I see programming language techniques as general tools for optimization and correctness that can help programmers analyze, verify and improve programs in any domain. My work draws from the wide spectrum of PL techniques, from superoptimization and satisfiability modulo theory to proof assistants, and applies it in new ways and to new problems.

Below I describe two of my projects. *Cassius* applies automated theorem proving techniques to verify that web page layouts are accessible to users with visual and sensorimotor disabilities. *Herbie* applies compiler techniques to translate scientific formulas into accurate floating-point computations. I then describe plans to apply programming languages techniques to web applications, visual design, and scientific experiments.

Cassius: Verifying Web Page Layouts

Web page layouts are programs: even "static" pages respond to inputs like browser size, operating system, and device type. Layouts can have bugs, such as overlapping text, inaccessible buttons, and misplaced elements, especially for users with non-standard preferences like larger font sizes, leaving important interfaces inaccessible to users with low vision or sensorimotor disabilities (such as the bug in Figure 1, on `healthcare.gov`). Such bugs result not just in lost revenue and frustrated users, but also in legal liability, since web page accessibility in the US is governed by the Americans with Disabilities Act.

Today, developers catch layout bugs with ad-hoc testing at multiple sizes or on multiple devices. It doesn't work: it cannot cover the many possible sizes and devices, so layout bugs persist, especially for users with non-standard preferences. Viewing web pages as programs shows how language semantics, program verification, and theorem proving can help developers find layout bugs, fix them, and verify that no further problems exist. Unlike classic applications of PL techniques, the goal is not functional correctness but *mobile-friendliness*, *usability*, and *accessibility*.

To apply PL techniques to web pages, I have developed a semantics for web page layout, a verification algorithm over this semantics, and a proof assistant for using this verification algorithm to build proofs of web page accessibility.

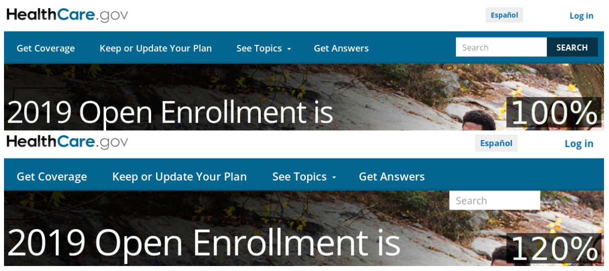

Figure 1: `healthcare.gov`, at 100% and 120% font size; at the larger font size, which might be used by a low-vision user, the dark blue search button (top right) nearly disappears from the page.

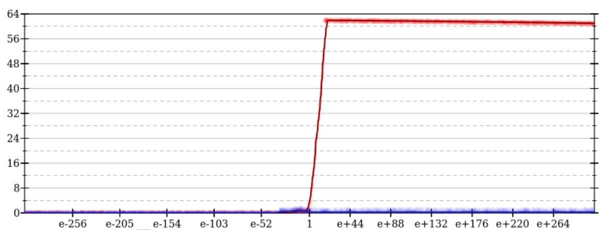

Figure 2: The error, in bits, of $\sqrt{x+1} - \sqrt{x}$ (red) and $1/(\sqrt{x+1}+\sqrt{x})$ (blue). The formulas are mathematically equivalent, but the latter is significantly more accurate for $x > 1$.

Semantics Formalizing web page layout is challenging due to its complicated specification. For example, if text is wider than its container, the text alignment command is ignored and the text is is left aligned—unless the text is in a right-to-left language. Prior to my work, the only descriptions of the layout process were the English-language standards text (ambiguous and incomplete) and browser implementations (million-line decade-old C++ code). I formalized a substantial fragment of the CSS styling language [7], including complex features like line height, margin collapsing, and floating layout [5]. To date, this remains the only formalization of a substantial fragment of web page layout.

Verification Verifying web page accessibility means proving that a web page is accessible across all browser sizes, operating systems, and devices. Proving program properties is a common programming languages task; however, applying the same techniques to visual layout is novel and under-explored. My semantics of web page layout made this ambitious idea possible. I developed a new logic to describe visual properties like overlap, alignment, and size. Using this logic, I formalized 14 accessibility properties, drawn from sources like Apple and Google accessible design guidelines, and from Department of Justice recommendations.

I then developed an algorithm, VizAssert, to automatically verify visual properties expressed in the logic [5]. Given a web page and a visual property to verify, the verifier uses the semantics to define the set of all possible renderings of the page, and then searches that set for renderings that fail to satisfy the property. To make this search efficient, I designed the logic to compile to assertions in the theory of quantifier-free linear real arithmetic, for which automated decision procedures already exist. This compilation required novel data structures for efficiently reasoning about line height, margin collapsing, floating layout, and other aspects of web page layout. VizAssert has verified the 14 assertions I formalized on 60 real-world web pages.

Modularity My verifier operated on whole web pages, but web developers generally work on web pages piece by piece, with components like headers, footers, and menus reused on many web pages. Fitting this workflow requires verifying web pages piece by piece. But since web page layout is highly context-dependent, this requires writing down specifications both for each piece of a web page and for the context that piece can be used in. To organize these tasks, I adapted the idea of a proof assistant. In a proof assistant, developers decompose software into components, write specifications, and prove them for each component, and integrate these components to prove properties of the software as a whole, with each step automatically checked for correctness. Web pages, too, can be decomposed into components, and have specifications for each component. So, I wrote a proof assistant to allow web developers to do both those tasks, and which then automatically check each component against its specification and that the component specifications imply whole-page properties. The key insight in this approach is that each component of a web page is checked to satisfy its specification in all web pages the component could be used in, guaranteeing that a component can be freely reused between multiple pages. As an additional benefit, each component can be verified independently and in parallel, allowing web page verification to scale to large pages.

My work on web page layout has opened a new research area at the intersection of PL and HCI. By combining insights from both fields, I have developed new tools for ensuring web page accessibility; these tools are now the basis of an NSF "Formal Methods in the Field" award.

Herbie: Improving Accuracy for Numerical Programs

Scientists write programs that analyze data, run simulations, and compute the consequences of physical laws. For these programs, a key correctness requirement is *accuracy*. Yet these programs usually approximate the arithmetic of real numbers with floating-point values. Because many real numbers are represented by the same floating-point value, floating-point approximations can cause large errors when values are close together. Thus, identical real expressions, such as $\sqrt{x+1} - \sqrt{x}$ and $1/(\sqrt{x+1} + \sqrt{x})$, can have divergent error behavior in floating point (as in Figure 2). Ensuring that floating-point computations are accurate requires expertise most working scientists lack.

I developed Herbie, a tool that automatically translates scientific formulas to accurate floating-point formulas [6]. Before Herbie, the best tools measured accuracy improvement in percentages; Herbie frequently improves accuracy by orders of magnitude. This success is possible due to Herbie's reinvention of classic compiler techniques (like equivalence graphs and backwards-chaining rewrites) in the context of mathematical expressions, optimizing for accuracy instead of performance.

Impact Herbie won a distinguished paper award at PLDI, a top PL venue. However, I was not content with purely academic impact. I have maintained and improved Herbie, taking it from a research artifact to a usable and intuitive tool. Herbie is used by hundreds of scientists and engineers, including at NASA and Sandia National Laboratories. Herbie is even recommended in a recent numerical methods textbook [3]. I have also mentored multiple undergraduates on projects extending Herbie, and have collaborated with colleagues on binary debugging tools [8] and 3D-printing environments [4] that require numerical accuracy. Herbie is part of the UW grant award in the DARPA BRASS program, adapting computations to run on different floating-point hardware.

Because of my difficulties finding a representative benchmark suite to evaluate Herbie on, I started the FPBench project to collect benchmarks from the floating-point research community [2]. FPBench has been used by colleagues at MPI-SWS, Rutgers, University of Utah, and University de Perpignan. I also developed FPCore, a common format for floating-point accuracy tools. FPCore is used to compare and combine multiple research tools, and I have collaborated with other research groups to carry out these comparisons [1]. FPBench is part of the ADA grant award for the DARPA/SRC JUMP program, providing a common suite of benchmarks to evaluate new hardware floating-point formats.

Research Vision

I have applied PL techniques to ensuring web page accessibility and to ensuring numerical accuracy, but I intend to apply these techniques to many more areas. In the past, I found new fields by reading widely and talking to researchers in other fields, and I am sure this will yield many more promising problems in the future. For now, I am excited about the possibility of applying PL techniques to verify full web applications, visual designs, and experiment designs. In the long term, I want to make PL techniques general-purpose tools for understanding, verifying, and improving programs from any domain.

My work on web page layout has made it possible to reason about how a static web page will look, but it has not considered dynamically-generated content. Meeting this challenge would make it possible to verify full web applications, from database to UI. Verifying the interaction of multiple programs is a frontier in verification, and my work on modular verification of web page layouts is a powerful starting point. My work allows separating the page into static and dynamic components, and using different techniques to reason about the two, such as layout verification for the static components and a program logic for the backend software. Verifying full web applications would push the boundaries of verification and also extend PL techniques to an important and growing class of applications.

My work on web page layout inspired me to look at visual design as an application area for PL. The wireframes and mock-ups produced by designers are logical specifications for whatever web page, poster, or printed page they are designing. Wireframes and mock-ups describe geometric relationships between objects: aligned, one below another, or equally tall. These relationships could be inferred from a wireframe; then an implementation of the mock-up can be verified against these relationships. Treating designs as specifications makes it possible to test designs against higher-level correctness requirements like readability or information density, and could formalize style transfer (where one application is automatically restyled to look like another), making it easier to use and more predictable. Today, visual design is a notoriously informal field; approaching it as specification-writing could allow new formal methods to be applied.

Looking outside computer science, I see experiment design as an application domain for PL techniques. An experiment may involve multiple independent measurements of some result that are statistically combined, or multiple staged experiments to allow early exits and to minimize costs. Designing an experiment involves choosing between different combinations of control and experimental groups, early exit options, and size of test populations. Viewing the experiment design as a program allows bringing PL techniques to bear to optimize experiment designs for cost or accuracy. Furthermore, a structured language for expressing experiment designs could also allow integrating software experiments (such as A/B tests) into larger systems more transparently.

My research uses the core algorithms and theories behind compilers, type checkers, and automated theorem provers as general purpose tools for understanding, verification, and improving programs like web page layouts and mathematical formulas. I believe these algorithms and theories have wide applicability. I see a lifetime of opportunities adapting PL techniques to other fields.

References

[1] Heiko Becker, Pavel Panchekha, Eva Darulova, and Zachary Tatlock. Combining tools for optimization and analysis of floating-point computations. FM '18, pages 355–363, 2018.

[2] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. Toward a standard benchmark format and suite for floating-point analysis. NSV '17, pages 63–77, 2017.

[3] Ronald T. Kneusel. *Numbers and Computers*. Springer, 2nd edition, 2017.

[4] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. Functional programming for compiling and decompiling computer-aided design. *Proc. ACM Program. Lang.*, 2(ICFP):99:1–99:31, July 2018.

[5] Pavel Panchekha, Adam T. Geller, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. Verifying that web pages have accessible layout. PLDI 2018, pages 1–14, 2018.

[6] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. PLDI '15, pages 1–11, 2015.

[7] Pavel Panchekha and Emina Torlak. Automated reasoning for web page layout. OOPSLA 2016, pages 181–194, 2016.

[8] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. Finding root causes of floating point error. PLDI 2018, pages 256–269, 2018.