Most of my teaching experience to date has come from a single class: UIUC’s CS 421, Programming Languages and Compilers. The course is wide-ranging, covering functional programming in OCaml, the basics of compiler design, programming language semantics, and a taste of formal methods in the form of Hoare logic. Having TA’d the course three times under two professors, when I first volunteered to teach it over the summer, I thought I knew the material fairly well. It wasn’t until I’d been primary instructor for a few weeks, though, that I realized that what I truly wanted to communicate wasn’t any of those subjects: it was a particular thought process that underlay all of them, characterized by phrases like “first find a rule that matches, and then apply it” and “what allows us to take the next step?”. The core purpose of computer science classes, I have come to believe, is to teach ways of thinking and problem-solving skills. I aim to teach not content, but an approach, one that is useful throughout computer science and beyond.

The two courses I taught in grad school – CS 173, Discrete Structures, and CS 421 – were both required courses for CS majors, and were difficult for many students. Both courses revisited familiar areas – basic math in 173, programming in 421 – and challenged what students thought they knew, with the goal of instilling a deeper understanding. The first time I asked a roomful of freshmen “Is zero even?” and got four different answers, it was clear that knowledge is not the same as understanding. By encouraging them to look for an explanation rather than an answer, I turned a seemingly trivial question into a tool for reshaping the way students thought about math. Teaching functional programming presented the same challenge. Many students felt as if we were asking them to program and then crippling them, taking away access to global variables and loops, and questioned why they would want to write code by such a convoluted method. There are some good arguments for using functional languages in the real world, but we also teach them precisely because of that method: the most valuable lesson of OCaml is not neat tricks for processing lists (though students do learn those, too) but the way of thinking it requires, the discipline of strong type systems, the effectiveness of recursion and divide-and-conquer as problem-solving strategies. Students who truly learn functional programming become better imperative programmers as well – and better mathematicians, better logical reasoners, etc.

This philosophy is as good as far as it goes, but it raises an important question: how do we transfer understanding of underlying principles? My answer involves clear presentation and careful assessment, but most of all, it hinges on interactivity. I believe that the best way to tell how much students understand is to hear the questions they ask, and the best way to help them understand more is to answer their questions, carefully and in detail. As the instructor for CS 421, which had both on-campus and online students, I made an effort to provide as many means as possible for students to ask questions. I was always happy to stop a lecture for a question, and made sure to repeat the students’ side of the exchange so that the online students could follow when watching the recording later. I encouraged as many students as possible to come to office hours (separate Skype office hours were set aside for online students), and offered one-on-one question-and-answer sessions by appointment. My greatest success was through Piazza, an online discussion board where students could not only post their questions but also answer each other’s. I made a point of being active and responsive on the board, occasionally surprising students with how quickly an answer appeared. I knew I’d truly managed to convey understanding, though, whenever one student explained to another the step they’d missed. I
always learn more about a subject when I teach it, and Piazza gave students the chance to learn in the same way.

Teaching a part-online class came with challenges as well. As part of the Graduate Teacher Certificate program, I met regularly with a teaching expert during the course. One of the issues we discussed was the steady decline in attendance: the recorded lectures were available to the on-campus students as well, and many students preferred to watch them from the comfort of their homes. My mentor encouraged me to think about whether and why this was a problem, and what I could do to address it. I worried that the fewer students came to class, the fewer questions would be asked, and the less I would be able to adapt my lectures to the class’s needs. In this instance, because of Piazza, students always had a chance to ask questions, and I had the opportunity to re-explain topics in different ways after the lecture. I also thought about what I would do to encourage attendance in a class that didn’t have to be equally accessible to online students, and how I could make coming to lecture worthwhile. In future courses, I plan to put more emphasis on solving problems in class, looking for a sweet spot between lecturing and office hours, to make sure that students are actively benefiting from my presence and from time spent in class.

This, then, is my teaching philosophy: that my goal is to convey understanding of the logical systems that underlie topics in computer science; that I can achieve this by giving students as many chances as possible to ask questions and discuss topics, in person and online, with course staff and with each other; that time spent in class should be valuable to students, and it is my responsibility to make lectures worthwhile. I look forward to bringing this approach to courses at all levels, and while my experience has been with the more mathematically oriented computer science classes, I believe it is more broadly applicable as well. (How should I refer to this?) Several professors in software verification, including Steve Zdancewic, Andrew Appel, and Adam Chlipala, have advanced a proposal to develop project-based courses in compilers and operating systems using interactive theorem provers, with the goal of highlighting the rigorous mathematical logic behind common programming projects. Whether by incorporating this methodology into my own teaching, or simply using it as a guide to the underlying principles of a wider range of CS topics, I would love the chance to build on this proposal and bring my approach to less traditionally rigorous classes. Formal methods may sometimes be intimidating to students, but with clear explanations and interactive discussions, they can help us reach beyond knowledge of facts to deeper understanding.