Statement on Scholarly Activities

Geoffrey Challen

To support my goal of giving everyone an opportunity to learn the basics of computer science and programming, I create, deploy, and maintain novel educational technologies—including systems for interactive content delivery, assessment, autograding, online tutoring, plagiarism analysis, state management, and data collection. Each system I have created serves a well-articulated pedagogical goal, and multiple tools are unique to my introductory computer science (CS1) course. I describe several of the most innovative in more detail below.

Interactive Walkthroughs

Demo: https://www.learncs.online/best#interactive-walkthroughs

Prior to the COVID-19 pandemic, live coding played an important role in classroom instruction in my CS1 course. Live coding allows students to observe and participate in the process of solving computational problems, and allows the instructor to externalize their thought process. It is considered by many to be a best practice when teaching students to program. So when migrating my CS1 materials online to support remote instruction during the pandemic, I knew that I wanted to preserve this component of our instructional approach.

A common way to do this is by posting a video showing the instructor interacting with the code. While workable, this approach has limitations. One of its biggest drawbacks is that the code the instructor is working with is not available to the students throughout the demonstration, since what is delivered to the display is just pixels. The aha moment for me came when I found myself watching a video live coding demonstration and inadvertently clicked on the screen to try to edit the code the presenter was manipulating. Of course, that only caused the video to pause. But I knew that I could create a way to preserve the ability to interact with the code during the live coding explanation.

What emerged from this observation is an instructional component I call an *interactive walkthrough*. An interactive walkthrough presents like a live coding video—students click a button, and then the code starts changing, accompanied by an audio explanation. However, the component is not a video: it's an animated editor, replaying a trace of changes made to the editor during recording by the presenter. Instead of the decontextualized pixels delivered by a live coding video, students have the actual code the instructor is working with in front of them at all times—free for them to experiment with. On my CS1 site, every interactive walkthrough is connected either to our Java and Kotlin playground backend—allowing students to run the code and examine the output—or to our homework autograder—allowing students to submit the code and examine the grading results.

Interactive walkthroughs have become a central feature of our successful online CS1 materials. Each daily lesson contains multiple short interactive walkthroughs, interspersed with text, and other interactive lesson components: playground examples, practice problems, and debugging exercises. Interactive walkthroughs allow us to use video sparingly: only when an explanation requires that we use some non-code component, like a whiteboard or prop.

One of the more exciting opportunities provided by interactive walkthroughs has been the chance to diversify the course's voice. As a white, male, cisgender, heterosexual, able, INTJ computer scientist, I'm very aware of the degree to which I fail to represent the diversity of computer science. While moving away from a lecture model of instruction was prompted by the pandemic, I have embraced the opportunity to invite others into our instructional community and allow my students to hear from a more diverse set of voices. Interactive walkthroughs have proven to be an ideal platform for encouraging contributions from course staff. Most are short, and recording is done entirely *in situ*—right in the browser, with the surrounding lesson material available for context.

Since Fall 2020, almost 300 people have contributed content to my CS1 course by recording at least one interactive walkthrough—including three instructors from two different universities, but mainly drawn from my course staff. Most concepts now have at least two explanations—one from an instructor, and one from a course staff member. Course staff who are mostly former students may lack the deep content knowledge of an instructor, but they can bring a complementary perspective to the material, and may end up providing the analogy or mental model that sticks for a particular student. In addition, our Java materials now almost all have explanations from at least two

instructors: myself and Colleen Lewis, who recorded a complete set of walkthroughs for Java in Fall 2021. A student that doesn't quite understand my explanation can review Colleen's, and then possible one provided by a staff member. I'm also collaborating with Luc Paquette and a graduate student to analyze how students develop preferences between the two instructors. Preliminary results support the hypothesis that having content from multiple instructors available helps support student success in the course, and that students do develop preferences for different instructors in ways both predictable and surprising.

Homework Autograder and Code Quality Analysis

Demo: https://www.learncs.online/best#homework

Students in my CS1 course benefit from lots and lots of practice. In the past some even requested additional practice problems—and every instructor will remember when a student asks for more work! In addition, as we began asking students to complete programming tasks on our weekly quizzes, it became important to provide a fresh stream of programming challenges for each quiz—updated each semester to avoid the inevitable problems with information sharing about the assessment—but also to continue to build up a library of problems for students to use for practice.

I authored our first set of daily homework problems in Fall 2018. It was a laborious process, usually requiring several hours to author a single exercise. The root of the problem was caused by the need to write test suites for each exercise to allow the autograder to establish correctness. Creating strong test suites is time-consuming, particularly if you aim to test all corner cases and possible failure modalities, and to provide enough inputs to prevent students from "solving" the problem through pure memoization. When authoring test suites for autograding, I frequently found myself embedding small mini-solutions inside my testing code. For example, to test code that returns the maximum value from an array, I would embed the correct code in the test suite, create a bunch of random arrays, and then compare the behavior of the solution and the student submission.

This caused me to identify a core difference between software testing and autograding—when using an autograder, the solution is known! This is not normally the case when writing software tests, where only the desired *behavior* of the solution is known. Instead of maintaining three sources of truth—the description, the reference solution, and the test suites—it should be possible to only provide only the description and the solution, and allow a system to generate a testing strategy automatically by observing the solution behavior.

The system we built to support rapid question authoring is called Questioner. Question authors provide a description and a solution. In many cases, Questioner can generate a robust testing suite with only that information—and without requiring a single test case. However, when inputs with special properties are needed, authors can provide input generators. Provided with the solution, Questioner does not only generate a testing strategy—it also validates that its strategy is effective by ensuring that it can reject incorrect submissions. To automatically generate incorrect submissions, Questioner performs source code mutation on the solution. For example, if the solution uses a strictly less-than comparison (<) Questioner will mutate that to less-than-or-equals (<=), and ensure that the testing strategy it has created can catch the error, and repeat this process for every mutated solution. We have a large and robust library of mutation strategies, and can usually generate dozens of incorrect submissions from the reference solution. Question authors can also provide explicit examples of incorrect submissions that should be rejected, but as our mutation library has improved the need for this has diminished, further accelerating the authoring process.

Overall Questioner allows rapid authoring of strong Java and Kotlin programming problems. It used to take several hours to write a single problem with unknown accuracy. Now I can author several highly-accurate problems in a single hour, which can be deployed on a timed assessment like a quiz the next day after a single round of staff review to ensure the description matches the solution. Since Fall 2020, I have used Questioner to write 700 Java and Kotlin programming exercises covering everything from first-day single-line "Hello, world" problems to graph traversal and recursive algorithms on binary trees. Each daily lesson has one graded homework problem, which I replace annually, and multiple practice problems. Each quiz also has several programming challenges, which I replace every semester. Old homework and quiz problems get moved to a practice page, where students now have many opportunities to test and reinforce their understanding of the concepts taught on the daily lessons. Because Questioner's validation process ensures that the resulting testing strategy is effective, it can allow course staff members or others with less experience with software testing to author equally-accurate problems, and our problem library includes contributions from several undergraduate course staff.

The solution-driven approach also allows Questioner to evaluate a large and growing number of code quality attributes—to assess not only that a submission works, but how good it is. One oft-cited weakness of autograding is that students do not get human feedback on their programs, and so may develop solutions that, while correct, are low quality. However, evaluating code quality requires a lot of experience or training, providing human feedback is time-consuming, and that feedback often arrives too late for a student to learn from.

Our goal with Questioner is to automate as much code quality feedback that might be provided by a human grader as possible, so that we can provide students with instant and insightful feedback on their efforts while freeing the course staff to focus on tutoring and other forms of direct student support. We began by measuring cyclomatic complexity, a measure of the number of code paths through a program that is easy to compute and correlates with the cognitive load required to understand a piece of code. In our experience, it does a good job of identifying overly-complicated submissions. When a student submits code that has many more code paths than the solution, we either provide a warning (early in the semester), or deduct a small amount from their score. In either case, we encourage them to simplify the code and resubmit it for full credit. (We always give students a chance to correct mistakes without penalty.)

Through deep integration with the Java Virtual Machine, and working with a former student talented in bytecode manipulation, we are now able to efficiently measure multiple aspects of code quality—including style and formatting compliance, cyclomatic complexity, dead code detection, execution efficiency, and memory utilization—all of which are being used to evaluate thousands of student submissions per day on my CS1 website and public materials. We are currently working on more advanced code quality analysis techniques, including approaches that utilize the syntactic features used by the submission, O(n) complexity categorization, and variable name analysis. Students in my CS1 course always have the option to receive in-person help with their code at the push of a button from dawn until dusk. But by pushing the limits on what aspects of code quality can be evaluated automatically, we not only provide students with immediate and accurate feedback, but also reduce the tutoring load on our course staff, helping ensure that students that do need human support can obtain it promptly.

Debugging Exercises

Demo: https://www.learncs.online/best#debugging-challenges

Students learning to program will naturally make mistakes. My CS1 course is pro mistakes! We remind students that, if they aren't making mistakes, they aren't learning—an axiom of learning. By making mistakes, students learn to recover from them and gain confidence in their abilities. In the past some students have requested more practice with identifying and correcting mistakes. They may find themselves confronted with an error on a timed assessment and be unable to correct it before time expires. It became clear that, for some CS1 students—and probably for all of them—they aren't naturally making enough mistakes to learn to correct them. If we could give them more practice fixing mistakes, they'd be able to get out of trouble more easily and complete programming tasks more confidently.

Our approach to generating debugging exercises was inspired by Questioner's validation process, which uses source code mutation on the solution to generate incorrect submissions. However, rather than just use the reference solution, our tool uses *all* the correct submissions for a given question submitted by previous students, resulting in a much larger set of incorrect examples. For a given question, the process of generating debugging exercises works as follows. First, we collect all previous student submissions marked as correct. We clean them up a bit to remove comments, student identifiers, and any bad language. (We don't find much.) Next, we run each one through our mutation engine, producing multiple new programs, most of which should be incorrect. We then validate that each exercise is in fact incorrect by using the Questioner autograder. Assuming it is, we save the incorrect mutated code as one debugging exercise for that problem.

To complete a debugging exercise a student not only needs to identify and fix the bug, but they must also do so without modifying more of the solution than was modified during the mutation process. This prevents students from simply rewriting the code from scratch using a different approach. It also ensures that students obtain one of the other important benefits of this exercise—the ability to work with and make small changes to code written by others. We normally require that students complete multiple debugging exercises for each question, which results in them being exposed to other ways of solving the problem by debugging code written by others.