Research Statement · Will Crichton

I design principled, practical systems to amplify the intelligence of programmers.

Note: *this is an early draft for public circulation, not the final version of the statement.*

All people should be able to program, and all programmers should be able to build complex software. However, computational literacy today is like textual literacy in antiquity; just as reading was the exclusive skill of the ancient privileged castes, so too is programming the exclusive skill of 1% of today's population. Programmers are themselves heavily stratified. Over half of programmers do exclusively front-end web development. The number of people who can build reliable software systems or analyze large datasets is vanishingly small. My mission is to lower the barrier to entry for programming at all levels of complexity.

My approach to this problem is to build systems that amplify the intelligence of programmers, in the sense used by Bush [1], Engelbart [7], and Licklider [8]. I identify core cognitive tasks that are a challenging part of routine programming, and I design tools to support those tasks with the combined powers of computation and representation. To both ground my ideas and maximize my impact, I work within programmer communities that apply cutting-edge tools to design complex systems, with my current focus being on the Rust programming language. Specifically, I have developed tools to help Rust programmers:

- Find relevant code by visualizing dependencies in the IDE [6], used by 5,000+ Rust developers.
- Learn programming languages with data-driven pedagogies [5], used by 50,000+ Rust learners.
- Link API documentation to relevant examples [2, 3], merged into Rust and used by 100+ Rust libraries.

It is difficult to design effective systems for intelligence amplification with intuition alone. People are hard to predict, and programs are hard to analyze. In my research, I seek to build theories as much as systems; to contribute to a shared foundation of knowledge about how programs works (programming language theory) and how people program (cognitive psychology). For example:

- I ran several experiments to understand how working memory influences program comprehension [4], which motivated my work on visualizing dependencies to overcome working memory limitations.
- I designed a formal semantics for my dependency analysis and demonstrated its generality by proving a key theorem, termination-insensitive non-interference [6].

To elaborate on these examples, I will first explain in-depth the research arc of the dependency analysis tool, which constituted my dissertation. Then I will explain my ongoing research around teaching Rust at scale. Finally, I will conclude with directions for future work.

From Working Memory to Information Flow

My dissertation work started with a simple question: which psychological theories can make concrete predictions about programming tasks? After reviewing the literature on applied cognitive psychology, I honed in a common theme: *working memory*. Working memory is the cognitive capacity for holding and processing information in the short-term. The key finding is that working memory has a universally limited capacity of 7-ish "chunks" of information. So I asked: how would a programmer's working memory limitations influence their ability to perform programming tasks?

In "The Role of Working Memory in Program Tracing" [4], I report on four experiments that examine the limits of human performance in a straightforward programming task: tracing, or mentally simulating a program's behavior with concrete inputs. When given a simple program like `x = 8; y = 2; z = 4` and then asked the value of y (i.e., paired-associate cued recall), we showed that most participants started to make errors after about 7 variables, which is consistent with previously-identified working memory limits. More interestingly, we asked participants to trace a program with an accumulating dependency structure

like `x = 8; y = x + 2; z = y - 1; ...`, and we provided an interface to track a participant's attention by blurring out code outside the region of the participant's cursor. We found evidence for two distinct tracing strategies (reading "linearly" top-down, or reading "on-demand" in reverse dependency order), which corresponded to distinct kinds of working memory errors (forgetting the value of a variable, forgetting the location of a previous computation). Overall, the experiments demonstrated that a programmer's working memory severely limits their ability to mentally maintain program state while tracing, and the nature of that state depends on the programmer's particular tracing strategy.

This result motivated the question: what kinds of cognitive support can help programmers overcome such working memory limitations? IDEs already provide limited support through tools like "Jump to Definition", but they provide no support for the general task of following the dependency structure of a computation. In research, that task has historically been the domain of *program slicing*—yet despite decades of work, no slicer is in widespread use today. So I focused on the problem of developing a new program slicer with two criteria: (1) the analysis is practical enough to run on large codebases, and (2) the interface is useful enough to provide cognitive support for program comprehension tasks.

In "Modular Information Flow through Ownership" [6], I describe a practical analysis for static slicing of Rust programs, or more generally for computing information flow (of which slicing is a special case). The key insight is that both alias analysis and mutation analysis can be made modular by careful use of the Rust type system. The information flows induced by a function call can be approximated by analyzing the ownership types (lifetimes and ownership qualifiers) of the inputs and outputs to a function, without needing the function body itself. To evaluate its soundness, we proved that this approximation satisfies termination-insensitive non-interference within a formal model of Rust. To evaluate its precision, we analyzed $\approx 400,000$ lines of Rust code and found that this approximation is equivalent to a whole-program analysis in 94% of cases, meaning that little precision is lost. The modular information flow algorithm is publicly available as the Flowistry tool (willcrichton/flowistry). I developed an IDE tool that interactively visualizes Flowistry's output as program slices, which has been used by over 5,000 Rust developers to date.

Teaching Ownership Types at Scale

A key cognitive task for all programmers is *learning* — programmers must continually adapt to new domains, new constraints, and new tools. My research therefore naturally extends to augmenting the process of learning about programming. The challenge of learning is particularly acute for Rust, whose combination of concepts from functional and systems programming is notoriously difficult to master. During my postdoc, I addressed this problem by developing an experimental platform for teaching Rust at scale (rust-book.cs.brown.edu). The Rust Book Experiment is a fork of *The Rust Programming Language*, the official textbook provided by the Rust community. The central idea is to embed interactive quizzes within the book. Learners benefit because the quizzes help them engage with the material. We benefit by collecting large quantities of data about which parts of Rust that people find most difficult. Over the last year, 50,000+ people have answered over 1,000,000+ quiz questions using our platform.

In "A Grounded Conceptual Model for Ownership Types in Rust" [5], I describe one part of this experiment focused on ownership types. We first borrowed an idea from learning science of the *concept inventory*, or a narrowly-scoped exam with multiple-choice questions whose distractors reflect common misconceptions. We developed an "Ownership Inventory" to evaluate a person's understanding of ownership in Rust. First, we designed test items by analyzing StackOverflow questions about ownership to identify concrete situations involving difficult ownership errors. Second, we ran a user study with $N = 36$ Rust learners and asked them to debug the ownership error in each situation. We used common incorrect responses in the user study as distractors for the Ownership Inventory.

Our core observation was that learners could recognize the surface reason for why a program is rejected (e.g. there are two mutable references to the same data), but they could not articulate the underlying reason

(e.g. with a particular input, the function would cause undefined behavior). We developed a new pedagogy of ownership types to address this issue. At the heart of the pedagogy is a conceptual model of ownership types as a flow-sensitive system of statically-tracked permissions (to read, write, or own data). We created a compiler plugin that automatically generates a diagram to show how statements in a Rust program affect the permissions on each variable. Finally, we developed a replacement chapter for the Rust Book about ownership that integrates these diagrams, which you can read at this link.

To evaluate the permissions pedagogy, we deployed the Ownership Inventory in the Rust Book Experiment and gathered several weeks of data with the original ownership chapter from *The Rust Programming Language*. Then we deployed our new chapter, and gathered another several weeks of data. Across 18 questions on the Inventory, we found that the new chapter improved average quiz scores by 10% from 48% to 58% ($N = 312$, $p < 0.001$, $d = 0.57$). More broadly, this experiment validated our hypothesis that gathering quiz data at scale would be a useful tool for designing and evaluating language learning interventions.

Future Work

The human factors of programming is an enormous research space with bewilderingly little prior work. Just within the Rust community, I have more ongoing research projects:

- Rust libraries use increasingly sophisticated type-level programming to provide static correctness guarantees, but type errors involving such libraries are often incomprehensible. We are working on exposing the internals of the type system to facilitate type-level debugging.
- Flowistry provides the capability to statically analyze dependencies in arbitrary Rust programs. We are developing a system for catching security and privacy violations in Flowistry's dependence graph.

But here I want to briefly discuss some research questions beyond Rust that I'm interested to work on.

What should be the successor to TEX? The modern document leverages our teraflops of processing power and millions of pixels to… render stylized text on emulated 8.5×11 pieces of paper. Absurd! That made sense at the birth of TEX in 1978, but the future of technical communication needs better document technologies. I am interested in this problem from the cognitive and PL perspectives. First, what are the essential aspects of reading comprehension that could be better facilitated by changes to the reading medium? For instance, I strongly suspect that "explorable explanations" are a far less cognitively effective augmentation to technical communication when compared to basic ideas like linking every paper-specific symbol back to its definition. But this begs the PL perspective: how should we design a document language to make the necessary cognitive augmentations practical for all authors, especially those without significant training in technical communication, education, psychology, and so on? My prototype system Nota (nota-lang.org) is one step in this direction, and I intend to continue exploring down this path.

How will access to AI programming tools change the nature of programming expertise? Tools like GitHub Copilot have made a splash in the programming community. For programmers, the question is: how should they most effectively incorporate these tools into their workflow? For CS educators: how should they change their curriculum and problem sets in response to the capabilities of these tools? For programming language designers: how should they design language syntax and semantics when knowing that future code is equally likely to be written by a machine as much a person?

Psychology provides some broad frameworks which are useful here, namely: recognition vs. recall. Programming has long been a recall-oriented activity — programmers had to commit to memory the specific names of keywords and what they mean, and then write programs by recalling those keywords. AI shifts the balance towards recognition — programmers can ask for a program in prose, and then recognize whether the generated program is what they wanted. I want to explore the cognitive factors that influence

program recognition: how the features of a program and the nature of a person's background affect help or hamper their ability to distinguish desirable from undesirable programs.

What are useful, lightweight metrics for comparing the expressiveness of software systems? Library developers love to market their software as being "easy to use" or "for humans". Researchers claim that their systems are often "more expressive" than prior work. However, the singular metric that often justifies these claims is lines-of-code. Despite its imperfection as a metric, but no one has yet come up with an alternative besides "run a large-scale user study" or "deploy the tool in industry and see who adopts it." So here's a hypothesis: the length of the argument for why a program is correct (say, the size of a proof in a proof assistant) has a higher correlation with program comprehension effort than the length of the program itself. If this hypothesis could be experimentally validated, then that would be grounds for developing a new expressiveness metric. A system that lends itself to short proofs across a benchmark of reference tasks could be predicted as more usable (to an expert) than a system requiring longer proofs.

References

[1] Vannevar Bush. 1945. As We May Think. *The Atlantic* 176, 1 (1945), 101–108.

[2] Will Crichton. 2021. Documentation Generation as Information Visualization. In *Proceedings of the 11th Annual Workshop on the Intersection of HCI and PL (PLATEAU 2021)*. http://reports-archive.adm.cs.cmu.edu/anon/isr2020/abstracts/20-115E.html

[3] Will Crichton. 2021. RFC #3122: Automatically scrape code examples for Rustdoc. https://github.com/rust-lang/rfcs/blob/master/text/3123-rustdoc-scrape-examples.md

[4] Will Crichton, Maneesh Agrawala, and Pat Hanrahan. 2021. The Role of Working Memory in Program Tracing. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 56, 13 pages. https://doi.org/10.1145/3411764.3445257

[5] Will Crichton, Gavin Gray, and Shriram Krishnamurthi. 2023. A Grounded Conceptual Model for Ownership Types. (*Under submission*).

[6] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. 2022. Modular Information Flow through Ownership. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3519939.3523445

[7] Douglas Engelbart. 1962. *Augmenting Human Intellect: A Conceptual Framework*. Technical Report. Stanford Research Institute.

[8] J. C. R. Licklider. 1960. Man-Computer Symbiosis. *IRE Transactions on Human Factors in Electronics* HFE-1, 1 (1960), 4–11. https://doi.org/10.1109/THFE2.1960.4503259