Teaching Statement · Will Crichton

To me, teaching is one of the great joys of academia. I love to guide students to a deep understanding of the fundamental ideas in computer science. This is evident in my track record: in undergrad, I was a TA every semester from sophomore to senior year, and I ran my own mini-course for two years. Students said in TA reviews that "out of all the TAs I had experiences with, Will stands out by far," and "[Will's] skill at lecturing is something to which I aspire." For three years of grad school, I taught CS 242, the graduate course on programming languages. Students said in course reviews that CS 242 was "the most valuable computer science course I have taken at Stanford" and "one of the best taught in the CS department."

Understanding how people learn is also an integral part of my research on amplifying the intelligence of programmers. Programming is, in a sense, a process of continual learning. Today's rote tasks are automated by tomorrow's abstractions, and a programmer must take time to learn those tools. In 2023, a programmer needs to learn countless concepts: programming languages, paradigms, frameworks, libraries, databases, devtools, and so on. Subsequently, a critical role for intelligence amplification is to help programmers more quickly acquire new skills. To that end, I have done research on topics like teaching ownership types in Rust [4] and improving the quality of automatically generated documentation [2, 3].

Teaching Experience

My formative experience was teaching six semesters as an undergraduate teaching assistant at Carnegie Mellon. In that time, I was a TA for *Functional Programming*, *Parallel and Sequential Data Structures and Algorithms*, *Parallel Computer Architecture and Programming*, and *Compiler Design*. Through recitations I learned how to work one-on-on with students to debug their mental models of course concepts. I also created a new mini-course, *Game Development on the Web*. This course taught me how to construct an informative and engaging curriculum under serious resource constraints like one hour of lecture per week.

My crystallizing experience with teaching three quarters as a solo instructor at Stanford. In 2017, Stanford did not have a professor available to teach CS 242: Programming Languages, so I volunteered to take over the course. "From Theory to Systems: A Grounded Approach to Programming Language Education" [1] describes how I redesigned CS 242 to make PL theory more accessible to the average CS student by teaching concepts in real-world contexts. The use of formal semantics is motivated with WebAssembly, the most widely-used programming language with a complete formal semantics [6]. The ideas in the lambda calculus are motivated with Rust, a widely-used systems programming language that combines functional and imperative programming. (Students even read Wadler's paper on linear types [9] in one assignment!) The final curriculum is online here: `https://stanford-cs242.github.io/f19/`

In anonymous course reviews for the last iteration of Fall 2019, students gave "How much did you learn from this course?" a mean of 4.3/5, and "How would you describe the quality of instruction in this course?" a mean of 4.4/5. Positive comments from students included:

- "This is the best CS class I've taken at Stanford. You're doing yourself a disservice by not taking it."
- "I really loved this class, and it probably will be one of the most impactful on my skills as a software engineer when I leave Stanford."
- "Will is one of the best lecturers at Stanford, hands down."

The course was not perfect. The experience taught me about the cost of high velocity in curricular development. Constant reshaping of the course meant occasional bugs and late releases of assignments. One student wrote, "I did not like this class because though I worked very hard [...] I still do not feel like I learned much. [...] Code was often changed multiple times before the due date." I take this feedback to heart, and I will continue iteratively improving on my teaching skills.

Teaching Philosophy

My teaching philosophy starts with curricular design. I focus on balancing declarative (what is a concept) and procedural knowledge (how to use a concept). Traditional instruction often focuses too much on either declarative knowledge or rote procedural knowledge, such as CS1 classes that teach the semantics of a for-loop, but do not teach how to debug programs involving a for-loop. I take inspiration from texts like *How to Design Programs* [5] which articulate procedural knowledge as "recipes" for building software.

As an example from CS 242, a traditional lesson on the lambda calculus will include declarative knowledge about concepts like substitution and alpha-renaming. Beyond explaining the mechanics of substitution, I will explain the procedural knowledge of how to use substitution in the design of a new language extension. I present lambda calculus extensions, show how to identify the binding structure of the feature, and describe both correct and incorrect ways to express the binding structure via substitution.

Most learning happens outside the classroom, so I take assignment design seriously. I create assignments that have a plausible connection to the real world (no "foo"s and "bar"s). Each problem has an articulable learning goal. I used to just design programming problems, but I have since come to appreciate how written problems can surface student misconceptions that can be glossed over with auto-grading.

To get a sense of how I lecture, I recommend skimming my Strange Loop 2021 talk on "Type-Driven API Design in Rust": https://www.youtube.com/watch?v=bnnacleqg6k

Connecting Teaching to My Research

Unlike most other candidates, teaching plays an essential role in my research. I study how to make advanced programming tools more usable, and a key factor is the tool's learning curve. For example, Coq has been around for 34 years, yet most people still learn Coq via a *de facto* apprenticeship at one of the few universities with Coq experts. The Rust roadmap's [8] number one priority is to "flatten the (learning) curve." Even for a popular language like C++, Stroustrup [7, p. 132] writes, "most students emerge from universities with only weak and inaccurate understanding of C++ [...] So much could be done to improve software if more developers using C++ knew how to use it better!"

But who is positioned to address this problem? CS education researchers and HCI researchers focus primarily on novice developers using languages like Python and Javascript. Software engineering researchers focus primarily on developers who already have expertise in their language of use. Programming language researchers have written many textbooks, but that has not been enough to overcome the learning barrier.

No, what we need is researchers with deep expertise in *both* language design and computing pedagogy who are ready to develop a science of how to teach these languages. A science of: what does it mean to be an effective systems programmer, or functional programmer, or proof engineer? How do we augment the languages and their devtools to teach the requisite skills, to ease learners into the long learning curve? How do we translate the large body of tacit knowledge that constitutes expertise into an explicit curriculum? I seek to address these kinds of questions in my own research. To see a concrete example of my vision, see the section "Teaching Ownership Types at Scale" in my research statement.

Courses I Can Teach

Once I become faculty, I would be excited to teach undergraduate courses such as: programming languages, compilers, functional programming, parallel computing, web development, and human-computer interaction. I can also teach graduate courses on topics such as: performance engineering, type-safe systems programming, design and implementation of domain-specific languages, tools for thought, and of course my specialty: human-centered design of programming tools.

References

[1] Will Crichton. 2019. From Theory to Systems: A Grounded Approach to Programming Language Education. In *3rd Summit on Advances in Programming Languages (SNAPL 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:9. https://doi.org/10.4230/LIPIcs.SNAPL.2019.4

[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, 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).*

[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to design programs: an introduction to programming and computing*. MIT Press.

[6] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[7] Bjarne Stroustrup. 2020. Thriving in a Crowded and Changing World: C++ 2006–2020. *Proc. ACM Program. Lang.* 4, HOPL, Article 70 (jun 2020), 168 pages. https://doi.org/10.1145/3386320

[8] Josh Triplett and Niko Matsakis. 2022. Rust Lang Roadmap for 2024. https://blog.rust-lang.org/inside-rust/2022/04/04/lang-roadmap-2024.html

[9] Philip Wadler. 1990. Linear types can change the world!. In *Programming concepts and methods*, Vol. 3. Citeseer, 5.