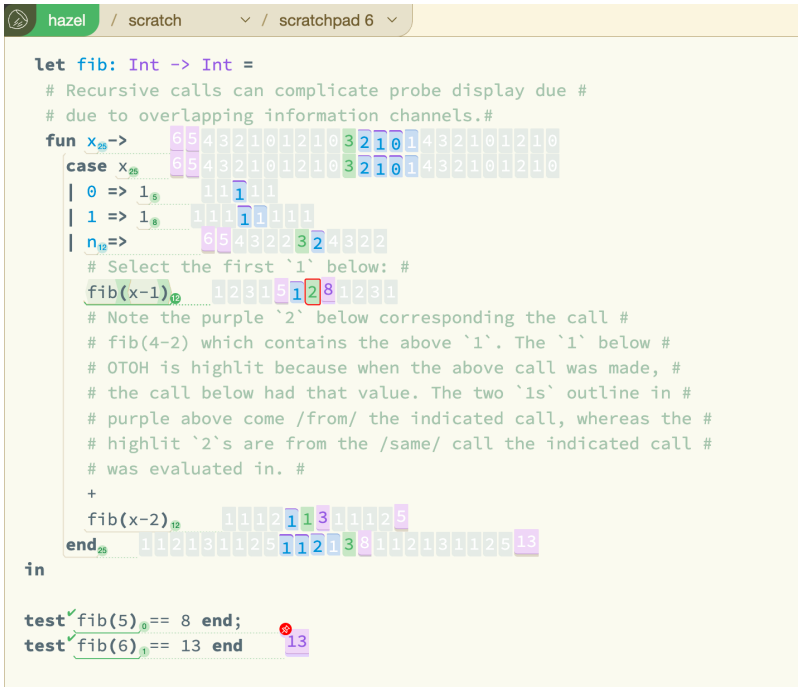

Figure 5: Here the user has pinned an external call to the recursive function 'fib(6)', which is autoprobed. The dynamic cursor (the red outline around the '2' near the center) 'points' to a particular moment in evaluation, and the other samples are colored according to their relation to the indicated sample. Green samples are part of the same function call, blue samples are from calls under the current call in the call stack, and pink samples are from calls over the current call.

our purposes, we consider an "LLM agent" to be any process which emits calls to external tools in a loop (in our case, code navigation, editing, and contextualization tools) to achieve a goal (Willison, 2025). While results in this domain are impressive, many pitfalls limit current agents, including high costs incurred iterating on syntax errors from improper edit patches, poor or misleading context due to reliance on RAG or plain text search, and failures of models to properly assess what tools can be productively employed in a given context.

We seek to address this through a process we provisionally term Semantic Context Engineering, which incorporates the above static and dynamic contextualization efforts, along with semantically contextualized structured edit actions. We intend to (a) provide a set of semantic query tools for models to build and maintain rich and precise code context, proactively and on-demand, and (b) to help contextually scaffold agent type- and test-driven development processes by providing tools for syntax and semantics-respecting structural edits and refactors to prevent the context-window-filling churn seen with current less structured tools like `grep` and `diff`-based patching.

We have built the basic harnesses to support an agentic loop in the Hazel editor, and are currently building out a variety of semantic tools for the model to use. As this is a very active area of research, we are exploring multiple leads, intended to winnow the most promising ones as the project solidifies. Broadly, we characterize our approach as giving the model navigation and semantic query tools to build code context, and then using that code context to selectively suggest to the model what editing tools might be most useful at the current stage in the process. We are exploring the following genres of tools:

- **Structured editing (in the large)**: Instead of diff-based patches, model edits will be conducted structurally, initially at the level of definitions, featuring actions like 'add definition', 'update definition', 'update function signature', etc.

- **Structured editing (in the small)**: Instead of code patches per se, we intend to offer more granular actions with clear semantic meaning. For example, an action to case on a particular value from the context, replacing a hole with a match expression, its branches already populated with the patterns of the relevant sum type. Similarly, refactoring actions to raise definitions etc. instead of wholesale rewriting.

- **Structured search & navigation**: We are investigating structured ways of providing code

maps to the agent by selectively expanding and collapsing branches of the overall code tree according to semantically derived relevance to the current task. For moving the focus within the tree, we will offer the ability to jump to definitions and filter the view to references rather than relying on purely text-based search

- **Static contextualization**: Both proactive contextualization as in our prior work, when the agent is focused on a hole, and tools for on-demand context, e.g. type-based lookup.
- **Dynamic contextualization**: We intend to surface the Live Probes for both manual and automatically exposing intermediate values during composition.

Implementation: I am currently supervising two undergraduate students working on the basic scaffolding for an agentic loop in the Hazel editor, building off of the prototype I implemented to support the static contextualization experiments. We have implemented preliminary versions of several of the above tools.

Evaluation: We will evaluate the finished system on an extension of the MVUBench evaluation suite from the static contextualization paper, featuring modification tasks which will be judged using held-out test suites. We intend to ablate over a selection of the above tools. Automated evaluation of coding agents is challenging, as the greater the realistic complexity of the task, the more a satisfying solution tends to be open-ended. We want the agent to be able to make real structural decisions about the code, especially rich fundamental data types, which will help it leverage the most gains out of the semantic tooling. Thus we want a testing strategy that minimizes coupling with application internals. We are investigating a variety of options, including browser-automation-based testing, but the tentative compromise plan is to give the agent prose descriptions which feature the specific data constructors for new actions to be implemented to add to the action type, and specify the signature of the accessor functions which will allow us to automatically extract the aspects of the state we want to test. The former allows us to issue precise commands, at the cost of constraining the shape of the action type, but the latter allows the model to freely determine the core data model type, as long as the accessors remain intact.

References

Blinn, Andrew *et al.*, (2024). "Statically Contextualizing Large Language Models with Typed Holes", *Proc. ACM Program. Lang.* Vol. 8 No. OOPSLA2. DOI: 10.1145/3689728. **available at**: https://doi-org.proxy.lib.umich.edu/10.1145/3689728.

Fleming, Scott D. *et al.*, (2013). "An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks", *ACM Trans. Softw. Eng. Methodol.* Vol. 22 No. 2. ISSN: 1049-331X. DOI: 10.1145/2430545.2430551. **available at**: https://doi-org.proxy.lib.umich.edu/10.1145/2430545.2430551.

Lerner, Sorin (2020). "Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming", *CHI '20: Proc. 2020 CHI Conference on Human Factors in Computing Systems*. ACM. DOI: 10.1145/3313831.3376494. **available at**: https://dl.acm.org/doi/10.1145/3313831.3376494.

Omar, Cyrus *et al.*, (2019). "Live Functional Programming with Typed Holes", *Proc. ACM Program. Lang.* Vol. 3 No. POPL. Hazelnut Live: dynamic semantics for incomplete programs. DOI: 10.1145/3290327. **available at**: https://dl.acm.org/doi/10.1145/3290327.

Rein, Patrick *et al.*, (2025). "An Information Foraging Interpretation of Liveness", *2025 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. To appear. IEEE, pp. 128–138. DOI: 10.1109/VL-HCC65237.2025.00022.

Willison, Simon (2025). "I think "agent" may finally have a widely enough agreed upon definition to be useful jargon now", **available at**: https://simonwillison.net/2025/Sep/18/agents/ (accessed 5 Nov. 2025).