Research Statment

Alex Sanchez-Stern

Research in the computer science literature has produced many techniques for producing high-quality, high-assurance software. Unfortunately, many of these techniques are difficult to use, and require special training and intense manual effort to produce high-quality results. **My research goal is to use programming languages techniques to bring expert knowledge to non-experts**. In doing so, I've worked on tools to allow more programmers to write bug-free numerical code [14, 19](PLDI 2015, PLDI 2018), secure web-applications, and high-assurance verified software [17, 18], (CPP 2020, MAPL 2020).

1 Synthesizing and Debugging Numerical Software

Floating point math is a notoriously leaky abstraction over the real numbers, making writing reliable numerical code extremely difficult [8, 9, 20]. When the abstraction fails, rounding errors have caused the retraction of scientific articles [1, 3, 2], legal regulations in finance [6], and distorted stock market indices [13, 16]. Developers working with floating point often try to fix rounding error through ad-hoc methods, such as randomly perturbing the code until it works on test inputs [20, 9], or increasing the bit-width of their floating-point values. Unfortunately, these approaches can lead to further errors and significant performance degradation, and are completely ineffective for many problems. When development of numerical software has access to significant resources, developers can use formal numerical analysis to produce accurate programs, but these techniques require significant expertise, and the process is still slow and complicated.

My colleagues and I at the University of Washington first addressed this challenge with Herbie [14]. **Herbie allows users to write down the real-number computation that they want to compute, and automatically transforms it into floating-point code which can compute it quickly and accurately**. This allows users to program in abstractions they understand, and brings techniques of numerical analysis, once restricted to a few experts around the world, into the compilers of everyday programmers. We published Herbie at PLDI 2015, where it received the Distinguished Paper Award, and had the distinction of being the first talk of the conference, before tracks were split. Herbie is available as an official Racket package, a user-friendly web interface, and a plugin to the Glasgow Haskell Compiler. It has had patches accepted to widely used math libraries like MathJS, and has been used by researchers at a variety of institutions, including NASA.

However, in order to use Herbie, a user must know which segments of their code are non-trivial numerically, and worthy of analysis. This is not always an easy task, especially since much of the large numerical software in use is legacy software, written decades ago and maintained carefully. Extracting numerical code from such software for repair is extremely difficult, especially since the operations which interact to cause error can be distributed throughout the program source code, and can execute at very different points in the programs execution.

To bring the power of Herbie to such large software, the Herbie team and I, as well as some of my new colleagues at UCSD, developed a binary analysis tool called Herbgrind [19]. Built using the Valgrind framework, **Herbgrind instruments binary programs to track the sources of floating-point error**, and gather a variety of information about the error into an error report which can allow the user to easily use Herbie to improve the numerical code. Herbgrind uses a novel variety of anti-unification to synthesize program fragments representing the dynamic flow of values, allowing operations to be analyzed across data structures and control flow. Herbgrind was published at PLDI 2018.

With these two tools, Herbie and Herbgrind, my colleagues and I worked to bring the domain expertise of numerical analysis to more programmers. While nothing can replace the consultation of a genuine expert, these tools have been empirically shown to be able to bring large amounts of the benefits of a numerical expert, in settings where obtaining the services of one of the few numerical analysis experts is infeasible.

2 Languages for Database Security

The correctness of many web applications depends on enforcing rules about what information a user can access and provide during the course of the application. From password-leaking vulnerabilities, to SQL injection, to inferring the private personal information of another user, the consequences of improperly gated data access can be catastrophic. Unfortunately, the logic that enforces data access protection is often scattered throughout an application and duplicated, making it brittle to changing application requirements and features.

Promising work has been done developing policy languages, an approach where the developer writes down data access permissions in a centralized place, and it is enforced at every access point. This prevents bugs due to permission mismatches throughout the code. But it does not address the possibility that the policies written down are themselves buggy, nor does it address the needs of web applications where data schemas and policies are constantly changing.

Past policy languages have specified the policy of data at any given point of the development process, but not how they change as databases are migrated and policies are updated. These changes are important, as migration can leak secret data to the public, and introduce buggy policies.

To address this gap, John Renner, Fraser Brown, Deian Stefan, Sorin Lerner, and I developed Caravan, a new kind of policy language. **Caravan allows the user to not just specify data layout and policies, but also data- and policy-migrations which transform the data and policies between versions**. Additionally, our policy language is constructed around an SMT semantics which allows it to quickly check properties of large policies, like “is any data that was secret before the migration publicly accessible after?”. Caravan is under submission to PLDI 2021.

Best security practices and in-depth security audits with experts have been, in the past, able to bring many of the same benefits to secure web-application development. However, these techniques require both expert knowledge and high amounts of review effort to produce reliable results, and are thus mainly available to larger development environments. With Caravan, our team brings the benefits of such techniques to more programmers and development efforts with less intensive effort.

3 Studying and Automating Proof Engineering

One of the most promising approaches to producing high-reliability software is *foundational verification*. In this approach, programmers write programs and verify properties of them in specialized tooling known as an “interactive theorem prover”, such as Coq [7] or Isabelle/HOL [15]. This approach has been used to verify a variety of software, including compilers [11], operating systems [10], database systems [12], file systems [5], distributed systems [21], and cryptographic primitives [4].

Unfortunately, the reliability of software produced in this manner comes at a high cost: expert effort. Even for those trained in the use of interactive theorem provers, proofs are extremely time-intensive to write. CompCert [11] took 6 person-years and 100,000 lines of Coq to write and verify, and seL4 [10], which is a verified version of a 10,000 line operating system, took 22 person-years to verify. And for those without a graduate degree in programming languages, these tools are generally inaccessible due to the grasp of concepts required.

So the question becomes, “how do we bring the benefits of foundational verification to a wider audience” ? To begin to address that question, Talia Ringer and I developed REPLICA, published at CPP 2020. **With REPLICA we conducted the first user-study of proof engineers, in order to understand what kinds of changes they make to proofs during development, and where their time is spent**. Using the data from this study, we could understand the needs of proof engineers, in order to make the field more widely accessible to programmers through tooling.

From the data gathered through REPLICA, my team at UCSD and I moved on to tackling one of the most pressing problems in the accessibility of proof engineering: the amount of effort required to write and maintain large numbers of proofs, many of which are re-expressing simple facts about custom data structures. **With Proverbot9001, published at MAPL2020, my team built a system to tackle this problem; a tool which uses machine learning to attempt to build proofs of arbitrary logical statements**. Unlike other work which takes a low-level approach to building the syntactic constructs of the proofs, Proverbot9001 makes use of the tactic systems present in modern proof assistants, where small self-contained search procedures are invoked in sequence to construct a final proof term. Proverbot9001 learns from existing proofs to determine likely-useful tactics, and uses this knowledge to guide a search tree which constructs the final proof. With this approach, Proverbot9001 is able to re-discover more than a quarter of proofs in the CompCert verified C compiler, drastically reducing the amount of human effort required to maintain such a projects.

Unlike in my other works, where domain expertise was encoded entirely by the tool authors, in Proverbot9001 we combined our own knowledge of proof engineering expertise with the statistical correlations and preferences learned by the ML process. In both approaches though, we were able to bring domain expertise to more programmers through well-designed tooling.

4 Future Work

My work so far has only begun to scratch the surface of what's possible in bringing hard-won expert domain knowledge to more programmers. As the amount of software in our world grows exponentially, so too does the need for reliability in that software; however we are still broadly under-equipped to allow most developers to deliver the quality of software which our new world demands. With software permeating ever-more personal aspects of our lives, the safeguarding of our digital data from malicious actors relies on being able to trust the software which we run.

The goal of my future work is to use PL techniques to develop tooling which brings expert knowledge to non-experts. By bringing these expert abilities and techniques to developers who do not have that expertise, I hope to do what much of good programming languages research does: democratize software development.

Broadly, my approach to this breaks down into two parts: (1) **Powering up PL:** using insights from fields like machine-learning to bring more power to classic PL problems like proof search, and (2) **Codify expertise:** using PL techniques to codify domain expertise to bring the benefits of programming expertise to more users.

Powering up PL My thesis work has already begun to explore the former, using the power of neural networks to drastically reduce the amount of time needed to verify software. I began this work in early 2016; since then teams from Google, CMU, UMass Amherst, Princeton, and others have explored this direction with a variety of approaches; to date however, none have been able to outperform Proverbot9001 on significant proof developments like CompCert.

But even with improving proof-synthesis technology, there is still a large gap between the specialized proof assistants in which verified software is produced, and the tooling in which most software is developed. Attempts to bridge this gap, like the Dafny programming language, are promising but still outside of the reach of most developers. In my future work, I intend to explore the connection between the invariants and proof approaches required to synthesize a proof of program correctness, and the developer intuition for correctness, which can and often is written down in informal comments accompanying the code. By bringing together these two tasks, I hope to decrease the developer effort required to verify software while at the same time increasing the detail and reliability of code documentation.

Proof search is just one of many problems in programming languages that can be tackled with this approach. Program synthesis, error minimization, and test input generation can all be tackled with the kind of machine-learning guided search that makes Proverbot9001 so powerful.

Codify expertise Unfortunately, much of what is important in software development of different domains has yet to be codified in the tools we use. Even the best search can't prevent programmers from writing bugs when the constraints of the domain are not codified in the language or tooling. My work in numerical software showed me what happens when programmers are given leaky abstractions they don't understand. But modern code deals not just with floating-point, but also data-structures with real-world constraints, complex tensor computations, and subtle ownership properties.

As one example, many machine learning models are defined in languages like Python using entirely dynamically-typed data structures. This leads to a whole host of errors at runtime, and many are not addressed by existing typed languages. Tensor size mismatches, indexing errors, and gradient misuse are all issues that can crash your program or invalidate it's results, often only becoming apparent after hours or days of training. But each of these can be fixed by bringing the constraints on these datatypes into the language; tensors can be size-indexed with indexed types allowing safe tensor operations, indices can be similarly bounded to prevent indexing issues, and affine types can ensure proper management of gradients. In my future work, I intend to bring these domain constraints into the language, so that more non-experts can use these tools safely and effectively.

Collaboration I also plan to continue my work collaborating with my colleagues in other fields to bring programming language techniques into the problems faced by developers in a variety of domains. My experience with Herbie, Herbgrind, and Caravan has shown me the wealth of opportunity in collaborations across domains to produce high-impact tooling. While these collaborations will depend highly on the colleagues and expertise I have available, I'm particularly interested in continuing my work applying Programming Language techniques to security domains, and exploring the interaction between tooling and machine learning from new perspectives.

References

[1] M. Altman, J. Gill, and M. P. McDonald. *Numerical Issues in Statistical Computing for the Social Scientist.* Springer-Verlag, 2003.

[2] M. Altman and M. McDonald. The robustness of statistical abstractions. *Political Methodology*, 1999.

[3] M. Altman and M. P. McDonald. Replication with attention to numerical accuracy. *Political Analysis*, 11(3):302–307, 2003.

[4] A. W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, Apr. 2015.

[5] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 270–286, New York, NY, USA, 2017. ACM.

[6] European Commission. *The introduction of the euro and the rounding of currency amounts.* Euro papers. European Commission, Directorate General II Economic and Financial Affairs, 1998.

[7] J.-C. Filliâtre, H. Herbelin, B. Barras, B. Barras, S. Boutin, E. Giménez, S. Boutin, G. Huet, C. Muñoz, C. Cornes, C. Cornes, J. Courant, J. Courant, C. Murthy, C. Murthy, C. Parent, C. Parent, C. Paulin-mohring, C. Paulin-mohring, A. Saibi, A. Saibi, B. Werner, and B. Werner. The coq proof assistant - reference manual version 6.1. Technical report, 1997.

[8] W. Kahan. Miscalculating area and angles of a needle-like triangle. Technical report, University of California, Berkeley, Mar. 2000.

[9] W. Kahan and J. D. Darcy. How Java's floating-point hurts everyone everywhere. Technical report, University of California, Berkeley, June 1998.

[10] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[11] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[12] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 237–248, New York, NY, USA, 2010. ACM.

[13] B. D. McCullough and H. D. Vinod. The numerical reliability of econometric software. *Journal of Economic Literature*, 37(2):633–665, 1999.

[14] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15. ACM, 2015.

[15] L. C. Paulson. Natural deduction as higher-order resolution. *CoRR*, cs.LO/9301104, 1993.

[16] K. Quinn. Ever had problems rounding off figures? This stock exchange has. *The Wall Street Journal*, page 37, November 8, 1983.

[17] T. Ringer, A. Sanchez-Stern, D. Grossman, and S. Lerner. Replica: Repl instrumentation for coq analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 99–113, New York, NY, USA, 2020. Association for Computing Machinery.

[18] A. Sanchez-Stern, Y. Alhessi, L. Saul, and S. Lerner. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2020, page 1–10, New York, NY, USA, 2020. Association for Computing Machinery.

[19] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock. Finding root causes of floating point error. *SIGPLAN Not.*, 53(4):256–269, June 2018.

[20] N. Toronto and J. McCarthy. Practically accurate floating-point math. *Computing in Science Engineering*, 16(4):80–95, July 2014.

[21] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. ACM.