The pace at which code is produced has far outstripped efforts to ensure its correctness, analogous to how buildings are hastily erected before engineers can guarantee their stability. My research applies programming languages and formal methods to strengthen the correctness and reliability of software in emerging domains.

I focus on making *new* software trustworthy and scalable through the design of domain-specific languages (DSLs) that offer reusable, safe interfaces. These DSLs not only help developers clearly express intent (reducing bugs), but also allow *program synthesizers* to automatically generate correct implementations from specifications. To improve assurance for *existing* software, I use formal verification to prove the absence of critical classes of bugs. I have applied this approach across diverse domains, from end-user applications like data visualization to expert systems such as privacy-preserving software and browser layout engines.

My future directions build on this foundation and lend themselves well to undergraduate research, as it spans both theoretical work in language and algorithm design and tool implementation projects with real-world impact.

Past Research

Verification and Compilation of Zero-Knowledge Proof Software In my thesis, I apply my approach to the zero-knowledge proof (ZKP) domain. ZKPs let one prove a statement's truth without revealing its content, enabling applications from secure age verification to certifying unaltered media. Yet adoption remains limited: developers must write complex arithmetic circuits, where bugs can allow attackers to prove falsehoods and compromise systems silently.

My key insight is to raise the abstraction level of ZKP languages: provide (1) a high-level language for safer development and verification, and (2) a compiler that optimizes these programs into efficient circuits.

For the language, I co-designed Coda [4], which combines familiar syntax with domain-specific abstractions and an expressive refinement type system. Developers can specify precise logical properties and verify them using the Rocq proof assistant. We re-implemented 77 circuits, formally proved their correctness, and uncovered six zero-day vulnerabilities. This work appeared at 2024 IEEE Security & Privacy, and has been adopted by Veridise, a leading blockchain security startup company, to perform security audits for critical ZKP infrastructure projects [6].

For compilation, I built an optimizing compiler targeting ZKP backends [5]. Since computations admit many equivalent circuit representations with drastically different costs, developers previously relied on painstaking manual tuning. Our compiler replaces this with program synthesis, automatically discovering optimal implementations. It reduces proof generation time—a key ZKP metric—by up to 94% over existing tools. This work will appear at 2025 Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), a premier venue for programming languages research.

Synthesis of Data Visualization Programs Visualizations are essential for exploring and sharing data, yet creating effective ones is difficult for non-experts who must master complex libraries like R, Python, or D3. To democratize visualization, we introduced Calico [3], a system that synthesizes visualization programs from user intent. Calico provides a refinement type-based DSL: instead of manually wrangling data, users supply type annotations that constrain visual attributes or specify relationships between visual components in the desired graph. Our synthesis algorithm then uses lightweight bidirectional type checking to generate candidate visualizations. Calico solved 98% of benchmarks from online forums and proved easy to learn in a user study with participants lacking visualization expertise. This work appeared at 2024 Automated Software Engineering (ASE), a leading software engineering venue.

Synthesis of Browser Layout Engine Web browsers rely on layout engines to render pages from HTML and CSS, but building them is notoriously complex and error-prone, often leading to visual glitches that impact billions of users. Our work [1, 2] tackles this with program synthesis. We designed a DSL, based on attribute grammars, for specifying layout semantics *declaratively*. A synthesis algorithm then generates a schedule ensuring attributes are computed only when dependencies are available, eliminating entire classes of layout bugs. The system also supports incremental layout, recomputing only what changes after small user interactions.

From declarative CSS specifications, our tool synthesized a layout engine we integrated into an experimental Mozilla browser. The result was correct-by-construction and avoided many bugs found in existing engines. This work appeared at 2022 Architectural Support for Programming Languages and Operating Systems (ASPLOS) and 2023 Programming Language Design and Implementation (PLDI), top venues in systems and programming languages.

Ongoing and Future Work

More Expressive DSLs for ZKP Programming Our Coda language currently has limited support for unbounded loops and mutable storage. Recent backend advances, such as recursive proofs and lookup tables, now make these features feasible. Can we design an ergonomic ZKP language that incorporates such expressiveness without sacrificing performance? Can compilers leverage backend features to optimize these programs automatically?

Formal Verification of Zero-Knowledge Virtual Machines An emerging paradigm compiles high-level ZKP programs into assembly for a dedicated *ZK virtual machine* (ZKVM), bridging conventional ISAs with ZK protocols. This enables expressiveness on par with mainstream languages but demands highly sophisticated ZKVM implementations. Since ISAs already have precise formal specifications, this presents an excellent opportunity for verification. Can we combine automated and interactive methods to fully verify realistic ZKVMs?

LLMs for Mathematical Theorem Proving In Coda, we used the Rocq proof assistant to prove circuit correctness—proofs that are long and tedious. Large language models show early promise in assisting with formal proofs, yet current approaches fail to scale beyond toy examples. Can we combine LLMs' inductive reasoning with formal methods' deductive power to prove more complex theorems in interactive provers? One of my ongoing collaborations explores this synergy.

Supporting Undergraduate Research

I am passionate about guiding undergraduates into research, inspired by my own liberal arts education at Vassar College. At UCSB, I strive to recreate that supportive environment which kindled my interest in computer science research.

In general, I integrate elements of research into teaching to spark curiosity and foster a "research mindset" through discovery learning (see my teaching statement). After class, I also connect with students to learn their interests and match them with opportunities. For example, I have encouraged undergraduates in my class to audit or enroll in my graduate course on program verification, and recommended highly motivated students to faculty for research assistantships.

I had the fortune to mentor four undergraduates. My approach is to first understand each student's "taste" and strengths, then design scaffolded tasks that build toward concrete deliverables. For example, Yanning Chen was fascinated by programming language *theory*, so I carved out a design space for her to explore various alternatives for compiler IR; her design later ended up in our final compiler! For students who who are avid systems builders (Nicholas Brown, Hanzhi Liu and Jiaxin Song), I created scaffolded projects with clear system input-output specifications, and met regularly to discuss implementation trade-offs, offer debugging tips, and pair-program when they got stuck.

In every case, my mentees made measurable progress that led to top-conference publications. They have since pursued selective PhD programs (UIUC, UCSB, University of Toronto, and Cornell University) or engineering roles at major tech companies (e.g., Meta).

I look forward to continuing this tradition as a faculty member. I'm excited to involve students in my ongoing projects: students can freely choose from a range of topics, from theoretical explorations of new DSL designs for ZKP programming to hands-on implementation of synthesis algorithms and verification tools. I am committed to providing the mentorship and resources they need to thrive as researchers.

References

[1] Yanju Chen, Junrui Liu, Yu Feng, and Rastislav Bodik. Tree traversal synthesis using domain-specific symbolic compilation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1030–1042, New York, NY, USA, 2022. Association for Computing Machinery.

[2] Junrui Liu, Yanju Chen, Eric Atkinson, Yu Feng, and Rastislav Bodik. Conflict-driven synthesis for layout engines. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[3] Junrui Liu, Yanju Chen, Bryan Tan, Isil Dillig, and Yu Feng. Learning contract invariants using reinforcement learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[4] Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Isil Dillig, and Yu Feng. Certifying zero-knowledge circuits with refinement types. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1741–1759, Los Alamitos, CA, USA, May 2024. IEEE Computer Society.

[5] Junrui Liu, Jiaxin Song, Yanning Chen, Hanzhi Liu, Hongbo Wen, Luke Pearson, Yanju Chen, and Yu Feng. Tabby: A synthesis-aided compiler for high-performance zero-knowledge proof circuits. volume 9, New York, NY, USA, October 2025. Association for Computing Machinery.

[6] Veridise. Semaphore audit report. https://veridise.com/audits-archive/company/semaphore/semaphore-groups-v3-2023-01-05. [Online; accessed 05-Sep-2025].