Teaching Statement · Will Crichton

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

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 with teaching was six semesters as an undergraduate teaching assistant at Carnegie Mellon. In that time, I was a TA for *Functional Programming* (twice), *Parallel and Sequential Data Structures and Algorithms* (twice), *Parallel Computer Architecture and Programming*, and *Compiler Design*. Running office hours and recitations, I learned how to work with students individually to debug their mental models about course concepts. I also created and taught a 1-unit mini-course for two semesters titled *Game Development on the Web*. This course taught me how to construct an informative and engaging curriculum under serious constraints, namely one hour of lecture per week with a single final project.

My crystallizing experience with teaching was three quarters as an instructor at Stanford. In 2017, due to staff shortages, Stanford did not have a professor available to teach CS 242: Programming Languages. The previous instructor (also a grad student) had just graduated. I volunteered to take over the course, redesigned it and, proceeded to teach it every fall for the next three years.

I describe the design of the course in my paper "From Theory to Systems: A Grounded Approach to Programming Language Education" [1]. The core idea is to make programming language theory more accessible to the average CS student by placing each idea in the context of real-world low-level systems. The use of formal semantics is motivated with WebAssembly, the most widely-used assembly 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). On each iteration of the course, I revised the curriculum and especially the assignments to find the right balance of student effort and insight. 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 would say this is one of the best courses I've taken and will revolutionize the way you think about software development!"

- “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.”
- “Very few times in my five years at Stanford I’ve seen a professor that cares as much and will put as much effort into their class.”

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 find the core challenge to be striking a balance between declarative and procedural knowledge: knowing *what* a concept is versus knowing *how* to use a concept. Traditional instruction focuses too much on either declarative knowledge or rote procedural knowledge, such as introductory CS classes that teach the definition 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 provide constructive “recipes” for building software, such as the example where students are led to reinvent insertion sort by following an inductive list-processing recipe.

As an example from CS 242, a traditional lesson on the lambda calculus will include declarative knowledge about syntax and semantics, how substitution and alpha-renaming work, and so on. The declarative/procedural line is fuzzy — for example, substitution is a procedure that students can work through. The key distinguishing factor is that procedural knowledge involves using a concept in open-ended ways. Understanding how to use substitution in the design of a new language extension is an important form of procedural knowledge. Concretely, I give a lecture about substitution that includes examples of identifying binding structure in new features, as well as seeing examples of incorrect applications of substitution.

Most learning happens outside the classroom, so I take assignment design seriously. I create assignment narratives that have a plausible connection to the real world (no “foo”s and “bar”s). I design each part of the assignment with an articulable learning goal that matches a specific section of the lecture curriculum. While I originally used almost entirely programming problems, I have since come to appreciate the role in purely written problems in spotting student misconceptions that can be glossed over in a program.

I use a combination of audience participation and charisma to keep lectures informative and at least a little fun or entertaining. (I have built a stage presence through several years of debate and mock trial.) I try to limit my use of slide-only lectures, especially because it is easy to rip through slides at a pace exceeding the cognitive limits of the audience. I tend to either write on a whiteboard or live code. For a polished example of a live coding lecture, you can peruse my Strange Loop 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 focus on how to help programmers use advanced programming tools, and a key challenge in that process 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. Even for a popular language like C++, Stroustrup [7, p. 132] writes:

> In most countries, most students emerge from universities with only weak and inaccurate

> understanding of C++ and the key techniques for using it. This is a serious problem for the C++ community. No language can succeed at an industrial scale without a steady stream of enthusiastic programmers mastering its key design and implementation techniques. So much could be done to improve software if more developers using C++ knew how to use it better!

The Rust roadmap for 2024 [8] similarly cites as its number one priority: "flatten the (learning) curve: scaling to new users and new use cases."

But who is positioned to address this problem? CS education researchers and HCI researchers focus primarily on undergraduate CS1 and K-12, not advanced languages. 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 novices 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 how I work in this space, see the "Teaching Ownership Types at Scale" section of 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: language-oriented 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. TODO: ADD THE CITATION ONCE THE ARCHIVAL VERSION IS RELEASED.

[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.