I want to leverage my applied computer science research background to develop and evaluate new technologies for computer science education. Education is a broad topic of global importance, and it is being addressed from across many academic disciplines. As a computer scientist, though, what excites me most is how recent developments in free online learning resources (e.g., Khan Academy) and Massive Open Online Courses (e.g., Coursera, edX, Udacity) have revitalized public interest in and grant funding for educational technology. However, I am dismayed by the lack of deep innovation from these efforts, which seem more focused on growth—more courses, more videos, more quizzes, more partners, more student sign-ups—instead of on radical advances in technology to empower learners.

I am interested in educational technology in general, but I want to start with CS education both due to personal expertise and to the increasing importance of teaching computational thinking as a fundamental kind of literacy akin to reading, writing, and mathematics. As someone who is grounded in applied CS research, I am in a unique position to make substantive advances across areas such as human-computer interaction and program analysis, all guided by a motivation to improve CS education.

My Research Background
Although my formal training was not in education, I will first introduce my research background to frame the later discussion of how I plan to use these skills in my CS education research agenda.

My Ph.D. dissertation was on tools to improve the workflow of computational scientists and data analysts, and my Master’s thesis (which won the annual MIT EECS department thesis award in 2006) was on a framework for detailed run-time instrumentation of C and C++ programs. These projects have generated almost two dozen publications (see CV for a full list). According to Google Scholar, my current h-index is 12 (12 publications each with at least 12 citations) and total citation count is 948.

My research experience spans four main areas: scientific data provenance, dynamic program analysis, human-computer interaction, and empirical software engineering.

Scientific Data Provenance: My Ph.D. dissertation presents new tools to help computational scientists organize, annotate, share, and reproduce their experiments. I created Burrito [10], a Linux-based tool that automatically logs the user’s computer activities, provides interfaces to annotate the captured provenance with notes, and then answers user queries such as, “Which script versions and command-line parameters generated the output graph that this particular note refers to?” I created another provenance-capture tool called CDE [5, 9], which largely eliminates “dependency hell” on Linux and makes it easy to deploy and reproduce computational experiments; over 10,000 people have downloaded the CDE binary.

Dynamic Program Analysis: I have developed several program instrumentation tools to collect, analyze, and alter run-time behavior. As part of my Ph.D. dissertation, I built a provenance-tracing Python interpreter that performs automatic memoization and dependency resolution [8] and another interpreter that adds run-time error tolerance and recovery to data analysis scripts [4]. As part of my Master’s thesis, I developed a dynamic analysis that infers programmer-intended “abstract types” from low-level concrete types in C and C++ programs [11].

Human-Computer Interaction: I have worked on research projects that were published at top-tier HCI conferences such as UIST and CHI. For another part of my dissertation, I developed a recommendation system for an end-user programming tool for data analysts [12]. Earlier in graduate school, I helped design a lab study that investigates how modern programmers use web-based resources [3]. In addition, HCI and user-centered design methodology crosscut many of my other projects, such as Burrito [10].

Empirical Software Engineering: I have performed mixed methods empirical research to investigate software engineering questions using a combination of questionnaires and large-scale data analyses. In 2009, I published a study of how Linux kernel developers responded to bug reports issued by the Coverity static analysis tool [7]. With colleagues at Microsoft Research, I investigated social and technical factors that influence whether software bugs in large commercial codebases such as Microsoft Windows are fixed [13], reassigned [14], or reopened [21] (winner of Best Paper Award).

Although I am open to collaborating on and extending these existing lines of work, my main passion is to apply the technical expertise and research techniques I have learned to innovate in education.
Intermezzo: Online Python Tutor

Before presenting my research agenda, I want to describe what got me passionate about building tools to improve CS education in particular. In 2010, I created a web-based program visualization tool called Online Python Tutor [6] (pythontutor.com). Over the past three years, this tool has attracted over 200,000 users, been incorporated into four academic research projects [1, 17, 18, 19], and been used in:

**Traditional classrooms:** Professors and teaching assistants in over a dozen universities have used Online Python Tutor during lectures and lab sessions. Class sizes range from 7 students in a Python-based summer CS0 course for non-CS majors at the University of Washington to over 900 students in the Fall 2012 offering of CS1 at UC Berkeley (called CS 61A). Instructors have told me via email that Online Python Tutor clarified concepts such as tracing parameters and return values through function calls, control flow, exceptions, scope, recursion, local variable lifetimes, and foundational CS threshold concepts [2] such as pointer aliasing and pointer-based parameter passing.

**Digital textbooks:** Three web-based textbook projects have incorporated Online Python Tutor visualizations. The CS1 instructor at UC Berkeley recently ported the classic Structure and Interpretation of Computer Programs textbook from Scheme to Python and put it online in HTML format. He embedded Online Python Tutor visualizations directly into the HTML webpages so that students can simultaneously read lessons and interact with running code on the same page (zoom in on figure to the right). Similarly, Miller and Ranum embedded Online Python Tutor into their popular digital textbook, How to Think Like a Computer Scientist: Interactive Edition [17], which attracts around 6,000 viewers per month and has been adopted by 25 universities. Lastly, the CS Circles digital textbook [19], which receives 10,000 visitors per month, has integrated Online Python Tutor as its visualizer.

**Online courses:** Another major source of users is students taking online CS courses. For instance, hundreds of discussion forum posts on Udacity courses such as CS101 (~186,000 students over two offerings) and CS212 (~40,000 students) make reference to Online Python Tutor. Similarly, it is listed as a student resource on the course wiki for 6.00x, the CS1 course from edX. Instructors of the Fall 2012 offering of CS1 from Coursera (~34,000 students) customized and deployed their own version of Online Python Tutor on their servers; they also used it in some of their recorded screencast lectures.

Online Python Tutor is not a “research-worthy” project on its own; after all, ideas in program visualization are decades old. However, its popularity shows the demand for online CS education technology and enables me to use it as both an intellectual and technical platform for launching new research.

Research Agenda

I want to build systems that enable CS students to better acquire, share, and retain knowledge. My applied CS research background [4, 5, 8, 9, 10, 11] enables me to make substantive technical innovations. And my HCI and empirical studies training [3, 7, 12, 13, 14, 21] enables me to evaluate the systems I build to ensure that they actually improve student learning rather than simply being technically interesting.

**Facilitating conversations around code:** Inspired by the power of peer teaching (see my Teaching Statement), I want to build tools to facilitate conversations around Online Python Tutor code visualizations. Researchers have already established the pedagogical benefits of in-context discussions within course materials [22] and data visualizations [15]; I want to transfer those ideas to CS education.

The base component here is an authoring environment for inserting annotations and interactive prompts directly into code visualizations; see figure on the right for an early prototype. These kinds of question and discussion prompts encourage active engagement (in constructivist style) rather than passive viewing, which is known to make visualizations more effective [16]. Even if textual answers are never graded, studies in cognitive science have shown that the mere process of reflecting on and explaining concepts to oneself can deepen understanding [20].

This sort of authoring environment can also replace PowerPoint and screen video capture for creating CS lecture presentations involving code, algorithms, or data structure visualizations. Students can
interact with, comment on, remix, and share lectures produced in this rich, code-aware format.

When this technology is used to build a discussion forum (a better Stack Overflow), program analysis can enable contextual searches like “what other discussions were about this kind of semaphore use idiom?”

**Collaborative visualizations:** Adding real-time concurrent editing to these visualizations enables students to work together and tutor one another while physically separated. A shared workspace can also be useful during both physical and virtual lectures: As the instructor is lecturing using code visualizations, students can follow along by viewing those same running examples on their laptops. At any point when curious or confused, students can instantly diverge from the live example and try out their own variants on-the-fly in private; they can then sync back to the “live feed” at any time. Students can also text chat with one another about the lecture, all within the context of the live lecture materials. If students opt-in to allowing the instructor to access such interaction data, then that could “close the loop” and help the instructor improve future lectures. For instance, if 80% of students are silently diverging from the live example at a certain point in lecture, then perhaps that part requires further clarification.

**Pedagogical code instrumentation:** Researchers have traditionally used compile- and run-time code instrumentation for debugging, testing, profiling, and optimization. Instead, I propose to instrument programs written by students to collect data that helps both instructors and students, using similar technical machinery to “optimize” for student learning rather than, say, performance or reliability.

For example, instrumentation data could help instructors improve their programming class assignments. Right now, it is hard to develop high-quality, bug-free assignments since instructors cannot easily receive accurate, fine-grained feedback from students. Even when students offer feedback, they often cannot articulate the root cause of their misunderstandings. Aggregated analytics data could automatically report trouble spots such as “75% of executed student code violated abstraction barriers when accessing class Foo.” This information might alert instructors to clarify assignment wording or to re-emphasize particular concepts in future lectures. Also, teaching/lab assistants can use analytics data to get “knowledge profiles” for specific students and then deliver more targeted one-on-one tutoring.

**Beginner-friendly visual error reporting:** Compilers, interpreters, and libraries are not designed for beginner-friendly error reporting; thus, cryptic error messages consistently demoralize beginners. Analyzing a large corpus of student code (e.g., over 200,000 code snippets from pythontutor.com logs) can yield insights into common novice mistakes. These patterns can inform the design of compiler frontend and run-time instrumentation to collect fine-grained provenance on error paths. Also, program evaluation semantics might need to be altered to better support CS pedagogy. The open HCI problem here is how to display errors in a beginner-friendly format, tightly integrated with code visualizations.

**Crowdsourced creation of high-quality programming tutorials:** There is no shortage of “learn to code” materials on the web, but quality is often inconsistent and difficult to assess due to lack of fine-grained feedback. Imagine a Wikipedia for CS materials where people can read lessons, run code examples live within a web browser, annotate visualizations when confused or enlightened, and then use both discussions and analytics to improve those materials. This process might start by first having enthusiasts find high-quality blog posts and code snippets on the web and then incrementally rewriting them into a consistent format. Reader feedback then closes the loop, bootstrapping further refinements.

**Next-generation digital textbooks:** The term “digital textbook” used to simply mean converting paper books into PDF and HTML formats. Current efforts such as CS Circles [19] and my collaboration with Brad Miller [17] are adding interactivity in the form of in-context visualizations and assessments. While this is a promising step, I have greater ambitions for the next generation of digital textbooks.

First, they should be **adaptive**, providing students with personalized, nonlinear, hyperlinked paths through the lessons and exercises. Specifically, if the system contains a pedagogically-useful model of the programming language it is teaching, then it can analyze student code to detect knowledge gaps (e.g., “seems like you’re struggling with breaking early out of for-loops”) and suggest remediation lessons.

Next, textbooks should be **personalizable**, allowing students to write “margin” notes and annotate the results of experimenting with interactive activities (e.g., playing with machine learning algorithms on a data set); this is like Burrito [10], except for students rather than scientists. By the end of a course, each student’s textbook copy will be uniquely theirs, reflecting their particular learning experience. Personalization not only reinforces learning but can also be another source of feedback to the author.

Lastly, textbooks should provide **visibility** into the author’s thought processes. Students can benefit from seeing all of the author’s false starts, failed drafts, and alternative explanations—not just the polished final copy. The open problems here are how to design an authoring environment that captures such provenance as well as a user interface for effective browsing.
References


