PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

November 15, 2021

(This is a publicly posted generic cover letter that describes my background and teaching experience. It is not tailored to a specific university. Please see my application materials submitted to your university for the tailored version I wrote to you.)

Dear Search Committee,

I am writing to apply for a teaching position at your university, starting in the 2022-2023 academic year. I received my Ph.D. from the Paul G. Allen School of Computer Science & Engineering at the University of Washington and have taught in the Allen school variously as a 50%-time and 100%-time temporary lecturer since March 2020. This academic year, I am teaching full-time in the fall and spring, and half-time in the winter. I believe my classroom teaching experience at the University of Washington, my background in the programming languages and systems research communities, and my teaching philosophy and future teaching goals make me a good fit for your university and I would be excited to work with new colleagues in this position.

I have been the instructor of record for 7 course offerings at the University of Washington. First, as a graduate student in winter 2017, I taught CSE 341 (undergraduate programming languages). Then, as a temporary lecturer, I have taught two more offerings of 341, as well as CSE 331 (software design and implementation), CSE 374 (non-majors intro to systems), CSE 490P (advanced undergrad programming languages and verification), and CSEP 505 (graduate programming languages). In addition, I TAed 9 courses as both an undergraduate (at Williams College) and graduate student from 2010 to 2019, and I am actively involved in mentoring and outreach work at the high school, undergraduate, and graduate levels since 2014. Throughout this time, I have worked with over 500 students at the undergraduate, BS/MS master's, professional master's, and Ph.D. levels, across different backgrounds, including non-majors and majors, and across both electives and required courses. Later this academic year, I am also scheduled to teach CSE 452 (distributed systems) and a CSE 490 special topics course on web browser internals from a systems perspective. This broad experience gives me the flexibility to teach a wide array of subjects, generally in the programming, languages, and systems areas, which I believe would make me a useful resource at your university.

My research focuses on building more reliable software by applying formal reasoning techniques to real systems. In my thesis work, I, along with my collaborators, built verified implementations of widely used distributed protocols, including the widely used Raft consensus protocol. Here, by "verified", I mean that we constructed a machine-checked proof of the global correctness of our implementation of the system. Such a proof was challenging to develop because of the scale and complexity of the system, and because we insisted on reasoning about real executable code, not just abstract models. My primary research expertise is in the tools and techniques required to build such proofs. I have also worked on a wide variety of other topics in programming languages, verification, and systems, including floating point, inductive invariant inference, dynamic program analysis, concurrency, and 3d printing. After graduate school, I worked full-time as CTO of Certora, a company that is building exactly these kinds of verification tools for blockchain smart contracts, where correctness is of paramount concern due to potentially catastrophic financial losses in the presence of bugs.

Throughout my career, my work has been driven by wonderful collaborative relationships. As I discuss in more detail in my teaching statement, I believe that my position in the research community enables a "stewardship of ideas" that directly supports my teaching by allowing me to remain up to date with the field, increasing the depth of understanding I am able to impart to my students, and providing opportunities to mentor students (especially undergraduates) in their own research projects. While I plan to continue participating in research and working with my collaborators, I want to be absolutely clear that I am seeking a full-time position where teaching will be my primary responsibility, and I have no plans to apply for industry, tenure-track, or research-first positions during this year's job cycle or in the foreseeable future.

My teaching philosophy is student-centered, collaborative, and stewardship-driven. This philosophy has been guided by my experience in the Allen school in the computing education seminar (CSE 590E), teaching in the classroom, and through teaching and research collaborations with colleagues. I have had the pleasure of participating in the computing education seminar (CSE 590E) since autumn 2020. It has been an absolutely transformative experience for me, giving me the vocabulary and techniques to cast my intuitions about teaching into crisp analyses and interventions. In particular, last fall, 590E spent the quarter discussing mastery-based grading, which inspired me to experiment with alternative grading techniques in CSE 374 and CSEP 505 later that year. While not perfect, I was very pleased with the early outcomes of these pilots, and I look forward to continuing to experiment with course design in the future, informed by education research.

Another highlight of my time in the Allen school has been the opportunity to collaborate with colleagues on the design of CSE 341 and to co-teach an offering of CSE(P) 505. In CSE 341, Dan Grossman, Zach Tatlock, and I have converged on a unified set of materials that we all contribute to each quarter and continue to improve. The original course design is entirely due to Dan. I have made several relatively minor contributions, including developing a new homework that uses JSON to teach pattern matching. I also converted the course from SML to OCaml and replaced Ruby with the Racket object system. Zach has also improved the OCaml workflow significantly by introducing the dune build system to the course. But what is most exciting to me about this situation is not the "who-did-what" of it, but the fact that we are building on a shared foundation of materials that is easy to reuse and easy to improve. It is a great feeling to fix a typo on the slides or a bug in the homework code and *know* that next quarter will be able to take advantage of the fix seamlessly.

I also had the opportunity to co-teach one section each of CSE 505 (for Ph.D. students) and CSEP 505 (for professional master's students) with Zach last spring. Officially, Zach was the instructor of record for CSE 505, and I was the instructor of record for CSEP 505. Zach and I both attended all lectures for both courses, and tag-teamed the lectures, alternating every 20 minutes or so. We completely redesigned the course from the ground up, producing new lecture materials and new homework assignments. This was one of the most enjoyable and productive professional experiences of my career. It helps that Zach and I work well together, but I also think co-teaching allowed us to create something greater than the sum of our parts. We piloted an additive points-based grading scheme that gave students the flexibility to choose what grade they wanted in the course and decide what assignments they needed to complete to achieve that grade. We also encouraged community building and student collaboration by providing a shared editable document where the entire class could construct a document explaining the (challenging!) concepts of the course on their terms. Over the quarter, students together produced over 100 pages of notes. I also developed an interactive web editor for one of the theoretical languages we study in the course, System F. By turning the theoretical into something interactive that they could practice with, students gained insight into what the language was like to program in, which helped them understand the theory more deeply. Together, the grading system, community notes, and interactive material design are examples of centering the student experience in the classroom, which I elaborate on further in my teaching statement.

My experience co-teaching 505 and collaborating on 341 make me very optimistic about the future of undergraduate and graduate CS education as we continue to scale both in terms of number of students and quality of education. It would be my great pleasure to continue to work towards these goals at your university.

Please find my CV, teaching statement, diversity statement, and references enclosed. I would be happy to answer any questions you may have or provide additional information. I look forward to hearing from you.

Sincerely,

James R. Wilcox
Lecturer, Paul G. Allen School of Computer Science & Engineering
University of Washington
Box 352355 / Seattle, WA 98195 / 615.332.1102
`jrw12@cs.washington.edu` / `https://jamesrwilcox.com`

James Rasmussen Wilcox

Curriculum Vitae

Paul G. Allen School of Computer Science & Engineering
University of Washington, Box 352350
Seattle, WA 98195

jrw12@cs.washington.edu
jamesrwilcox.com
Office: AC210

EDUCATION

Ph.D.	Computer Science & Engineering	University of Washington	Aug. 2021
M.S.	Computer Science & Engineering	University of Washington	Dec. 2015
B.A.	Computer Science and Mathematics	Williams College	June 2013

PRIMARY EMPLOYMENT

Paul G. Allen School of Computer Science & Engineering, University of Washington.

- Full-time Temporary Lecturer. September 2021–present
- Part-time Temporary Lecturer. March 2020–June 2021

TEACHING INTERESTS

I am interested in teaching programming, languages, and systems courses across all levels and backgrounds, and to both CS majors and non-majors. My primary teaching experience is in the "300- to 500-level" range, i.e., from undergraduate students who have just finished CS2 up through advanced undergraduate, master's, and Ph.D. students, but I am also open to teaching CS1 and CS2 with appropriate support.

CLASSES TAUGHT AT UW (7 offerings total)

CSE 331: Software Design and Implementation.

- Instructor of record: Fall 2021. 110 students. Managed 9 TAs.
- This required class comes after CS2 and teaches students how to write code of higher quality and increased complexity. We study Hoare logic, abstract data types, design patterns, testing, and advanced Java concepts.

CSE 341: Undergraduate Programming Languages.

- Instructor of record: Fall 2021, Fall 2020, and Winter 2017. 70–110 students and 5–7 TAs per offering.
- Contributed new homework and slides to pool of curriculum materials shared by all course instructors.
- Ported the course from SML to OCaml and from Ruby to object-oriented Racket.

CSE 374: Intermediate Programming Concepts and Tools.

- Instructor of record: Winter 2021. 80 students and 7 TAs.
- This is an introductory systems programming class *for non-majors*, covering C programming, the shell, and basic software engineering techniques such as version control and testing.
- Piloted contract-based assignments and grading.

CSE 490P: Advanced Programming Languages and Verification.

- Instructor of record: Spring 2020. 10 students and 1 TA.
- Pitched and designed this new advanced undergraduate seminar on the theory and practice of programming languages and verification. Students learned to do metatheory and implement type checkers and interpreters.
- Developed new course materials including ∼25 lectures and 12 homeworks (6 written and 6 programming).

CSE 505, CSEP 505: Graduate Programming Languages.

- Instructor of record: Spring 2021. Co-taught with Prof. Zachary Tatlock. 50 students and 4 TAs.
- Two different student populations: Ph.D. students and professional master's students.
- Piloted additive and transparent grading system.
- Developed new web IDE system for programming in System F.

ADDITIONAL TEACHING EXPERIENCE

- Teaching Assistant, CSEP 505, Spring 2019. Instructor: Prof. Zachary Tatlock.
- Outreach: I worked with a local high schooler weekly on learning to program in JavaScript. 2014–2017.
- Teaching Assistant, CSE 505, Autumn 2013. Instructor: Prof. Zachary Tatlock.
- Undergraduate Teaching Assistant, Williams College, Computer Science and Math departments, 2010-2013.

OTHER EMPLOYMENT

- Advisor (part-time), Certora, January 2021–present
- CTO, Certora, June 2019–December 2020
- Research Assistant, Prof. Zachary Tatlock, University of Washington, Seattle, WA, 2013–2019
- Research Intern, Dr. Jay Lorch, Microsoft Research, Redmond, WA, Summer 2017
- Research Assistant, Prof. Stephen Freund, Williams College, Williamstown, MA, 2012–2013
- Research Intern, Prof. Scott Shenker, ICSI, Berkeley, CA, Summer 2011
- Systems Administrator, Mary Bailey, Williams College, Williamstown, MA, Summer 2010

AWARDS AND FELLOWSHIPS

- Distinguished paper award, PLDI 2020
- Distinguished paper award, PLDI 2015
- National Science Foundation Graduate Research Fellowship, 2013–2018

ADVISING AND MENTORING

Undergraduate Research Supervised

- Steve Anton. Formal verification of the Raft consensus protocol. 2015-2016.
- Ryan Doenges. Verifying distributed systems with dynamic participants. 2015-2017.
  Now a Ph.D. student at Cornell.
- Miranda Edwards. Compositional verification of distributed systems. 2016-2017.
- Justin Adsuara. Verified serialization. 2017-2018.
- David Thien. Compiler testing. 2018.
- Ethan Shea. Automated verification of distributed systems. 2018.
- Taylor Blau. PL techniques for 3D printing and compositional verification of distributed systems. 2017-2018.

SIGPLAN Mentoring Program

- Karuna Grewal. 2020–present. I helped mentor Karuna through the Ph.D. application process. She is now a Ph.D. student at Cornell.
- Dani Wang. 2020–present. I helped mentor Dani through the Ph.D. application process. They are now a Ph.D. student at UT Austin.
- George Pîrlea. 2020–present. George is a second-year Ph.D. at NUS.
- Aaron Weiss. 2021–present. Aaron is a fifth-year Ph.D. student at Northeastern.
- Siddharth Bhat. 2021–present. Siddharth is a first-year Ph.D. student at the University of Edinburgh.

SOFTWARE MAINTAINED

`mypyvy`, Primary Developer and Maintainer, 2018–present

- An intermediate language and toolkit for manipulating symbolic transition systems.
- Transition systems are written in a decidable fragment of first-order logic.
- Supports verifying and synthesizing inductive invariants and several forms of bounded model checking.
- Has provided the basis for implementing tools described by several of my recent papers (CAV19, POPL22), as well as several papers by other researchers of which I am not an author.
- `mypyvy` itself is implemented in statically typed (!) Python and uses Z3 and CVC5 as underlying solvers.
- Available on GitHub: https://github.com/wilcoxjay/mypyvy

SERVICE

- POPL 2022 PC member
- OSDI 2021 ERC member
- ASPLOS 2021 ERC member
- VMCAI 2021 PC member
- OOPSLA 2020 external reviewer
- PLDI 2019 external reviewer
- POPL 2019 external reviewer
- OOPSLA 2018 external reviewer
- POPL 2017 external reviewer

PUBLICATIONS

Invited Articles

- Highlights in Systems Verification.
  James R. Wilcox.
  Communications of the ACM (**CACM**), February 2018.

Conference Publications

- Property-Directed Reachability as Abstract Interpretation in the Monotone Theory.
  Yotam M. Y. Feldman, Sharon Shoham, Mooly Sagiv, and **James R. Wilcox.**
  Principles of Programming Languages (**POPL**) 2022 (to appear).
- Induction Duality: Primal-Dual Search for Invariants.
  Oded Padon, **James R. Wilcox**, Jason R. Koenig, Kenneth L. McMillan, and Alex Aiken.
  Principles of Programming Languages (**POPL**) 2022 (to appear).
- Learning the Boundary of Inductive Invariants.
  Yotam M. Y. Feldman, Sharon Shoham, Mooly Sagiv, and **James R. Wilcox**.
  Principles of Programming Languages (**POPL**) 2021.
- Armada: Low-Effort Verification of High-Performance Concurrent Programs.
  Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, **James R. Wilcox**, and Xueyuan Zhao.
  Programming Languages Design and Implementation (**PLDI**) 2020.
  Distinguished Paper.
- Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations.
  Chandrakana Nandi, Max Willsey, Adam Anderson, **James R. Wilcox**, Eva Darulova, Dan Grossman, and Zachary Tatlock.
  Programming Languages Design and Implementation (**PLDI**) 2020.
- Inferring Inductive Invariants from Phase Structures.
  Yotam M. Y. Feldman, **James R. Wilcox**, Sharon Shoham, Mooly Sagiv.
  International Conference on Computer-Aided Verification (**CAV**) 2019.

- Functional Programming for Compiling and Decompiling Computer-Aided Design.
  Chandrakana Nandi, **James R. Wilcox**, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock.
  International Conference on Functional Programming (**ICFP**) 2018.
- Modularity for Decidability of Deductive Verification with Applications to Distributed Systems.
  Marcelo Taube, Giuliano Losa, Kenneth McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, **James R. Wilcox**, and Doug Woos.
  Programming Languages Design and Implementation (**PLDI**) 2018.
- VerifiedFT: A Verified, High-Performance Dynamic Race Detector.
  James R. Wilcox, Cormac Flanagan, and Stephen N. Freund.
  Principles and Practice of Parallel Programming (**PPoPP**) 2018.
- Programming and Proving with Distributed Protocols.
  Ilya Sergey, **James R. Wilcox**, and Zachary Tatlock.
  Principles of Programming Languages (**POPL**) 2018.
- Œuf: Minimizing the Coq Extraction TCB.
  Eric Mullen, Stuart Pernsteiner, **James R. Wilcox**, Zachary Tatlock, and Dan Grossman.
  Certified Programs and Proofs (**CPP**) 2018.
- Programming Language Abstractions for Modularly Verified Distributed Systems.
  James R. Wilcox, Ilya Sergey, and Zachary Tatlock.
  Summit on Advances in Programming Languages (**SNAPL**) 2017.
- Planning for Change in a Formal Verification of the Raft Consensus Protocol.
  Doug Woos, **James R. Wilcox**, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson.
  Certified Programs and Proofs (**CPP**) 2016.
- Array Shadow State Compression for Precise Dynamic Race Detection.
  James R. Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund.
  Automated Software Engineering (**ASE**) 2015.
- Verdi: A Framework for Formally Verifying Distributed System Implementations.
  James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson.
  Programming Languages Design and Implementation (**PLDI**) 2015.
- Automatically Improving Accuracy for Floating Point Expressions.
  Pavel Panchekha, Alex Sanchez-Stern, **James R. Wilcox**, and Zachary Tatlock.
  Programming Languages Design and Implementation (**PLDI**) 2015.
  Distinguished Paper.

Workshop Publications

- Verification of Implementations of Distributed Systems Under Churn.
  Ryan Doenges, **James R. Wilcox**, Doug Woos, Zachary Tatlock, and Karl Palmskog.
  Workshop on Coq for Programming Languages (**CoqPL**) 2017.
- Information-centric networking: Seeing the forest for the trees.
  Ali Ghodsi, Scott Shenker, Teemu Koponen, Ankit Singla, Barath Raghavan, and **James Wilcox**.
  Workshop on Hot Topics in Networks (**HotNets**) 2011.
- Intelligent design enables architectural evolution.
  Ali Ghodsi, Scott Shenker, Teemu Koponen, Ankit Singla, Barath Raghavan, and **James Wilcox**.
  Workshop on Hot Topics in Networks (**HotNets**) 2011.

Journal Publications and Theses

- Compositional and Automated Verification of Distributed Systems.
  James R. Wilcox.
  Ph.D. Thesis, 2021.
- Sets Characterized by Missing Sums and Differences in Dilating Polytopes.
  Thao Do, Archit Kulkarni, Steven J. Miller, David Moon, Jake Wellens, and **James Wilcox**.
  Journal of Number Theory, 2015.

- ShrinkWrap: Efficient Dynamic Race Detection for Array-Intensive Programs.
  James Wilcox. Stephen N. Freund, Advisor.
  Williams College Undergraduate Honors Thesis, 2013.

TALKS

- Intro to Dafny.
  Guest Lecture. University of Utah. Salt Lake City, Utah. September 2021.
- Transition Systems, Verification, and Dafny.
  Guest Lecture. University of Utah. Salt Lake City, Utah. March 2020.
- Verifying Distributed Systems with `mypyvy`.
  Seminar Talk. University of Utah. Salt Lake City, Utah. March 2020.
- Compositional Verification of Distributed Systems.
  Thesis Defense. University of Washington, Seattle, WA. May 2019.
- Goldilocks the Verification Engineer.
  Galois. Portland, OR. March 2019.
- Verifying Distributed Systems.
  New England Systems Verification Day. Cambridge, MA. October 2018.
- Compositional Verification of Distributed Systems.
  - Invited talk. Tel Aviv University, Israel. March 2018.
  - PLSE retreat. Leavenworth, WA. September 2016.
- VerifiedFT: A Verified, High-Performance Dynamic Race Detector.
  Conference talk at PPoPP. Vienna, Austria. March 2018.
- Programming and Proving with Distributed Protocols.
  Conference talk at POPL. Los Angeles, CA. January 2018.
- Programming Language Abstractions for Modularly Verified Distributed Systems.
  Conference talk at SNAPL. Monterey, CA. May 2017.
- Verdi: A Framework for Formally Verifying Distributed System Implementations.
  - Invited talk. University College London. United Kingdom. June 2016.
  - Invited talk. Google. Seattle, WA. June 2016.
- Planning for Change in a Formal Verification of the Raft Consensus Protocol.
  Conference talk at CPP 2016. St. Petersburg, FL. January 2016.
- Array Shadow State Compression for Precise Dynamic Race Detection.
  Conference talk at ASE. Lincoln, NE. November 2015.

Teaching Statement

James R. Wilcox

I view the challenge of developing and teaching a class as a *design* problem, where the student is viewed an active participant in the resulting system, whose experience must be carefully considered and centered during the design process. Every student is a unique individual who brings different skills and needs into the classroom. My experience in course design has led me to develop an approach that emphasizes tight feedback loops, places grades as subservient to feedback, and invites students to participate as active stewards of our shared intellectual heritage. Through these techniques, I aim to develop inclusive courses that invite all students into a community of learners that welcomes and supports them as they grow to carry our field into the next generation.

Designing Scalable, Interactive, and Adaptive Feedback Loops

In addition to being a computer scientist, I am also a student airplane pilot. When controlling either a computer or an airplane, feedback is essential. I cannot debug a problem if I have no information beyond "something is wrong", and I cannot fly an airplane without information about its orientation in space. As a teacher, I also know that feedback is essential to the student's learning experience. Students need to know where they are succeeding and where they are not in order to allocate their effort wisely. These two forms of feedback are related but distinct, and I believe good course design carefully considers them both, and their interactions. For example, if a student comes to me for help with a homework question, not only do I want to give them feedback on their current attempt, but I also want to give them feedback on their strategy, and to teach them where to look to get the feedback they need from the system. From the perspective of a student's entire experience in a course, they will get some feedback from course staff, and perhaps some from their peers, but they will get far more feedback from the tools and systems we ask them to use and interact with, because these tools and systems are inherently more scalable than finite course staff resources. As a result, selecting or designing tools that provide appropriate interactive and adaptive feedback, and explicitly teaching students how to use these tools effectively is essential to student success.

When I teach at the 300 level, aimed at a wide range of undergraduate CS majors or non-majors, I do a significant amount of live coding in front of students. I try to show them not just how to write the solution, but how they can derive the solution by working with the course tools effectively. I like to tell students that it's important to know when to turn your brain on and when to turn it off. Some problems are best left to tools rather than our brains. In the undergraduate programming languages class (CSE 341) at the University of Washington, we study both statically (compile-time checked) typed and dynamically (run-time checked) typed languages. In a statically typed language, the compiler will *guarantee* that our program doesn't accidentally use a number where a list was expected, or that we don't refer to a field that doesn't exist. When using this tool, we can safely turn that part of our brain off and trust in its feedback. For example, if we want to refactor a program in such a language to remove a field from a class, we don't need to worry about remembering exactly where all the references to that field are. Instead, just run the compiler, and fix all the errors it reports. As another example, in CSE 374 (non-majors introduction to systems), I emphasize liberal use of additional flags to the C compiler that turn on more warnings and enable sanitizers.

I don't draw a strong distinction between designing course materials and designing tools. Sometimes, the best way to teach a topic is to build a tool that let's students explore it. For example, when teaching CSE 490P (advanced undergraduate programming languages) and CSEP 505 (graduate programming languages), I found that students struggled with some of the theoretical languages that we study. These languages are unlike mainstream programming languages in that they are defined by a pen-and-paper description, and they are not executable, so students cannot try out example programs themselves without a great deal of cognitive overhead of churning through the pen-and-paper rules. To improve their experience, I either have students implement the languages themselves (490P), or I provide such an implementation for them (505, see https://jamesrwilcox.com/f/). Then I explicitly ask students to write a few programs in these languages and *interact* with these tools, which I have found helps students get comfortable with the concepts; it also relates the theory to students' previous programming experience more directly, and makes it more fun.

Grades are Second to Feedback

While interactive feedback from tools helps students on a minute-by-minute basis with their work, feedback from course staff and course peers at a coarser granularity helps guide students in developing their own sense of taste in terms of well written code or proofs. Traditionally, this coarse course feedback is communicated by, or at the very least conflated with, grades. I have been lucky to participate in many conversations about grading with my colleagues at the University of Washington, especially in CSE 590E, the computing education research seminar. Out of these conversations, I have experimented with several alternative grading strategies, with goal of moving grades out of the way of giving effective feedback.

When I taught CSE 374 in winter quarter 2021, I piloted a contract-based grading scheme, where each assignment asked students to write a program that exercised certain features of the C or Bash language that we were studying that week. We organized both peer and staff feedback on these assignments. All feedback was conducted through code reviews using Gitlab, and assessed solutions according to their understandability and whether they correctly used the desired features. From a grading perspective, I marked students essentially on a completion basis. Even though I told students that *any* program that met the contract would receive full credit, I was blown away by the creativity and effort that many students put into their programs. For example, several students implemented various kinds of games, some of them graphical, even though they had no prior experience with C, and the class did not cover graphics at all. On the other hand, an important hazard with this approach is that some students do not like to feel "forced" to be creative. To mitigate this, with each assignment, I also provided a few optional ideas for programs that they could write that would allow them to meet the contract.

Another instance of moving grades out of the way of feedback was my experience co-teaching CSE 505 and CSEP 505 with Zach Tatlock. We piloted an additive, points-based grading system. At the beginning of the quarter, we published exactly how many assignments of various kinds there would be, and how many points they would each be worth. We also published a formula for computing a 4.0-scale grade from the total points a student received over the whole quarter. We explicitly told students that they did not have to do all the assignments. They could select their desired target grade and decide which assignments, and how much of each, they wanted to complete. This gave students agency over their grade and course experience, and many students reported they appreciated the grading system in the course evaluations.

Active Intellectual Stewardship

I have always viewed teaching as intimately connected with research. In 2018, I applied to a few tenure-track positions before deciding that that route was not for me, and in my teaching statement at that time, I wrote that teaching is an integral part of my role as a scientist and researcher. Since then, I have gained a significant amount of additional teaching experience, and I now feel the same way as I did then, but from the other side. I view my ongoing participation in the programming languages and systems research communities as an important aspect of my teaching approach. I consider myself incredibly lucky to be able to learn and teach ideas from such a rich field, and it is my goal to pass on the best of these ideas to my students, and to support them participating in the community. This is a form of *intellectual stewardship*, in which I view my role as a maintainer and promulgator of these beautiful ideas.

My background is in programming languages (PL), and one of my favorite aspects of PL is how good theory leads to good code. For example, in CSE 490P (and, if I were to teach it at some point, CSE 401, compilers), an important learning outcome is the ability to read what are called "inference rules." These are a somewhat intimidating notation that is widely used in the PL community to write down inductive definitions. They are especially common when writing down type systems, which have a very deep theory. Through my participation in the PL research community, I have been lucky enough to learn a great deal of this theory, and I believe it can be made accessible to undergraduates in a way that benefits them. For example, one can specify the type system for the MiniJava language used in CSE 401 in just a page or two of inference rules. When students learn to read and fully understand this notation, implementing a typechecker in their compiler becomes "just" a matter of translating the inference rules to code.

More generally, I view the history of ideas as a sequence of steps that filters and polishes the best ideas in order to pass them on to wider and wider audiences. I try to invite students to join me in the process

of polishing these ideas by phrasing them in their own terms. Not a quarter goes by that I don't learn something more deeply myself because of a question a student asks, or from a connection they draw, or from the way they rephrase an idea. The resulting learning from and with students, not just teaching at them, is an aspect of my experience that I cherish.

Another aspect of stewardship is mentoring in both teaching and research. As a grad student, I had the pleasure of working with 7 undergraduate student researchers on various projects. I continue this aspect of my mentoring now through the SIGPLAN-M long-term mentoring program, in which I mentor 5 junior Ph.D. students from around the globe. As an instructor, I have worked with over 30 (primarily undergraduate) TAs, many of whom have gone on to graduate work in CS. I also consider myself a mentor to students in my classes. I am always surprised, for example, how many CS majors at the University of Washington don't know that Ph.D.s in computer science are fully funded, and I make it a point to mention this in every undergraduate class I teach.

My experiences as an instructor have been challenging but incredibly rewarding. By viewing course design *as design*, with the student experience centered, I strive to help students find the feedback they need, to move grading out of the way of good course feedback, and to walk with students on our journey as intellectual stewards of the ideas we are lucky to explore together.

Diversity Statement

James R. Wilcox

I sometimes forget the sheer magnitude of the number of lives I have impacted, even though I only have a few years of teaching experience. Students bring a diverse set of backgrounds, skills, and needs into the classroom, and their interactions with my courses are filtered through their learned experiences. In my classroom and in my mentoring relationships, I strive to create a welcoming environment in which everyone is valued for who they are. Such an environment cannot exist without conscious effort to mitigate implicit and explicit biases in myself, nor without an accurate analysis of the power structures that students and I must navigate. For my own part, I believe that I must start by acknowledging my own position of privilege as a white man in a field traditionally dominated by white men. It falls on me to help dismantle effects, both subtle and overt, that maintain this unacceptable status quo.

In my classroom, I try to create policies and materials that meet students where they are, and help them get to where they want to go. When I was a grad student, I sang in the University of Washington music department's chamber choir, and one thing I learned from the choral conducting professor there about how to rehearse a choir is that every singer might need something different to achieve the group sound. There isn't one piece of feedback you can give to the whole choir and get the desired result. My experience is that this carries over to teaching. For example, in many of the 300-level classes I teach, students can take the class immediately after CS2, or as late as their last quarter on campus. This incredibly broad set of backgrounds means that an assignment that one student finds easy, another student will find unbearably difficult. To mitigate this fundamental challenge, I try to design adaptive pedagogies, where students can succeed on their own terms, as I elaborate on in my teaching statement.

I also take these issues seriously beyond my own classroom, at the department, school, and university levels. During my experience as a graduate student at the University of Washington, I witnessed a handful of situations where explicitly biased positions, especially around the topic of the role of women in technology, were advocated for by certain faculty members. I was among a group of graduate and undergraduate students who used our union-bargained contract to file a workplace grievance with university labor relations, which described how these discussions created a hostile working environment. This grievance was successfully resolved with management, who agreed to meet with an advisory committee made up of graduate students to improve official email communication protocols, among other topics. Since then, the school has successfully implemented a new email and Slack policy that codifies pre-existing norms about what kinds of discussions are appropriate to have on mandatory work mailing lists. I was pleased with the way that management cooperated with students to resolve the grievance, and I look forward to continuing to make progress on creating a welcoming culture.

One of the primary features I am looking for in my search for a permanent teaching position is a university with a commitment to fight for equitable outcomes. By meeting students where they are, continually learning about my own biases, and analyzing power structures for their unintended systemic effects, I believe I can make valuable contributions in such a fight.

References

Prof. Zachary Tatlock
Paul G. Allen School of Computer Science & Engineering
University of Washington
ztatlock@cs.washington.edu

Prof. Daniel Grossman
Paul G. Allen School of Computer Science & Engineering
University of Washington
djg@cs.washington.edu

Prof. Mooly Sagiv
School of Computer Science
Tel Aviv University
msagiv@post.tau.ac.il