Teaching Statement

Robert Rand

Arnold Lebow had the best job at Yeshiva University when I was an undergraduate student there. In addition to being the longest-serving member of the mathematics faculty, he was the resident theorist of the computer science department. That meant that in addition to teaching the advanced calculus and abstract algebra series, he taught theoretical computer science and occasionally discrete math. And two of these classes set me down the path of studying computer science and teaching it.

Dr. Lebow's Advanced Calculus employed the Moore Method, a radical approach to teaching pioneered by the topologist Robert Lee Moore. At the beginning of each semester, he gave us a booklet of definitions and propositions, which we were to either prove or disprove to the satisfaction of our peers. Every day, students would arrive early to class to write down their proofs, and we would spend the class defending them, challenging them, and writing new proofs. When Dr. Lebow couldn't attend one class, I arrived to see two proofs of a theorem and one disproof. The two proofs were ones that I had attempted and abandoned, so I focused on the counterproof (all were ultimately shown to be flawed). The entire course was an eye-opening experience: I remembered what I learned and took away valuable lessons about students' ability to teach themselves, given the right structure.

By contrast, the success of Dr. Lebow's Theory of Computation course can be credited to a number of people: Turing, Kleene and the other fathers of theoretical computer science for developing such a compelling field of study, the students in my class for diving into it, and Dr. Lebow for his excellent (though traditional) approach to teaching it. But I think the most credit goes to Michael Sipser, the author of Introduction to the Theory of Computation. Sipser's book is a masterwork. It takes students on a journey from Automata Theory to Computability Theory to Complexity Theory, with every chapter motivating the next and drawing examples from previous ones. Each homework problem serves a purpose, deepening the reader's understanding and sometimes introducing them to new problems. Ever since I took this course, I've thought about how I would teach it, how I would explain automata theory in terms of the many automata we see every day (even as they are replaced by Turing machines) and how I would tie complexity theory to real problems and computability theory to the limits of the knowable.

And so, when Mike Hicks offered me a postdoc at the University of Maryland, my immediate response was "Would I be able to teach Theory of Computation?" (I had five exciting postdoc offers, so I felt like I had some leverage.) "Wouldn't you rather teach this?" Mike responded, with a pointer to Maryland's course on Program Analysis and Understanding. To give context to my response, I have to briefly discuss the greatest textbook the world has ever known and the future of teaching.

Software Foundations is a four-volume series on Formal Verification and Programming Languages. It is also a series of lecture slides, replete with "poll the audience" style quizzes to ensure class participation. It is also an executable program in the Coq proof assistant. To be more precise, Software Foundations is a library of Coq files, containing exposition, quizzes, slides, exercises, unit tests, solutions, and proofs to be done in class, along with a set of toggles for what an instructor

wishes to provide for their students and in which format. (Typically, instructors will provide one set of Coq and HTML files for following along in-class and another for after class.) As a proof assistant, Coq checks every proof written in it for correctness, so while students may not be able to complete a given proof, they cannot get it wrong. This facility, in addition to the utilities and template Software Foundations provides, is why so many teachers are excited about extending the series to cover subjects like Discrete Mathematics and Algorithms, Random Testing and Probabilistic Programming, and (in my case) Quantum Computing.

With that context, I accepted Mike's offer and immediately set about remaking Software Foundations in my image. Having taken this course with the lead author (Benjamin Pierce, a brilliant pedagogue), I had a high bar to clear and wanted to play to my strengths. In particular, I love live coding and live proof, having honed these techniques during my Python minicourse at Penn and my Coq tutorials at the International Conference on Functional Programming (2015) and Principles of Programming Languages (2016). Expanding the number of programs and proofs that we worked on together in class helped me keep the class engaged while checking that students were keeping up. I also believe strongly in making things as transparent as possible. Coq is built on the observation that a powerful enough type system can encode arbitrarily complex mathematical statements. Coq "proofs" are just programs that have the desired type, though we tend to write them in Coq's proof interface, which hides programs behind mathy words like "induction" and "contradiction". Software Foundations doesn't pull back the curtain until late in the first volume and many instructors just leave it in place. By contrast, I explained Coq's proof system via examples starting in the first lecture and I think it gave students had a better understanding of what they were doing. I'm currently working with the authors to incorporate my new material into the series, even if not in the first lecture, and I've trialed a new version in this semester's version of the course (taught by Leonidas Lampropoulos).

I also had the chance to use some wholly original material in Program Analysis and Understanding: my textbook-in-progress, Verified Quantum Computing. VQC draws on my work on the QWIRE and SQIR quantum programming languages and aims to teach quantum computing through formal verification. It works: Quantum computing is so mathematically rich that students *want* to convert quantum programs into mathematical representations and analyze them. Conveniently, VQC is also a nice entry point to my research agenda, which revolves around formally verifying quantum systems. I'll be presenting VQC as a tutorial at this January's Principles of Programming Languages and at WiSQCE, the Winter School on Quantum Computing at Emory.

Another valuable application domain for proof assistant-based teaching is a Discrete Structures course, following an approach pioneered by Michael Greenberg at Pomona College. Such courses typically cover propositional and predicate logic, proofs and induction, combinatorics and probability theory, and perhaps a taste of graph theory or computability theory. Propositional and predicate calculus beg to be taught in a proof assistant, which immediately draws attention to logical errors. Proof assistants like Coq are even more valuable for teaching induction, which many students struggle with. But computer scientists understand recursion (and it's a crucial subject for those who don't) and what is an inductive proof but an application of recursion? As part of the modifications I discussed above, I revised Software Foundations to explicitly teach *recursive proofs* and have students define induction principles (over numbers or lists or even predicates) using recursion. Bringing concepts together in this way, and watching them click in the minds of students, is my goal and purpose as a teacher.

I'm not wedded to a specific tool, though: I don't see how we can teach graph theory in a proof assistant, so unless a compelling story appears, I won't attempt to. Graph theory asks for circles on a whiteboard connected by lines, and I'm happy to draw those circles and explore what we can do with them. I have experience with this approach: I've taught Discrete Structures

on many occasions, starting in 2009, mostly before I knew about proof assistants. In the spring of 2010, I was the recitation instructor for Discrete Structure at Yeshiva University. The new professor (unbeknownst to the department) was suffering from early-onset Alzheimer's and I had to teach much of the material during recitation. This was less than ideal for the students, but I am glad that I was in a position to salvage the course and I benefited from the experience. I've also taught the Discrete Math lecture at Penn, substituting for Val Tannen. Stepping into a lot of shoes means getting a lot of perspective on how to teach this important material, which is why I frequently volunteer to substitute or guest lecture (most recently for Xiaodi Wu's graduate course in Quantum Information Processing course at Maryland).

Teaching subject matter as diverse as Discrete Structures, Formal Verification and Quantum Information Theory has allowed me to make new connections between subjects and convey powerful intuitions to my students. These, along with the Theory of Computation course I still haven't taught, are the kinds of courses I am delighted to teach, and in which I hope to further hone my teaching skills.