Research Statement

Yao Li

liyao@cis.upenn.edu
https://lastland.github.io

Formal verification is useful for building reliable software: it helps discover bugs, provides machine-checked and explicit documentation of invariants, and offers a deeper understanding of code. However, formal verification has also been conceived as hard, uneconomical, or impractical, because it is typically less accessible and takes more effort than other approaches like testing. In other words, provably reliable software is considered a luxury.

As a researcher who works on machine-checked formal verification, I believe formal reasoning should be an intrinsic part of software development. **To realize this vision, my research aims to (1) advance the state of the art of verification on real-world software and (2) make verification easier to use from a *programming languages* perspective.** The Curry-Howard isomorphism—a celebrated discovery in the field of programming languages—states that propositions are types and proofs are programs. We have made much progress in making programming easier in the last few decades, there is no reason we cannot do so for verification.

Guided by this vision, I have worked on a broad spectrum of research on verification. My research topics span from reasoning about offline systems like compilers [1] to online systems like networked servers [2,3], from verifying purely functional programs written in Haskell [1,4] to imperative programs written in C [2,3], from specifying functional correctness [1-6] to performance [7]. My research has been published in premier conferences on programming languages and formal verification, such as ICFP, ITP, and CPP.

My research is funded by NSF expeditions in computing "The Science of Deep Specification," a joint expedition of seven PIs from four institutions (Princeton, Penn, Yale, MIT), and NSF grant "Mechanized Reasoning for Functional Programs," a grant I co-wrote with my advisor Prof. Stephanie Weirich.

Past and Ongoing Research

The DeepSpec Web Server. Almost every piece of software depends on assumptions made about other software, which can be external libraries, operating systems, and/or compilers, *etc.* To ensure that a piece of software is correct, we also need to prove the correctness of its "underlying" software. The DeepSpec project is a joint expedition of four institutions (Princeton, Penn, Yale, MIT) that aims to build such full stack verified systems. As part of the expedition, we demonstrated how to specify, test, and verify a networked server [2,3]. The server follows the HTTP protocols to perform some simple tasks. The server is written in C. It is compiled by CompCert, a verified C compiler, and runs on CertiKOS, a verified operating system.

As one of the student leaders of this project, I focused on two questions. First, how can we develop a specification for complex systems like a networked server? The ideal specification should permit different implementation models but also only accept correct ones. On the practical side, it should also be easy to understand so that it can be inspected by humans. Developing such a specification is particularly challenging for networked servers, as servers are typically implemented using various concurrency models. Our work proposed the use of a high-level linear specification to describe the servers' behaviors in a straightforward and easy-to-read "one client at a time" style. A correct implementation that employs an implementation model like the event-driven model, can be shown to behave observationally (by observing over the network) the same as the linear specification via *network refinement*, a variant of linearizability we proposed that accounts for possible network delay and reordering.

Second, how can we integrate disparate testing and verification work that was done separately using different specification styles? For example, the Verified Software Toolchain (VST) contains a set of tools for verifying C programs and its specifications are based on Hoare-style separation logic; the verified operating system CertiKOS uses functional specifications with an “oracle” that represents the external environment; the property-based testing tool QuickCheck uses executable functional specifications. We proposed *interaction trees*, a general structure for representing reactive computations, as the *lingua franca* that ties these tools together. Our work shows that we can use interaction trees to represent effectful programs, use them for testing, integrate them in Hoare-style separation logic, and use them to axiomatize system calls provided by CertiKOS.[1]

Program Adverbs. Even vastly different programming languages can share similar characteristics. As a concrete example, let’s look at Makefile and a register-transfer level (RTL) design language like Verilog: even though they are used for different applications, they share common characteristics that (1) they both enforce a static data dependency and (2) they are executed in parallel (for “make -j”). In formalizing programming languages, we often repeat defining syntactic or semantic structures that capture similar characteristics like these. Wouldn’t it be nice if this type of characteristics can be crystalized using some reusable structures and implemented as libraries? This is exactly where our work [5] comes in.

We proposed a class of structures called *program adverbs* based on inspirations from functional programming. Each program adverb represents some characteristic of a language’s syntax and/or semantics. For example, “statically” is an adverb that represents the data dependencies it quantifies over are all statically known at compile time; “statically in parallel” is another adverb that is similar to “statically”, but all the effects it quantifies over are executed in parallel. These adverbs enable proving general properties that are true for corresponding characteristics.

Besides making program adverbs reusable structures, we took a step further to define program adverbs in a composable way, allowing a tailored model for languages containing different characteristics. We identify five basic program adverbs and two add-on adverbs and demonstrate that program adverbs are useful general structures via three distinct examples: a circuit, a data retrieval library that batches queries, and a simplified networked server that resembles the style of the DeepSpec web server [2].

Simple Reasoning about Laziness. Lazy evaluation is a distinguished feature in functional programming. It enables better compositionally in the code and potentially better performance at runtime. However, the Achilles' heel of lazy evaluation is that it makes the performance analysis much more complicated. First, the evaluation of multiple values is *interleaved* in a lazy program, which forces the analyzer to keep track of various states (and higher-order states). Second, a partially evaluated value might get evaluated further in the future, therefore a performance analysis cannot be done *locally*.

Instead of dealing with the complexity directly, we proposed a novel and simple framework for reasoning about lazy computation costs [7]. Our work avoids the *interleaved* computation by using *nondeterminism* to simulate an “oracle” that knows *exactly* the demand of every value in advance. With a clairvoyant knowledge of the demand, we can compute only the needed value in advance without changing the computational cost. This key idea is based on prior work by Hackett and Hutton [8]. Our work showed that this idea can be embedded using a simple[2] monad called the clairvoyance monad. We also presented an automatic translation strategy from a lambda calculus with folds to the monadic semantics.

To make the performance analysis *local* and modular, we proposed a programming logic based on dual specifications: a pessimistic specification that describes all—but possibly over-estimated—costs and an optimistic specification that describes some branches with more accuracy. Our paper demonstrated that the programming logic can be effectively used to reason about laziness via classic examples regarding tail recursion.

Verifying Haskell. How far are we from verifying existing and unmodified real-world Haskell programs? Our research explored this question based on a tool called hs-to-coq, which automatically translates total Haskell programs to Coq using shallow embeddings. We verified two commonly used real-world Haskell programs: (1) the containers library, which implements data structures like the weight-balanced trees and Patricia tries [4] and (2) parts of the Glasgow Haskell Compiler [1]. Both programs are written with a heavy focus on performance and are used extensively by industrial Haskell

[1] Interaction trees have been further developed as a library for representing recursive and impure programs in Coq by some of my co-authors [9].

[2] The monad is simple in that it can be defined using around 20 lines of code in Coq.

developers. We showed how we can address some theoretical and practical challenges in verifying real-world programs. Our work is also the pioneering work that showed that verifying large-scale, unmodified real-world Haskell programs using theorem provers is possible.

Future Research

Verifying Real-World Distributed Systems. Ensuring the reliability of distributed systems is much more tricky than single-machine programs: It is hard to get distributed systems right and even harder to effectively test them. I believe this is where formal verification can help greatly. Existing verification projects mostly focus on protocols, but a distributed system consists of various components. Failures in the implementation detail of these components or the subtle interaction among components could bring down the entire system as well. Therefore, an impactful research direction is verifying real-world distributed systems in a way that both high-level protocols and low-level implementation details are considered.

The project is a spiritual successor of the DeepSpec expedition, which studied the connection between specifications. Compared with DeepSpec, there are many new challenges. For example, the components in a distributed system are connected via a network, where packets could drop or get reordered, where protocols at different levels with different purposes run together. The project has a significant practical impact as many existing systems today are distributed. It also raises many interesting research questions, such as: What are the criteria of a *good* specification for a distributed system? How to connect the commonly used specifications in distributed systems like session types, π-calculus, temporal logic, and separation logic? What is a good practice for managing verification that is potentially very large? What techniques can be used to scale verification to large real-world distributed systems? How to adapt verification to new components or topological changes in the network?

To start, I would like to investigate a distributed database implemented using microservices. My prior work on verifying networked servers [2,3] qualifies me to work on the project. I am also excited to collaborate with experts in distributed systems and computer networks.

Gradual Formalization. What prevents programmers from verifying their programs? Suppose that you want to verify your program using some parallel framework, but no syntax or semantics representation (the step of defining the representation is known as *semantic embeddings*) of the framework exists yet. You can define the representation yourself, but the work is usually nontrivial. Therefore, I believe it is *crucial* to provide easy ways for semantic embedding if we want to make verification more practical.

I propose we should investigate a *gradual* and *reusable* approach to semantic embedding. Graduality allows us to only embed a subset of the language, jump-start reasoning about applications that only use the subset, and grow the embeddings later without redoing all the previous work. To achieve graduality, one challenge is a modular way of adding both new cases and functions to an existing data type without recompilation.[3] This is an old problem with many existing workarounds—each of them makes different trade-offs for the lack of support in the host language. Unfortunately, these trade-offs, both theoretical ones and practical ones, have not been well studied. While a silver bullet solution to the expression problem might not exist, I would like to conduct a study to better understand the trade-offs made by different workarounds. I would also like to investigate what language support can we add to better support existing and new solutions.

Reusability, on the other hand, allows us to save effort for common language features. Existing formalization work often repeats similar syntactic and semantic definitions as well as proofs, so libraries that contain reusable syntactic or/and semantic structures would help greatly. My work on program adverbs [5] is an example of such a library, but there are more syntactic structures (*e.g.*, assignments) or semantic structures (*e.g.*, memory model) that require further investigation. When designing these libraries, I would also like to look beyond conventional languages and consider language features from, *e.g.*, quantum languages, probabilistic languages, declarative languages for neural networks, *etc.*

[3] The problem is also known as the expression problem.

Formal Methods as Profilers. I believe that verification is not just useful as validation of programs' correctness, the verification insights can also be useful for writing and optimizing our code. One such place to demonstrate this vision is in formally analyzing and comparing different evaluation strategies—especially when lazy evaluation is involved.

Originating in functional programming, lazy evaluation has been adopted in many mainstream programming languages like Python, Java, and C++11, *etc.* Lazy evaluation is also crucial for many applications like build systems, stream processing, and data querying,[4] *etc.* However, the problem that it is hard to understand the performance under lazy evaluation has made laziness a constant source of surprise and has hindered its wider adoption.

My proposal is developing a framework for formally analyzing and *comparing* different evaluation strategies on the same piece of code. Furthermore, we can develop the framework into a tool that "coaches" programmers in optimizing their code by suggesting where to apply a more eager/lazy evaluation strategy. The tool can also be used to guide compiler optimizations, if a certain level of automation and precision is achieved in some subsets of the language.

The interesting research questions involved in this work include: What is a good way to compare performance under different evaluation strategies? Is there a programming logic that facilitates the comparison of performance? How to model data structures with mixed evaluation strategies? How to reason about the interaction between different evaluation strategies? How much automation can we achieve in the analysis and comparison? Can we combine the analysis with other analysis techniques like dynamic analysis, runtime verification, *etc.*? My prior work on analyzing lazy evaluation [7] can serve as a starting point for this work. The research is also closely related to systems and software engineering research and offers a good opportunity for cross-domain collaboration.

Works Cited

[1] Antal Spector-Zabusky, Joachim Breitner, **Yao Li** and Stephanie Weirich, "Embracing a mechanized formalization gap," *CoRR,* vol. abs/1910.11724, 2019.

[2] Nicolas Koh, **Yao Li**, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce and Steve Zdancewic, "From C to interaction trees: specifying, verifying, and testing a networked server," in *the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, Cascais, Portugal, 2019.

[3] Hengchu Zhang, Wolf Honoré, Nicolas Koh, **Yao Li**, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce and Steve Zdancewic, "Verifying an HTTP Key-Value Server with Interaction Trees and VST," in *The 12th International Conference on Interactive Theorem Proving, ITP 2021*, Rome, Italy (Virtual Conference), 2021.

[4] Joachim Breitner, Antal Spector-Zabusky, **Yao Li**, Christine Rizkallah, John Wiegley, Joshua Cohen and Stephanie Weirich, "Ready, Set, Verify! Applying hs-to-coq to real-world Haskell code," *Journal of Functional Programming,* vol. 31, e5, 2021. First published at ICFP 2018.

[5] **Yao Li** and Stephanie Weirich, "Program adverbs: Structures for embedding effectful programs," 2021. In submission.

[6] Jay Bosamiya, Sydney Gibson, **Yao Li**, Bryan Parno, Chris Hawblitzel, "Verified Transformations and Hoare Logic: Beautiful Proofs for Ugly Assembly Language," in The 12th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE), Los Angeles, CA, 2020.

[7] **Yao Li**, Li-yao Xia and Stephanie Weirich, "Reasoning about the garden of forking paths," *Proceedings of the ACM on Programming Languages,* vol. 5, no. ICFP, pp. 80:1--80:28, 2021.

[8] Jennifer Hackett and Graham Hutton, "Call-by-need is clairvoyant call-by-value," *Proceedings of the ACM on Programming Languages,* vol. 3, no. ICFP, pp. 114:1--114:23, 2019.

[9] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce and Steve Zdancewic, "Interaction trees: representing recursive and impure programs in Coq," *Proceedings of the ACM on Programming Languages,* vol. 4, no. POPL, pp. 1--32, 2019

[4] Some examples include the Nix build system and the LINQ library used in C#.