Xiaohong Chen
https://xchen.page

Research Statement
Last Update: November 21, 2022

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

Formal methods use various logics, calculi, formal semantics approaches, 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 trust base, whenever a correctness claim is made or proved using formal methods. Oversight of the underlying assumptions can lead to a compromise of correctness. My **research vision** is to build a trustworthy, systematic, and efficient infrastructure to make, store, use, cite, and assert---*with absolute formal guarantees*---correctness claims about any programs, program properties, and programming languages. My research is unfolded on the following **three themes**:

T1. How to use *one fixed logic* to axiomatically define the formal semantics of any programming languages and express any program properties?
T2. How to use *one fixed proof system* to reason about programs using directly the formal semantics, prove any program properties, and check them using *one fixed proof checker*?
T3. How to automatically generate complete, rigorous, transparent, human-accessible, and machine-checkable logic proofs as formal correctness guarantees of programs, programming language implementations, and formal analysis tools? In short, how to do *proof generation*?

My **prior Ph.D. research** spans all three themes, and I will continue to work on them in the future. Regarding (T1), I proposed *matching logic* (http://matching-logic.org) as a unifying logic foundation for programming languages [1], program specification [2-5], and program reasoning [1,2,6,7]. Regarding (T2), I designed a powerful proof system for matching logic with only 15 proof rules and implemented a matching logic proof check with only 240 lines of code [8,9]. 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 impacts**. 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), Peking University (China), University of Bucharest (Romania), Alexandru Ioan Cuza University of Iaşi (Romania), Eötvös Loránd University (Hungary), and Masaryk University (Czechia) have started collaborations on the study of matching logic and the development of related tools. I have participated in all collaborations and am the leader of 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 to improve the formal correctness guarantees of the semantics-based tools in the K framework (https://kframework.org). 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.

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. *How can we capture these semantics approaches, formal logics, and calculi in one unifying logic, where each of them is faithfully defined as a logical theory using axioms and notations*?

I proposed matching logic as an answer to the above research question. Matching logic is a practical and unifying foundation for programming languages, program specification, and program reasoning, where all the above semantics approaches, formal logics, and calculi are definable as logic theories. I worked with the startup Runtime Verification Inc. to incorporate matching logic into the K framework, which is 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, Runtime Verification Inc., and Uniswap. Matching logic is now the logic foundation of K, with all the complete formal semantics of the above real-world programming languages.

Proof System and Proof Checker. A key question to 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 Hilbert-style proof system for matching logic with only 15 proof rules and proved several completeness properties [1]. For example, I proved that the first-order (FO) fragment of matching logic is complete, 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 the (relative) completeness property for reasoning about programs' functional correctness, showing that matching logic is a (relatively) complete formal verification framework. I proved that initial algebra semantics are definable in matching logic, and various induction principles are derivable using the proof system, showing that matching logic is a unifying logic framework for induction and recursion [2].

I implemented a matching logic proof checker with a minimal, 240 lines of trusted code base, making it one of the smallest of its kind. Matching logic proofs are encoded into proof objects 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 [9,10] for future research. For example, a research team from Eötvös Loránd University (Hungary) uses the database to encode and check matching logic proofs carried out using the Coq theorem [11].

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, contexts, fixpoints, and user-defined recursive properties, as well as invariant synthesis [7,12]. The prover is generic and effective, as shown on a cross-language benchmark. For example, the prover successfully proved all 280 challenging heap properties in the SL-COMP competition for separation logic provers, when instantiated by the matching logic theory of finite-domain heaps.

Proof Generation. In 2020, I initiated a long-term research agenda in my Ph.D. thesis proposal 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. I prototyped a proof generator for a semantics-based interpreter proof of concept [9]. The proof generator takes two inputs: a formal semantics of a programming language and a program execution trace. It outputs a machine-checkable proof object that certifies that the execution trace is correct with respect to the formal semantics. The proof generation technique has caught the industry's eye, especially the blockchain community, for it can provide the best-known assurance levels for smart contracts. In 2022, the Ethereum Foundation offered $30,000 in funding 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) using matching logic to specify and reason 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 [13], 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]. I used matching logic as a unifying logic framework for neural networks, where various types of neural networks and their properties can be specified in a uniform way [3]. Regarding (2), I am participating in developing KPLC, a high-performance executable formal semantics of programmable logic controllers (PLC). I co-authored [15] and presented an efficient internal algorithm of K that compiles formal semantics into high-performance LLVM code for fast execution of concrete (non-symbolic) programs.

Future Directions

I will continue pursuing my research agenda on matching logic and proof generation to improve formal correctness guarantees of programs, programming languages implementation, and formal analysis tools. My methodology is to generate complete, rigorous, transparent, human-accessible, and machine-checkable proof objects as formal correctness certificates for any properties of any programs in any programming languages. Each proof object clearly states the program property being proved and the underlying theory of a formal programming language semantics in one fixed logic---matching logic---as well as the entire proof steps to be automatically checked by a small, fixed proof checker, thus reducing the trusted code base to the minimum. Any programming language implementations, formal analysis tools, verification algorithms, or proof strategies will no longer belong to the trusted code base but are heuristics of finding and generating the final formal correctness certificates.

My **five-year research plan** is to continue my research on the three themes (T1)-(T3). I propose to 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 via 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 objects are automatically generated for smart contracts execution and verification, bringing their trustworthiness to a high level. The second is to combine proof generation with succinct non-interactive arguments of knowledge (SNARKs) to obtain more succinct cryptographic proof certificates for all programs in all programming languages. Below, I elaborate on some research projects of potentially high impact.

Proof-Certifying Formal Verification. Formal verification is the act of proving the correctness of a program with respect to its formal specification. In terms of trustworthiness, it *transfers* the trust in the program being verified to that in the verifier. As a result, it makes the trusted (or, one may say, untrusted) code base more manageable. Instead of trusting countless unverified programs, one can verify them and trust the verifier. I propose to study proof-certifying formal verification to eliminate verifiers from the trusted code base of formal verification, by developing a proof generator that takes two inputs: a formal semantics of a programming language and a successful verification run that verifies the correctness of a given program. It outputs a correctness certificate for the verification result. I have started developing a prototype that outputs correctness for the verification of deterministic programs [16]. 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**. Today, matching logic and K have been used to formalize and verify more than seven blockchain consensus protocols, 31 smart contracts, and four blockchain-oriented programming languages. 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 semantics-based interpreter or verifier of K. My research will address the above issue by reducing computation or formal analysis to machine-checkable proof certificates. These proof certificates will make smart contract verification more transparent and accessible to stakeholders and customers, as what is currently hidden behind the complex algorithms, transformations, heuristics, and optimization---the actual logic reasoning---will become explicit. 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 the 240-line matching logic proof checker, which is 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 generated for program execution and formal verification, from millions or more lines of code to a constant size. The key idea is to produce a cryptographic SNARK proof that states that there exists a sequence of proof steps for a given execution trace or a verification run, which will pass the checking of the fixed, matching logic proof checker. This way, we can generate SNARKs as succinct proof certificates for any properties of any programs in any programming languages. I have started cooperating with the startup RISC-zero Inc. to develop a SNARK proof generator for matching logic. I was invited to the New England Verification Day 2022 to share the research progress.

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 them, 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 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. In [17], I co-worked with a research team from Eötvös Loránd University (Hungary) and mechanized matching logic in Coq. I am also involved in another parallel effort led by the Institute for Logic and Data Science at the University of Bucharest (Romania) in mechanizing matching logic in Lean.

References

[1] X. Chen and G. Roşu, "Matching μ-logic," in Proceedings of LICS, 2019.

[2] X. Chen, D. Lucanu and G. Roşu, "Initial algebra semantics in matching logic," Technical Report https://hdl.handle.net/2142/107781, 2020.

[3] X. Zhang, X. Chen and M. Sun, "Towards a unifying logical framework for neural networks," in Proceedings of ICTAC, 2022.

[4] X. Chen and G. Roşu, "A general approach to define binders using matching logic," in Proceedings of ICFP, 2020.

[5] X. Chen and 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), 2023.

[6] X. Chen, D. Lucanu and G. Roşu, "Capturing constrained constructor patterns in matching logic," Journal of Logical and Algebraic Methods in Programming, vol. 130, 2022.

[7] X. Chen, T. Trinh, N. Rodrigues, L. Peña and G. Roşu, "Towards A unified proof framework for automated fixpoint reasoning using matching logic," in Proceedings of OOPSLA, 2020.

[8] Github, Matching logic proof checker, https://github.com/kframework/proof-generation/blob/main/theory/matching-logic-240-loc.mm, 2021.

[9] X. Chen, Z. Lin, M.-T. Trinh and G. Roşu, "Towards a trustworthy semantics-based language framework via proof generation," in Proceedings of CAV, 2021.

[10] Github, Matching logic theorem database, https://github.com/kframework/proof-generation/tree/main/theory, 2022.

[11] Github, AML Formalization, https://github.com/harp-project/AML-Formalization, 2022.

[12] T. Trinh, X. Chen, N. Rodrigues and G. Roşu., "Automatic abstraction for fixpoint reasoning in matching logic," PLDI Submission (under review), 2023.

[13] M. Saxena, X. Chen, S. Song, S. Meng, L. Sha and G. Roşu, "Rewriting-based computer-interpretable clinical practice guidelines," Technical Report https://hdl.handle.net/2142/116016, 2022.

[14] M. Saxena, X. Chen, N. Rodrigues and G. Roşu, "Formal semantics of hybrid automata," Technical Report https://hdl.handle.net/2142/106822, 2020.

[15] B. Collie, T. Kasampalis, X. Chen and G. R. D. Guth, "An efficient language-agnostic semantics-based interpreter," PLDI Submission (under review), 2023.

[16] Z. Lin, X. Chen, M.-T. Trinh, J. Wang and G. Roşu, "Generating proof certificates for a language-agnostic deductive program verifier," OOPSLA Submission (under review), 2023.

[17] P. Bereczky, X. Chen, D. Horpacsi, T. Mizsei, L. Pena and J. Tusil, "Mechanizing matching logic in Coq," in Working Formal Methods Symposium (FROM), 2022.