Research Statement

Adam Blank

My research aims to use technology to improve student understanding and education quality. Over the years, the sizes of introductory courses have been increasing, which has in turn made it more difficult to teach. Courses effected by this are forced to make various concessions, such as:

- moving to large, impersonal lectures
- sacrificing learning outcomes to lower load on graders
- asking students to wait at office hours
- returning assignments slowly, and
- reducing quality of feedback

There are many partial solutions to these problems. In certain disciplines, student work can be specified formally (e.g. programming, algebra), which allows courses to write programs to auto-grade it. Intelligent tutoring systems give personalized feedback for sufficiently simple subject matter. Essays can be graded by classifiers driven by machine learning algorithms, if courses are willing to sacrifice giving comments to students. Peer grading can be used if students are sufficiently advanced or if courses can tolerate a lower quality of feedback and accuracy in grading.

My current work focuses on solving these problems in domains with student work that is written in a *fuzzy* medium (e.g. a short essay or mathematical proof) that contain very little *subjective* content. This class of student work includes programming style, free-form mathematical proofs, and many short answer science questions. As a result, it is particularly relevant to computer science education, and I have primarily investigated applying it to mathematical proofs written by introductory students. My work approaches these problems on two fronts: (1) by augmenting the traditional cycle in which assessments are released, completed, graded, and returned, and (2) by using particular technologies to solve approachable problems. This work uses various techniques spanning human computation, machine learning, AI, programming languages, and learning science.

Augmenting the Assessment Cycle

In most courses, assessments follow a cycle from distribution to the students getting comments on their work. I have augmented this cycle in several ways to combat the low quality and slow rate of feedback.

Uncoupling Comments and Scores. Traditionally, courses grade homeworks by printing them out, or accepting handwritten submissions, and writing down the places where students make mistakes on the paper. Paper grading has several major issues: lack of consistency in scores, redundancy in comments, and bias toward certain students. To deal with some of these problems, many courses use strict rubrics which associate particular mistakes with adjustments to scores. Unfortunately, rubrics introduce their own problems. To write a reasonable rubric, someone needs to predict a full class of mistakes (or create a less specific rubric), and the only way to change the score adjustment associated with a mistake is to re-grade all the homeworks. Additionally, graders must still re-write explanations of the student mistakes on every paper.

My work introduces a different strategy called *judgement-based grading*. Using my software, graders create a set of *judgements*, which function similarly to items in a rubric, as they come up in grading. Throughout this process, the grader (1) does not assign any point values, and (2) may re-use the comments they have given by clicking a button rather than re-writing them. Additionally, graders may

select a location to attach to the judgement. Once all the judgements have been assigned, graders may assign point values to individual items with a view that shows how the distribution of scores is affected.

A Different Kind of Peer Review. Since judgement-based grading allows graders to apply a rubric without choosing point values, students can be asked to do peer reviews using my system without worrying about how the final score should be calculated. Additionally, these peer reviews are targeted toward training students to get better at checking their own work and understanding the space of mistakes they might have made. My work explores various questions about these peer reviews from to how helpful they are to students to how accurate the results are.

Using Technology in the Classroom

Technology can add value to the classroom both as a student-tool for increasing understanding and as a way for instructors to handle their courses.

Teaching Quantifiers and Sets using a Compiler. Introductory discrete mathematics courses are almost always responsible for introducing students to the symbols, phrases, and definitions that are used in proofs. For example, classical implication is very counter-intuitive, because students often initially believe that it is related to *causality*, but it isn't. Learning the "language of mathematics" is just as difficult as learning a new programming language or a foreign language, and, because of this, it's very important that these courses help students in whatever way possible. My work introduces a toy programming language, {Set$_1$y}, which gives students a computational environment for exploring the mathematical language. Instead of guessing if a statement is true, they can check their understanding by instantiating it in various ways using {Set$_1$y}.

Teaching Algorithm Analysis with Abstract Interpretation. Algorithm analysis is a critically important topic to introductory computer science. Because deciding the $\mathcal{O}(-)$ complexity of an arbitrary program is undecidable, in general, most courses resort to teaching these ideas with cartesian graphs and runtime comparisons. My work uses *abstract interpretation* to decide run-time bounds (which are sometimes loose) on an incomplete, but large, set of programs. In an, appropriately scaffolded assignment, students were asked to write pieces of this algorithm to improve their understanding. When they were done, their program was able to show merge sort is $\mathcal{O}(n \log n)$ and nested for loops with bounds of n were $\mathcal{O}(n^2)$.

Future Work

Studying Compiler Error Messages for New Programmers. Many compilers output cryptic, verbose, or domain-knowledge specific error messages. These can be overwhelming, unreadable, or useless to new programmers. I intend to do user studies and attempt to reliably understand which error messages are useful and not useful to introductory programmers.

Implications of Autograding Student Code. As we autograde more and more student code, I have anecdotally observed students becoming less competent debuggers–they rely on the autograder's output to be able to review their code. I intend to explore this change in debugging strategy to link it to overuse of autograding, large class-size, or some other recent change in computer science education.

Using Abstract Interpretation for More Teaching Applications. There are other applications of abstract interpretation to programming topics that could be used to help students understand the topics better. In particular, exposing a pointer analysis of a large set of programs to students might be very helpful in teaching pointers. Similarly, an analysis of overlapping subproblems could prove very useful to students first learning dynamic programming.