Teaching Statement

Mukund Raghothaman

I consider teaching to be one of the most rewarding aspects of life in academia, and also one of the most challenging. At the lowest level, I expect students I teach and mentor to leave with knowledge of concrete concepts such as programming abstractions, invariants and assertions. At a higher level, true success as a teacher will only be achieved if I have affected the way they think: if an undergraduate student who proceeds to a career as a software engineer pauses to think about program correctness, if an undergraduate student proceeds to graduate school to do research, if a graduate student comes up with an elegant and rigorous solution to a research problem, and go on to develop good taste in choosing her own research problems, and if each student leaves the classroom with a sense of having learned something exciting.

Teaching experience. In graduate school, I have been a TA for two courses: an undergraduate course on the theory of computation, and Software Foundations, a graduate-level course on programming languages. My duties included conducting office hours, answering questions, and helping in setting and grading assignments and exams. I also contributed material to the Software Foundations textbook, primarily through a formalization of information flow security, which we unfortunately could not cover in class due to lack of time.

Teaching philosophy. My main goal will be to make courses challenging while still remaining approachable. Students are capable of far more than they believe they are, and I think that it is my role, as a teacher, to help them achieve their full potential. An important part of this is to foster in them the confidence and curiosity to approach new material. It is only by discussing new theorems and techniques, and challenging peers on their misconceptions, that familiarity is achieved, and the “foreignness” of concepts eliminated. I am aware that years of learning about and doing research in the field make it difficult for me to judge the inherent difficulty of concepts when first encountered by students. I will therefore make a conscious effort in breaking down concepts and solicit active, continuous feedback from students to make sure that my classroom style is effective. Unfortunately, I am also aware that this is easier to achieve in a small classroom than in the large hundred-student classrooms commonly seen in core undergraduate courses. Within the classroom, I will attempt everything possible, such as the use of anonymous response tools such as clickers, to encourage participation and make learning interactive. Outside the classroom, such as for assignments and exams, providing meaningful feedback for assignments and exams is hard, because the student cannot judge the correctness of their solutions, and cannot be sure that their understanding is complete even if they successfully solved all the assignments. I witnessed one solution to this problem in Software Foundations, the graduate programming languages course I attended and for which I was later a TA: because the entire course was formalized within an interactive theorem prover, Coq, students could actually judge the correctness of their assignments themselves, by simply getting Coq to print “No more sub-goals”. While encouraging experimentation and innovative solutions to assignments, this approach might be more unforgiving than is desirable for most other courses, especially at the undergraduate level. I believe that these ends can be at least somewhat achieved by new social mechanisms such as double-blinded peer review of assignments and exams.

Courses on programming languages and formal verification have traditionally been important in challenging students to think deliberately about software correctness. Another reason these courses are becoming relevant is that tools such as SAT and SMT solvers are achieving industrial robustness, and can be very useful to practicing computer scientists. Many modern package managers are based on SAT solvers, and simple applications to prove their utility—Sudoku and Slitherlink solvers, for example—are readily available. I look forward to
integrating these techniques into existing courses, or designing new courses on program synthesis to expose students, both at the undergraduate and graduate level, to the enormous utility of modern constraint solvers.

**Research advising.** I look forward to mentoring students as they begin their own research careers. As a mentor, I would emphasize the importance of good problem phrasing and the rigorous evaluation of proposed solutions. Especially for undergraduate students, I view research as a way to deepen one’s knowledge in a self-directed manner. As I look back at my own career in graduate school, I appreciate the freedom that my own advisor afforded me through the years, only actively stepping in when I seemed to be stuck on unproductive approaches. I hope to follow a similar style, where my role as an advisor is simply to perform periodic course-correction, and introduce the student to related problems and potential solution strategies, while leaving them fully in charge of the direction of the research project.

**Potential courses.** I am excited to teach undergraduate courses on the theory of computation, discrete mathematics and logic for computer science, programming languages, compiler design, and introductory courses such as an introduction to computer programming. At the graduate level, I will design and teach courses on program analysis, formal verification, and program synthesis. The course on program analysis would cover classic ideas such as Hoare logic and abstract interpretation, temporal logic, model checking, and applications of automata theory, and recent ideas such as hyper-properties and applications to information flow security. The program synthesis course would be more exploratory, starting from the synthesis of reactive systems, and proceeding to recent developments such as counter-example guided inductive synthesis, uniform solver formats such as SyGuS, and with brief excursions into program synthesis with vague and underspecified user inputs.