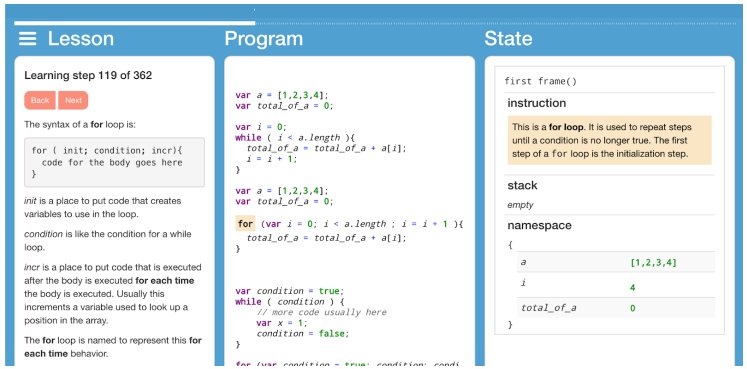
Greg Nelson

Major Research Accomplishments:

- *PLTutor teaches code reading skills better than a 10 week CS1 course, in about 4 hours*
- *Award-winning paper on use of scientific theory in computing education research*
- *Award-winning collaboration improving quantitative evaluation in HCI*
- *Award-winning collaboration in information visualization recommender systems*



Research Statement

I envision a world where everyone can quickly learn formal systems for thinking about and solving problems. For example, programming languages are a relatively new notation to describe processes in the world, and have been used in all the sciences to make huge advances for 70 years. Yet today less than 1% of the billions of people on our planet know even the basic elements of these notations. These powerful notations could show the underlying unity of concepts and ideas across domains, making them easier to learn. They could enable better ways to express, debate, and collaboratively improve social processes and institutions - advancing the very fabric of democratic society. To do this, we need equitable ways to teach everyone their fundamentals and rigorously assess effectiveness.

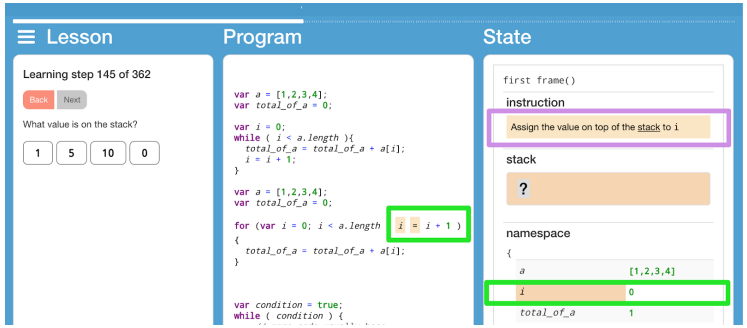
To enact this vision, I invent learning technologies for formal systems, with an emphasis on rigor in design and evaluation. In my dissertation, I've just begun to pursue this vision by looking closely at programming language (PL) semantics. In particular, I've investigated three things: how to teach them, how to assess that knowledge, and how to productively apply empirical social science theory to design learning technology and assessments.

PLTutor: Teaching PL Semantics in half a day

When we try to teach programming, students struggle; many don't even learn how to read code, let alone write it. Of every hundred CS1 students, 32 dropout or fail [1]. Even when students finish CS1, more than 60% can't predict the execution of simple programs [2]. Why don't the best methods from the past 50 years of teaching and research work well, such as showing a program and talking through it, showing input and output, and stepping through the program with visualizations?

My research indicates the problem is granularity: we don't show programs executing in enough detail and we don't show and assess a complete set of examples. Novices have to learn how each part of code causes state changes. Without showing small detailed steps, novices have to infer them; for example, even showing a program execute at the line level, novices have to infer the steps inside the line, such as a *for* loop's three parts and order of evaluation. Without showing a complete set of examples, novices have to infer how constructs combine with each other, such as sequential and nested statements. Without assessing a complete set of examples, novice misunderstandings persist and accumulate, then novices write buggy code and have to debug their code and language understanding at the same time.

To leverage this insight, I designed PLTutor, an interactive textbook that uses a human-written curriculum to show and assess a complete set of example programs [3]. PLTutor acts like an expert showing you example programs in a debugger, stepping and pausing to explain the



programming language, then testing you by asking, "What happens when this code executes?" For each example program, a human expert writes *learning steps*, interleaving conceptual explanation, program execution visualization, and assessment. Learning steps are a new abstraction that decouples teaching a program from the program's execution.

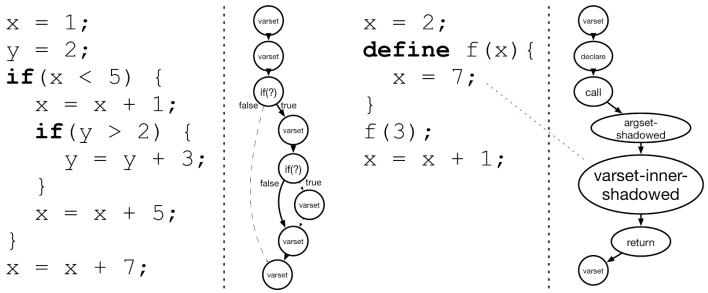
To build PLTutor, I had to redesign the PL language stack. Where normal runtimes are made to be fast, storing the least detail possible, I made a new abstraction for storing execution steps that connect machine state changes to the individual code tokens that cause them. I also made the granularity of the steps smaller (for example, a variable declaration takes three steps instead of one). Learners step through these simpler steps, seeing highlighted code tokens & state changes (see green boxes at left) and an automated natural language description of the step's execution rule (for example, "Assign the value on the top of the stack to *i*", see purple box at left). To enable assessments, the runtime wraps all program values so a learning step can specify what value to assess, and hide it until then.

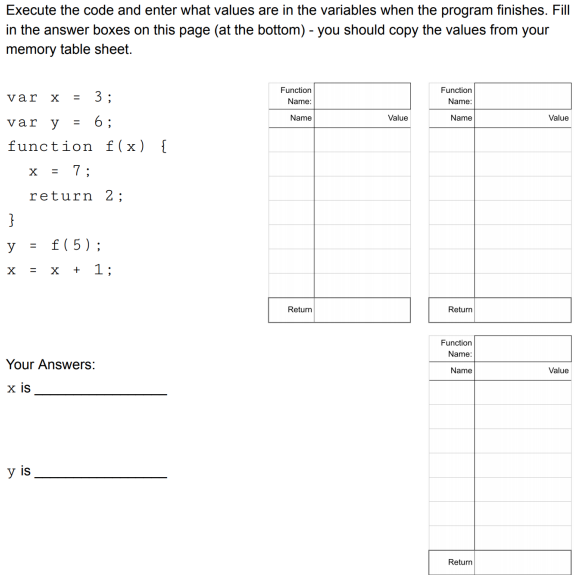
To evaluate PLTutor, I gave students a pre-test, then PLTutor or a writing tutorial, then a post-test; PLTutor taught reading skills better than a 10 week CS1 course, in about 4 hours — an order of magnitude faster — while also improving equity [3]. To evaluate PLTutor, students in the first week of a CS1 course took a pre-test, used PLTutor for ~4 hours, then a post-test; scores increased by 1.8 standard deviations ($p<.004$), ending above finished CS1 students. I gave other students a writing tutorial (Codecademy). More than 10% in the writing group failed their CS1 midterm — none for PLTutor. Beyond the classroom, in ongoing work I'm investigating PLTutor's impact on diverse learners trying to learn online, and finding the most impactful ways to design learning technology to help them.

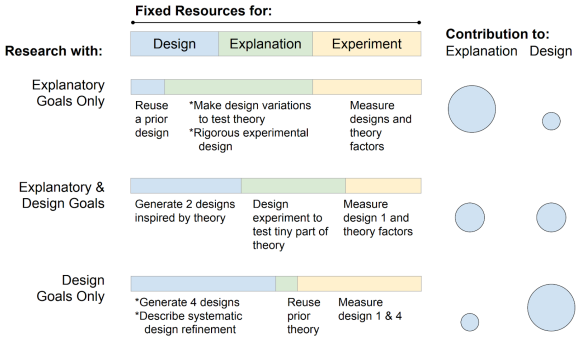
PLTutor provokes similar questions about granularity for other formal systems. I'm excited about finding the best granularity for showing and teaching formal systems from theory of computer science, databases, AI and ML, and within programming languages - for example, analysis of algorithms, relational algebra, deep learning, and type systems.

Diagnostic Assessment for PL Semantics

Biased, non-specific assessments of programming language knowledge create flaws in teaching and hiring. Automated and human teachers use large, non-specific code execution questions, with programs of 7-20 lines that require 20-30 units of knowledge; a student that knows 95% of them will get a zero, feel they've made no progress, and the instructor can't specify what they don't know [4]. Millions of employers use resumes to filter who to interview; if we had unbiased knowledge assessments instead, that would be more efficient and equitable.







I developed an advanced diagnostic test of programming language knowledge; it systematically assesses program tracing skills (mentally executing a program) [5]. The key idea is a model of the learner's program execution knowledge that extends control flow graphs to include combinations of semantics and scoping. To cover each part of the model, I designed a question template that reduces guesses and mistakes, then made my test by generating questions systematically that cover the learner model. Because each question is so specific, we can triangulate test performance to reliably infer what a learner knows, then give specific feedback.

Building upon modern best practices in psychometrics and educational assessment, I constructed an argument for the assessment's validity and rigorously evaluated the claims of the argument. I drew on the latest framework from educational assessment for validity and introduced it to the computing community [5]. I gave my test to 31 learners, randomly taught a specific part of knowledge they didn't know, then gave a post test. The test helped learning; the post-test showed they learned the part we taught 77% of the time, and the learner's self-reported learning matched. The test was accurate, with average guess and mistake rates less than 5% and 8% per question. Learners' self-confidence dropped slightly after the test but came back stronger after the guided feedback.

I went beyond using best practices for rigor, to improving them; I made a new experimental method for estimating test accuracy that reduces the needed sample size of learners taking the test. I reduced the sample size from polynomial in the model parameters, to linear; the key idea is ways to observe hidden variables like guesses and mistakes, instead of fitting them implicitly from test answers alone. For my test's evaluation, my sample size was ~100 times smaller.

Advancing Discourse on Scientific Theory and Design for Learning

To build a scientific discipline for teaching formal systems, we need to combine the best of scientific theory and innovative design. If we just have innovative design, some designs will work, but without theory for understanding why, improving them will be hard because we'll lack guidance for exploring the combinatorial space of design choices. In the worst case, designs will have flawed evaluations with low quality, non-standard evaluation measures we can't compare across systems. If we just have scientific theory, we will try to test and understand simpler, more predictable problems, at the expense of making novel designs that lack theory saying they should work. In the worst case, we may use theory to reject empirical evaluations because theory says the design should not work.

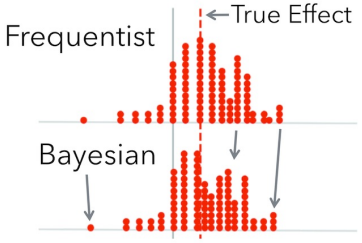
I defined these trade-offs and new ideas for balancing them in an award-winning paper at the top computing education research conference [6]. I explained the fundamental trade-off between theory and design - trying to rigorously understand why your design works is different

from broad design exploration into areas we don't yet understand. I argued we need more computing specific theories of learning and high-quality assessments to evaluate and compare learning technologies. I proposed auditing peer review to track bad logics used, where and how often, and how to give feedback to reviewers. I argued a principled minimum standard to share novel design work.

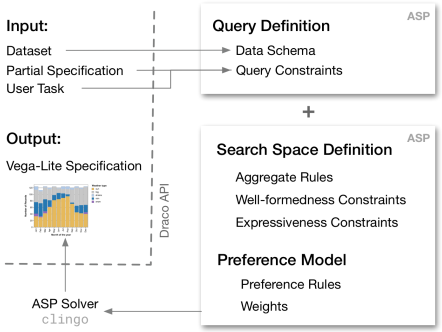
My argument has had lasting and widespread impact. The ICER program chairs invited my paper to be the subject of a one hour panel, the only time this has happened in ICER's 14 year history. This panel led to a follow-up panel at the biggest practitioner conference in computing education. In the years after, every best paper at ICER built upon the topics I raised. The conference changed its reviewing guidelines and criteria. A working group started to come up with better ways to do peer review.

Advancing HCI through Collaborations

While my core interest is on literacy for formal systems, I also greatly enjoy collaborating on big ideas in adjacent areas. Here are three examples.



For example, one opportunity I saw was to improve quantitative evaluations in human-computer interaction research. I saw a major problem: poor estimates of effectiveness from small evaluation studies and lack of meta-analyses, which aggregate multiple studies to make better estimates; without good estimates we can think tools are great when they aren't and ignore problems we think are solved. I saw the opportunity, invited Matt Kay to meet, convinced him, and we co-developed the key ideas: applying a Bayesian statistical method that improves estimates by using the evaluation data from prior studies, while also aggregating effects for prior tools like a meta-analysis. Our work won an honorable mention best paper at the top HCI conference [7]; later work improving statistical practice continues to build on it.



Another opportunity I saw was in data visualization; I wrote a paper draft and posed a new research question that reinvigorated the work. Dominik Moritz had the start of a great idea but was not going to pursue publishing it: using a constraint language to encode knowledge about good visualization designs gleamed from empirical studies, then recommend good designs. We co-developed the first prototype, then I refined the framing and wrote a full paper draft that convinced Dominik it was publishable; he said "I can see how this could become a full paper and would be happy to support you in making it happen." Then I posed a new question: how to learn weights for the constraints from experimental data. Dominik got excited, and Chenglong joined the work to build a new machine learning system to solve the problem. The finished paper won the best paper award at the top visualization conference [8], and opened a new sub-area of work within visualization recommender systems.

As a third example, I saw a long-standing scoping flaw in designing advanced topic assessments in many domains, including computing education. I saw the problem: learners can have weaknesses with an advanced topic or its prerequisites, but most advanced topic questions are designed to only diagnose the advanced topics. I saw the opportunity: inventing low cost, high benefit ways to add small parts to existing advanced questions for computing, so learners show work in a more structured way. These *differentiated assessments* could improve instructors' feedback to disadvantaged students with weaker prior knowledge, improving equity while also helping everyone. To do this work, I led a proposal for a conference working group; it convinced seven people from four countries to join. I co-led this group during the pandemic. Our peer-reviewed report on design principles for differentiated assessments was accepted [9], and we're planning empirical evaluation studies. I've also talked to educational assessment experts, who say the area of differentiated assessments appears novel for that field; we're exploring another paper in their venues.

Future Work

I want everyone to learn formal systems to think about and solve problems, and I have many ideas for future work. I want to automatically generate a curriculum and teaching system for any programming language, based on its interpreter. I want to figure out how to assess and teach programming environments that use program synthesis to help write and debug code. I want to make rigorous assessments and teaching for learning machine learning. I want to build tools that analyze machine-made and human-made computing curricula, then merge them to get the best of both, building on my expertise in knowledge modeling in my assessment work. My scientific theory work argued how computing education needs standardized measures to compare tools; HCI has the same problem, and I want to fix it and improve the rigor of the whole field.

I'm excited to work on these, but also many other big ideas that require collaboration across CS and beyond.

References

1. C Watson, F WB Li. Failure rates in introductory programming revisited *ACM Conference on Innovation & Technology in Computer Science Education (ITiCSE 2014)*
2. Lister et al. A multi-national study of reading and tracing skills in novice programmers *ACM SIGCSE Bulletin* 36 *(SIGCSE 2004)*
3. G L Nelson, B Xie, A J Ko. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1 *ACM Conference on International Computing Education Research (ICER 2017)*

4. A Luxton-Reilly, A Petersen. The Compound Nature of Novice Programming Assessments. *ACM Proceedings of the Nineteenth Australasian Computing Education Conference 2017*
5. G L Nelson, A Hu, B Xie, A J Ko. Towards validity for a formative assessment for language-specific program tracing skills. *ACM 19th Koli Calling International Conference on Computing Education Research (Koli 2019)*
6. 🏆 G L Nelson, A J Ko. On Use of Theory in Computing Education Research. *ACM Conference on International Computing Education Research (ICER 2018)* *Best paper award*
7. 🏆 M Kay, G L Nelson, E B Hekler. Researcher-Centered Design of Statistics: Why Bayesian Statistics Better Fit the Culture and Incentives of HCI. *ACM CHI Conference on Human Factors in Computing Systems (CHI 2016)* *Honorable mention best paper award (top 5%)*
8. 🏆 D Moritz, C Wang, G L Nelson, H Lin, A Smith, B Howe, J Heer. Formalizing visualization design knowledge as constraints: Actionable and extensible models in Draco. *IEEE Transactions on Visualization and Computer Graphics (InfoVis 2018)* *Best paper award*
9. G L Nelson, F Strömbäck, A Korhonen, M Begum, B Blamey, K H Jin, V Lonati, B MacKellar, M Monga. Designing Assessments for Advanced Courses that also Diagnose Prerequisite Skill Issues: A Design Investigation for Research and Practice. *ACM Proceedings of the 25th Conference on Information Technology in Computer Science Education 2020 (ITiCSE 2020), to appear in*