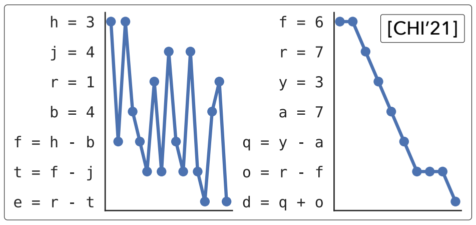
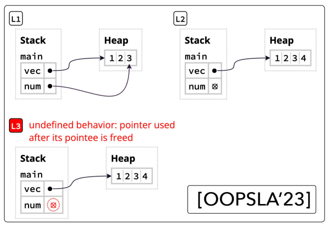
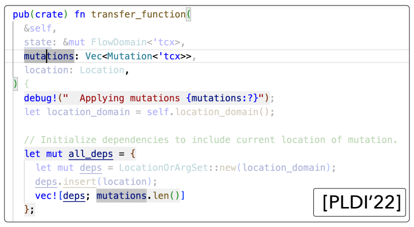
Research Statement · Will Crichton



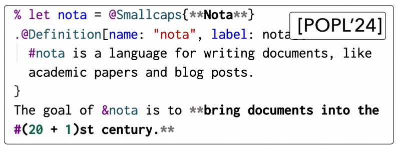
```
h = 3          f = 6
j = 4          r = 7
r = 1          y = 3
b = 4          a = 7
f = h - b      q = y - a
t = f - j      o = r - f
e = r - t      d = q + o
```



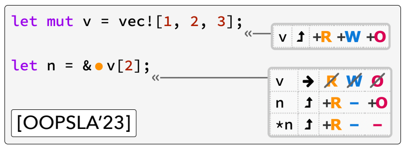


```
pub(crate) fn transfer_function(
  &self,
  state: &mut FlowDomain<'tcx>,
  mutations: Vec<Mutation<'tcx>>,
  location: Location,
) {
  debug!("  Applying mutations {mutations:?}");
  let location_domain = self.location_domain();

  // Initialize dependencies to include current location of mutation.
  let mut all_deps = {
    let mut deps = LocationOrArgSet::new(location_domain);
    deps.insert(location);
    vec![deps; mutations.len()]
  };
```

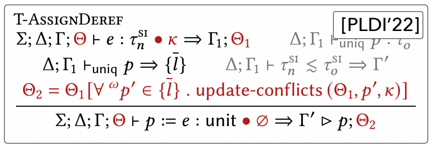


```
% let nota = @Smallcaps{**Nota**}
.@Definition[name: "nota", label: nota_
  #nota is a language for writing documents, like
  academic papers and blog posts.
}
The goal of &nota is to **bring documents into the
#(20 + 1)st century.**
```



```
let mut v = vec![1, 2, 3];

let n = &●v[2];
```



T-AssignDeref

$$\frac{\Sigma;\Delta;\Gamma;\Theta \vdash e : \tau_n^{\text{SI}} \bullet \kappa \Rightarrow \Gamma_1;\Theta_1 \qquad \Delta;\Gamma_1 \vdash_{\text{uniq}} p : \tau_o \qquad \Delta;\Gamma_1 \vdash_{\text{uniq}} p \Rightarrow \{\bar{l}\} \qquad \Delta;\Gamma_1 \vdash \tau_n^{\text{SI}} \lesssim \tau_o^{\text{SI}} \Rightarrow \Gamma' \qquad \Theta_2 = \Theta_1[\forall\,{}^{\omega}p' \in \{\bar{l}\}\ .\ \text{update-conflicts}\,(\Theta_1, p', \kappa)]}{\Sigma;\Delta;\Gamma;\Theta \vdash p := e : \text{unit} \bullet \varnothing \Rightarrow \Gamma' \triangleright p;\Theta_2}$$

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. The number of programmers who can build reliable software systems or analyze massive 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 computing pioneers like Bush [1], Engelbart [8], and Licklider [9]. I identify core cognitive tasks that are a challenging part of routine programming, and I design systems to support those tasks. To both ground my ideas and maximize my impact, I work within programmer communities that apply cutting-edge tools to design complex software, 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 [7] (top-right), used by 5,000+ Rust developers.
- Learn programming languages with data-driven pedagogies [5] (center), 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 four experiments to understand how working memory influences program comprehension [4] (top-left), which motivated my work on visualizing dependencies to overcome working memory limitations.
- I formalized my static dependency analysis for Rust (bottom-right) and demonstrated its generality by proving a key theorem, termination-insensitive non-interference [7].
- I designed a formal semantics for document languages to provide a precise theoretical foundation for my ongoing work into designing a successor to LaTeX [6] (bottom-left).

Overcoming Working Memory Limitations with Information Flow Analysis

Intelligence amplification requires an understanding of programmer intelligence. So 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 found a common theme: *working memory*, or 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. I asked: how would a programmer's working memory affect their ability to perform programming tasks?

"The Role of Working Memory in Program Tracing" [4] reports on four experiments examining the limits of human performance in a straightforward programming task: *tracing*, or mentally simulating a program's behavior. When given a simple program like `x=8; y=2; z=4` and then asked the value of `y` (i.e., a paired-associate cued recall task), we found that most participants started to make errors after about 7 variables, consistent with working memory theory. More interestingly, we asked participants to trace a program with an accumulating dependency structure like `x=8; y=x+2; z=y-1; ...` We provided an interface to track a participant's attention by blurring-out code not under the participant's cursor. We found evidence for two distinct tracing strategies: reading "linearly" top-down and reading "on-demand" in reverse dependency order. These strategies corresponded to distinct working memory errors: forgetting the value of a variable, and forgetting the location of a prior computation. Overall, the experiments showed that a programmer's working memory severely limits their ability to mentally maintain program state, and the nature of that state depends on the programmer's chosen strategy.

This result motivated the question: how can tools amplify a programmer's working memory to overcome these limitations? IDEs provide limited support with "Jump to Definition", but they provide no support for generally following the dependency structure of a computation. This 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 developing a new program slicer with two criteria: (1) the analysis is practical enough to run on large codebases, and (2) the interface provides cognitive support for program comprehension tasks.

"Modular Information Flow through Ownership" [7] describes a practical static slicer for Rust, 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. Our algorithm can analyze function calls only using ownership annotations on the type signature, without needing the function body itself. To evaluate soundness, we proved that this approximation satisfies termination-insensitive non-interference within a formal model of safe Rust. To evaluate 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

Another key cognitive task for all programmers is *learning* — a particular problem for Rust, whose combination of concepts from functional and systems programming is notoriously difficult to master. During my postdoc, I set out to systematically improve Rust's learning curve by developing an experimental platform for teaching Rust at scale. The Rust Book Experiment (rust-book.cs.brown.edu) is a fork of *The Rust Programming Language*, the Rust community's official textbook. The key 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.

"A Grounded Conceptual Model for Ownership Types in Rust" [5] describes one part of this experiment focused on ownership types. We first ran a formative study with $N = 36$ Rust learners who answered open-ended problems about ownership. We found 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 code would cause undefined behavior). We developed a new pedagogy of ownership types to address this disconnect. At its heart is a conceptual model of ownership types as flow-sensitive *permissions* to read, write, or own data. We built a compiler plugin that takes a Rust program and generates a diagram showing how each statement affects permissions. Finally, we wrote a replacement chapter on ownership for the Rust Book that uses these diagrams (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 9% from 48% to 57% ($N = 312$, $p < 0.001$, $d = 0.56$). 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

I have several ongoing research projects within the Rust community:

- Rust libraries increasingly rely on the type system for static correctness which leads to incomprehensible diagnostics. We are making a type-debugger to help users diagnose errors when the compiler cannot.
- 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 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 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 the useful framework of *recognition vs. recall*. Programming has long been a recall-oriented activity — programmers commit to memory the syntax and semantics of a language, and write programs by recalling that knowledge. AI shifts towards recognition — programmers ask for a program, and recognize whether the generated code is satisfactory. I'm interested in the cognition of 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 in Rust. *Proc. ACM Program. Lang.* OOPSLA2 (October 2023). arXiv:2309.04134 *(To appear)*.

[6] Will Crichton and Shriram Krishnamurthi. 2024. A Core Calculus for Documents. *Proc. ACM Program. Lang.* POPL (January 2024). arXiv:TODO-FILL-ME-IN-BEFORE-SHARING-THIS *(To appear)*.

[7] 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

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

[9] 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