Xiaohong Chen
https://xchen.page

My research interests are in **logic** and **formal methods** (FM), with a focus on improving the trustworthiness of programs, programming language implementations, and formal analysis tools.

Formal methods use various logics, calculi, formal semantics, and type systems to formalize programs and/or programming languages, specify their correctness claims, and prove those claims using nontrivial algorithms, encodings, translations, optimizations, and heuristics. However, there are always underlying assumptions, or the so-called trust base, whenever a correctness claim is made or proved using formal methods. Consider formal verification, which is the act of proving that a program satisfies its formal specification using a program verifier. In terms of trustworthiness, formal verification *transfers* the trust in the program being verified to that in the verifier and makes the trust base more manageable: instead of trusting countless unverified programs, one can verify them and trust a single program: the verifier itself. In other words, the correctness of a formally verified program can still be compromised if the verifier itself is buggy. Correctness claims that are made using formal methods cannot and should not be blindly trusted. We must be aware of the underlying trust base.

My **research vision** is to build a trustworthy, systematic, and efficient infrastructure to make, store, certify, and use—*with absolute certainty and a minimal trust base*—correctness claims about any programs, program properties, and programming languages. I call it a *trustworthy language framework*. Programming languages are defined using logical axioms and mathematical notations, known as their formal semantics. Language implementations and formal analysis tools are directly generated from the formal semantics. Programs properties, including not only formal verification properties, but also (concrete and symbolic) program executions, model checking properties, safety and liveness properties, and even results of parsing, are encoded as logical formulas in the framework. Correctness claims are made by various stakeholders, including language designers and standards committees, program developers, customers, and investors. These correctness claims are then certified by complete, rigorous, transparent, and human-accessible *proof certificates* and automatically checked by a simple program of hundreds of lines of code, known as the *proof checker*. This way, we reduce the trust base to a minimum. Programming language implementations, formal analysis tools, verification algorithms, proof strategies, and automated reasoning techniques need not belong to the trust base; instead, they become heuristics for finding and generating the final, corresponding proof certificates.

I pursue the above vision of a trustworthy language framework and unfold my research on the following **three themes**:

T1. *What logic* should such a trustworthy language framework be based on, that enables one to formally define any programming language and express any program property?
T2. *What proof system* enables one to formally reason about programs, prove any program properties, and automatically check the proofs using a minimal proof checker?
T3. How can we generate complete, rigorous, transparent, human-accessible, and machine-checkable proof certificates for programs, programming language implementations, and formal analysis tools? In short, how do we perform *proof generation*?

My **prior Ph.D. research** spans all three themes, and I will continue to work on these themes in the future. Regarding (T1), I co-designed *matching logic* (http://matching-logic.org) as a unifying logical foundation for programming languages [14][15], program specification [3][5][6][8][12][13], and program reasoning [2][4][6][9][11][13][15]. Regarding (T2), I designed a powerful proof system for matching logic with only 15 proof rules and implemented a proof checker using only 240 lines of code [9][16]. Regarding (T3), I initiated long-term research on *proof generation* to bring the best-known assurance levels to software and computers. As a proof of concept, I implemented a proof generator for concrete (non-symbolic) execution of imperative programs [9].

I work on both foundational theories and practical applications, and my research has had both **academic and practical impact**. On the academic side, matching logic has attracted attention from the global PL/FM community for its simplicity, elegance, expressive power, and reasoning power. Research teams, groups, and labs from King's College London (UK), University of Kent (UK), University of Paris-Saclay (France), Peking University (China), University of Bucharest (Romania), Alexandru Ioan Cuza University of Iaşi (Romania), Eötvös Loránd University (Hungary), and Masaryk University (Czechia) have started collaborating on the study of matching logic and the development of related tools. I have participated in all collaborations and I am leading the core projects.

On the practical side, I am cooperating with the startup Runtime Verification Inc. to transfer my research on matching logic and proof generation into practical products. I helped integrate matching logic into the K language framework (https://kframework.org) to improve its maintenance, reusability, correctness, and trustworthiness. My recent proposal on studying proof generation raised a $30,000-funding from the Ethereum Foundation for its potential to significantly increase the safety, reliability, and transparency of cryptocurrencies and smart contracts [20].

Prior Work

Matching Logic. Formal methods use rigorous mathematical tools to specify and verify computing systems. Since the 1960s, various formal semantics approaches have been proposed, including Floyd-Hoare axiomatic semantics, Scott-Strachey denotational semantics, initial algebra semantics, and operational semantics. Various formal logics and calculi have been used/proposed to express and reason about program properties, including first-order logic, higher-order logic, equational logic, fixpoint logic, separation logic, modal logic, temporal logic, dynamic logic, type systems, as well as their variants and extensions. A trustworthy language framework must have a mathematically solid, expressive, and minimal logical foundation to support all the above formal approaches and systems in terms of logical theories, axioms, and notations.

Built upon previous work, I co-designed matching logic [15] as a practical and unifying foundation for programming languages, program specification, and program reasoning, where all the above semantics approaches, formal logics, calculi, and type systems are definable as logical theories. I worked with the startup Runtime Verification Inc. (RV) to incorporate matching logic into the K framework—a practical formal language semantics framework that has been used to formalize real-world programming languages (C, Java, JavaScript, Python, Rust, x86-64, Ethereum VM, LLVM, etc.), to model and verify safety-critical systems by Boeing, DARPA, NASA, and Toyota, and to model and verify smart contracts, consensus protocols, programming languages, and virtual machines by ConsenSys, DappHub, Ethereum Foundation, Gnosis, IOHK, MakerDAO, Uniswap, and RV itself. This way, matching logic possesses complete definitions of the above real-world programming languages and serves as a basis for formal reasoning about programs.

Proof System and Proof Checking. The key question for a unifying logic such as matching logic is: *What set of proof rules does it use for formal reasoning, and how powerful are they*? To answer this question, I designed a 15-rule matching logic proof system and proved several theorems that demonstrate its completeness/effectiveness in various domains [15]. For example, I proved that the proof system is complete in the first-order (FO) fragment, extending the classical result on hybrid completeness to many-sorted and polyadic settings. I proved a novel global completeness property for all theories in the FO fragment that feature equality. I proved that the proof system is (relatively) complete for functional program correctness and thus offers a (relatively) complete program verification framework. I proved that initial algebra semantics are definable in matching logic and that various induction principles are derivable using the proof system, showing that matching logic offers a unifying logical framework for induction and recursion [13].

I implemented a matching logic proof checker using 240 lines of code, making it one of the smallest of its kind, serving as the minimal trust base. Matching logic proofs are encoded into proof certificates and checked by the proof checker. More than 100 matching logic lemmas/theorems were manually proved and encoded into a public database of matching logic formal theorems [15][17] for future research. A research team from Eötvös Loránd University (Hungary) uses the database to encode and check matching logic proofs carried out using the Coq proof assistant [18].

I implemented a prototype automated prover for matching logic. The prover consists of a set of high-level proof strategies in separate modules for reasoning about data structures, frames, fixpoints, and user-defined recursive properties, as well as invariant synthesis [2][11], and is shown to be generic and effective using a cross-logic benchmark with challenging properties from various logics.

Proof Generation. In 2020, I initiated a long-term research agenda on proof generation as a logically sound and practically feasible means to address the community's long-standing need for more trustworthy language implementations and analysis tools. As a proof of concept, I prototyped a proof generator for a semantics-based program interpreter [9]. The proof generator takes as input (1) a formal programming language semantics and (2) a program execution trace, and outputs a machine-checkable proof certificate that the execution trace is correct with respect to the given formal semantics. The technique has caught the industry's eye, especially of the blockchain community, because it provides the best-known assurance level for smart contracts. In 2022, the Ethereum Foundation awarded $30,000 for the proposal "Trustworthy Formal Verification for Ethereum Smart Contracts via Machine-Checkable Proof Certificates," which I helped write.

Other Work. I am also interested in (1) specifying and reasoning about computation beyond traditional scenarios and (2) developing more efficient semantics-based tools. Regarding (1), I co-designed MediK as an executable formal semantics of an interactive medical guidance system [8], which interacts with a web interface and forms an end-to-end guidance system for medical experts and physicians. I used matching logic to specify hybrid systems and properties [14] and hyper-properties [10], which can express complex security policies such as noninterference. I used matching logic as a unifying logical framework for neural networks, where various types of neural networks and their properties can be specified in a uniform way [5]. Regarding (2), I participate in developing KPLC, a high-performance executable semantics of programmable logic controllers (PLC). I co-authored [1] and presented an efficient algorithm that compiles the formal semantics of any programming language into high-performance LLVM code for fast program execution.

Future Directions

I first present my short-term research plan and then discuss my long-term vision.

My **five-year research plan** is to continue my research on the discussed themes (T1)-(T3). I will keep studying the fundamental problems of matching logic regarding its expressive power, completeness properties, decision procedures, and more efficient proof strategies. I will extend my proof-of-concept work on proof generation to support trustworthy and transparent semantics-based program execution and formal verification. I will continue developing tool support for matching logic, with a focus on bringing more interactive reasoning power to real-world programming languages by connecting matching logic with existing proof assistants.

I am eager to transfer my research into practical products and have identified two promising directions. The first is proof-certifying smart contracts, where matching logic proof certificates are generated for smart contract execution and verification, improving their trustworthiness. The second is to combine proof generation with succinct non-interactive arguments of knowledge (SNARKs) to obtain more succinct cryptographic proof certificates. Below, I elaborate on some of my planned research projects of potentially high impact.

Proof-Certifying Formal Verification. Formal verification transfers the trust in the program being verified to that in the verifier. *Proof-certifying formal verification* pushes one step further and eliminates program verifiers from the trust base, by generating proof certificates for formal verification. These proof certificates will make formal verification more transparent and accessible, as what is currently hidden behind the complex algorithms, transformations, heuristics, and optimization—the actual logic reasoning—will become explicit. The key component is a proof generator that takes as input (1) a formal programming language semantics and (2) a successful verification run of a given program, and outputs the proof certificate for the verification result. Since the proof certificate will be checked, neither the proof generator nor the verification algorithm belong to the trust base. I have started developing a prototype for the verification of deterministic programs [4]. Although the current prototype can only handle small academic programming languages, it is semantics-based and works with various programming paradigms, including imperative, functional, and assembly languages. I will extend the prototype to support non-deterministic programs and real-world programming languages.

I identify two promising directions to transfer proof-certifying execution and verification into impactful, practical products. The first is **proof-certifying smart contracts**. As of today, matching logic and K have been used to formalize and verify more than 8 blockchain consensus protocols, 43 smart contracts, 4 blockchain virtual machines, languages, and specifications [19]. Still, to trust the executions of these smart contracts or their verification results, one needs to trust an unverified implementation of a programming language or a virtual machine, or the unverified interpreter or verifier of K. My research will address the above issue by reducing computation to proof checking. Proof-certifying smart contracts will lead us to a next-generation blockchain, where smart contracts are executed only once. Execution results are published to the blockchain with their corresponding proof certificates, so validators need not re-execute transactions but check the proof certificates using a fixed checker—the only program running on-chain.

The second direction is **SNARK-ing the proof certificates**. The term SNARK, which stands for succinct non-interactive arguments of knowledge, refers to the production of a cryptographic proof where one can show the possession of certain information. SNARKs will significantly reduce the sizes of the proof certificates from millions or more lines of code to a few hundred bytes and reduce the proof checking time. The key idea is as follows. Instead of producing a huge proof certificate with the entire proof steps, one produces a cryptographic SNARK proof that only states that the proof generator possesses/knows the proof steps. That is, it states the *existence* of such a proof certificate. In addition, the SNARK proof can be generated on-the-fly along with the proof generation process, so one never needs to store the entire proof certificate. I have started cooperating with the startup, RISC-zero Inc. to develop a SNARK proof generator for matching logic and explore its application to proof-certifying smart contracts. I shared the latest progress at the New England Verification Day 2022.

Matching Logic Tool Support. Proof assistants such as Coq, Isabelle, and Lean are widely used in defining formal language semantics and reasoning about programs, with decent support for both interactive and automated reasoning. On the other hand, matching logic processes the complete formal definitions of many real-world programming languages. There is a great need for more interactive reasoning support for these languages, especially when the fully automated matching logic theorem prover fails to deliver. Therefore, I propose exporting the formal semantics in matching logic to existing proof assistants to get the best of both worlds. Matching logic will obtain more interactive reasoning power from these proof assistants, and users who are already familiar with these proof assistants can get access to the formal semantics of numerous real-world languages. I worked with a research team from Eötvös Loránd University (Hungary) to mechanize matching logic in Coq [7]. I am also involved in another parallel effort led by the Institute for Logic and Data Science at the University of Bucharest (Romania) to mechanize matching logic in Lean.

My **long-term research vision** is that of a *trustworthy language framework* that incentivizes programming language designers to design and implement their languages by defining a formal semantics and nothing else. All language tools are automatically generated from the formal semantics and certified by complete, rigorous, transparent, human-accessible, and machine-checkable proof certificates. Such a language framework would be easy to use, with an intuitive interface and user-friendly notations, yet built on a solid mathematical foundation; be modular and extensible, so it scales to real-world languages; be powerful and expressive, so it can define any programming languages and express any program properties; should allow us to define a language once and for all, in contrast to the state of the art where different tools require different models of the language; and most importantly, should have a minimal logical trust base that is fully comprehensible and accessible to all its users. Any form of computation or formal analysis that is carried out by the framework, including parsing a piece of code, executing a program, verifying it against its formal specification, etc., will be accompanied by a corresponding proof certificate and checked by one fixed proof checker. Programming language implementations, formal analysis tools, verification algorithms, proof strategies, and automated reasoning techniques need not belong to the trust base anymore because they become heuristics for finding and generating the final proof certificates.

References

[1] B. Collie, T. Kasampalis, X. Chen, D. Guth, G. Roşu, "An efficient language-agnostic semantics-based interpreter," PLDI Submission (under review).
[2] T. Trinh, X. Chen, N. Rodrigues, G. Roşu, "Automatic abstraction for fixpoint reasoning in matching logic," PLDI Submission (under review).
[3] X. Chen, G. Roşu, "Defining binders in matching logic (featuring a case study of contexts in K)," Journal of Functional Programming (JFP) ICFP Special Issue (under review).
[4] Z. Lin, X. Chen, T. Trinh, J. Wang, G. Roşu, "Generating proof certificates for a language-agnostic deductive program verifier," OOPSLA Submission (under review). Technical Report https://hdl.handle.net/2142/112785.
[5] X. Zhang, X. Chen, M. Sun, "Towards a unifying logical framework for neural networks," in Proceedings of ICTAC, 2022.
[6] X. Chen, D. Lucanu, G. Roşu, "Capturing constrained constructor patterns in matching logic," Journal of Logical and Algebraic Methods in Programming, vol. 130, 2022
[7] P. Bereczky, X. Chen, D. Horpácsi, T. Mizsei, L. Peña, J. Tusil, "Mechanizing matching logic in Coq," in Working Formal Methods Symposium (FROM), 2022.
[8] M. Saxena, X. Chen, S. Song, S. Meng, L. Sha, G. Roşu, "Rewriting-based computer-interpretable clinical practice guidelines," Technical Report https://hdl.handle.net/2142/116016, 2022.
[9] X. Chen, Z. Lin, T. Trinh, G. Roşu, "Towards a trustworthy semantics-based language framework via proof generation," in Proceedings of CAV, 2021.
[10] J. Tusil, X. Chen, G. Roşu, "Hyperproperties in Matching Logic," Technical Report https://hdl.handle.net/2142/109298, 2021.
[11] X. Chen, T. Trinh, N. Rodrigues, L. Peña, G. Roşu, "Towards A unified proof framework for automated fixpoint reasoning using matching logic," in Proceedings of OOPSLA, 2020.
[12] X. Chen, G. Roşu, "A general approach to define binders using matching logic," in Proceedings of ICFP, 2020.
[13] X. Chen, D. Lucanu, G. Roşu, "Initial algebra semantics in matching logic," Technical Report https://hdl.handle.net/2142/107781, 2020.
[14] M. Saxena, X. Chen, N. Rodrigues, G. Roşu, "Formal semantics of hybrid automata," Technical Report https://hdl.handle.net/2142/106822, 2020.
[15] X. Chen, G. Roşu, "Matching μ-logic," in Proceedings of LICS, 2019.
[16] GitHub, Matching logic proof checker, https://github.com/kframework/proof-generation/blob/main/theory/matching-logic-240-loc.mm.
[17] GitHub, Matching logic theorem database, https://github.com/kframework/proof-generation/tree/main/theory.
[18] GitHub, AML Formalization, https://github.com/harp-project/AML-Formalization.
[19] GitHub, Runtime Verification Publications, https://github.com/runtimeverification/publications
[20] News Article, https://cs.illinois.edu/news/new-academic-research-grant-through-ethereum-will-increase-safety-of-smart-contracts-and-blockchain-applications