Research Statement

Runming Li

My research studies programming languages through the lens of (dependent) type theory and category theory, with the goal of better supporting mechanized reasoning about different aspects of programming. I treat dependent type theory both as a mathematical object to be studied and as a metalanguage in which we specify and verify properties of computations. Dependent type theory enables mechanized reasoning, which is becoming an essential tool for understanding and verifying software systems and mathematical arguments. My recent work can be broadly categorized into two complementary directions.

On one side, I investigate new ways to formulate and mechanize the meta-theory of programming languages and type systems in a modular and reusable manner, akin to how semanticists reason on paper. On the other side, I develop type-theoretic frameworks that build synthetic abstraction boundaries, so that mechanized verification of cost and behavioral properties of algorithms and data structures can follow the same modular principles as good library design. Together these two lines of work aim at using dependent type theory as a unified synthetic foundation for mechanized reasoning about programming languages and modular verification.

Mechanized Meta-Theory of Programming Languages

As features of programming languages and type systems grow more complex, pen-and-paper proofs of their meta-theoretic properties, such as progress and preservation, compiler correctness, normalization, and termination, become increasingly error-prone and tedious to maintain. Mechanizing meta-theory in proof assistants systematically handles the many detailed cases that such proofs require and provides a higher degree of confidence in their correctness. The traditional approach is to encode the syntax of the language in a proof assistant and then carefully carry out inductive proofs over this encoding. My recent work on mechanizing a novel style of operational semantics [1, to appear: **POPL 2026**] follows this approach: my collaborators and I designed a style of operational semantics that is more ergonomic and obtained fully mechanized proofs of core meta-theoretic properties in the Agda proof assistant.

The downside of this traditional, analytic approach is two-fold: first, mechanized proofs are often tightly coupled to the specific syntax and semantics of the language being studied, making it difficult to reuse proof patterns across different languages; second, mechanized proofs tend to be low-level and detailed, obscuring the high-level ideas that guide pen-and-paper proofs. For example, a common pitfall is that proofs about substitutions in variants of lambda calculus often require tedious and repetitive reasoning about variable binding and renaming that distracts from the main proof ideas and are hard to generalize to other languages.

To address these challenges, I approach mechanized meta-theory from a *synthetic* perspective, using type-theoretic and categorical structure to capture high-level proof patterns in a reusable, modular way. For example, I proved meta-theoretic properties of a complex dependent call-by-push-value language using *synthetic Tait computability* [2], a modal type-theoretic framework that internalizes logical-relations arguments. This allows us to carry out high-level proofs without getting bogged down in low-level syntactic details: many tedious aspects, such as substitution and naturality conditions, are hidden behind the abstractions of the modal type theory.

On paper these proofs are elegant and concise, but mechanizing them still incurs significant overhead, which motivates my recent work on mechanizing synthetic Tait computability [3, to appear: **CPP 2026**]. A well-known obstacle is that proof assistants based on intensional type theory, such as Agda and Coq, require extensive bookkeeping of propositional equalities. My collaborators and I circumvent this by working in Istari, an experimental proof assistant based on extensional type theory. Our mechanization shows that extensional type theory as a practical metalanguage can bring many advantages for mechanizing complex meta-theoretic arguments, in a style that is much closer to on-paper proofs. Moreover, the development is modular: the same synthetic constructions can be reused to prove meta-theoretic properties of different type theories. For instance, function types with the same categorical structure in different languages share the same synthetic proofs, which can be reused in our setting. Concretely, we developed a reusable library

for synthetic meta-theoretic reasoning in Istari and used it to mechanize meta-theoretic proofs for two different dependent type theories. This work is a step toward a systematic framework in which sophisticated logical-relations arguments can be mechanized synthetically.

Modular Verification in Dependent Type Theory

My undergraduate work [4] studied the verification of cost and behavioral correctness for parallel join-based red-black tree algorithms. In the Agda proof assistant, we mechanized a standalone development of red-black trees, proving that the join operation preserves the red-black invariants and that the resulting parallel algorithms achieve logarithmic work (sequential cost) and span (parallel cost).

The main difficulty is scaling such verification to downstream client code, *e.g.*, a graph algorithm that uses red-black trees as a library, which I began to study in my doctoral research with collaborators. In proof assistants like Agda, client proofs inevitably see the internal representation of the data structure, such as whether a node is red or black. This breaks data abstraction and burdens clients with irrelevant details: from the perspective of a graph algorithm verifier, a red-black tree is just one efficient implementation of a finite map, no more special than a list-based one. In common programming languages such as Standard ML and OCaml, module systems hide these representations behind interfaces, but this story does not carry over to proof assistants based on dependent type theory. To illustrate this problem, addition on natural numbers can be defined by induction on either the first or the second argument; although these definitions can be proved equal in an appropriate sense, their internal shapes differ, so replacing one with the other would necessarily break downstream proofs that depend on the specific definition.

To address this challenge, my collaborators and I proposed a *synthetic* approach to data abstraction in dependent type theory [5, to appear: **POPL 2026**]. The key idea is that concrete implementations and abstract interfaces form a *phase distinction* captured type-theoretically by a modal system with two modalities: one exposes the private, algorithmic representation, and the other exposes only the public, behavioral interface. This structure validates a noninterference principle: client code written in the abstract phase cannot depend on private implementation details or on cost information that is meant to remain internal, achieving modularity. Unlike traditional, uniform module systems, this phase-based approach supports data abstraction driven by arbitrary domain-specific knowledge supplied by the user.

At the heart of this work is a fracture theorem that internalizes the classical notion of an abstraction function, mapping concrete implementations to their abstract representations. Every type can be decomposed into a concrete component, an abstract component, and an abstraction function relating them, and then reconstructed as a glue type that packages these pieces together. This "abstraction functions as types" viewpoint turns the abstraction-function methodology from an external proof pattern into a first-class type-theoretic construction. The modal structure we use arises naturally from univalent foundations; while univalent foundations and homotopy type theory have been successful in formalizing mathematics, their applications to computer science are still under-explored, and our work opens a path toward using univalent ideas for modular mechanization in dependent type theory.

This framework has already shown practical value: I mentored two undergraduate students who carried out independent mechanization projects inspired by this work, one verifying the parallel cost of a prefix-sum algorithm [6] and the other verifying the amortized cost of the self-adjusting splay tree data structure [7].

Future Research Directions

Looking ahead, I aim to develop new techniques and tools that facilitate type-theoretic mechanized reasoning. For example I am currently looking into extending the synthetic approach to meta-theory to support compiler verification, in particular for ML-style module calculi. Such module systems often exhibit a similar phase distinction between static (types, kinds, and signatures) and dynamic (terms and modules) components, and I am looking into using the univalent-inspired modal structure from our earlier work to synthetically capture compiler correctness criteria in this setting.

References

[1] David M Kahn, Jan Hoffmann, and **Runming Li**. "Big-Stop Semantics: Small-Step Semantics in a Big-Step Judgment". In: *Proc. ACM Program. Lang.* 10.POPL (Jan. 2026). DOI: 10.1145/3776718. URL: https://doi.org/10.1145/3776718.

[2] **Runming Li** and Robert Harper. *Canonicity for Cost-Aware Logical Framework via Synthetic Tait Computability*. 2025. arXiv: 2504.12464 [cs.PL]. URL: https://arxiv.org/abs/2504.12464.

[3] **Runming Li**, Yue Yao, and Robert Harper. "Mechanizing Synthetic Tait Computability in Istari". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP '26. Rennes, France: Association for Computing Machinery, 2026. ISBN: 979-8-4007-2341-4/2026/01. DOI: 10.1145/3779031.3779085. URL: https://doi.org/10.1145/3779031.3779085.

[4] **Runming Li**, Harrison Grodin, and Robert Harper. *A Verified Cost Analysis of Joinable Red-Black Trees*. 2023. arXiv: 2309.11056 [cs.PL]. URL: https://arxiv.org/abs/2309.11056.

[5] Harrison Grodin, **Runming Li**, and Robert Harper. "Abstraction Functions as Types". In: *Proc. ACM Program. Lang.* 10.POPL (Jan. 2026). DOI: 10.1145/3776673. URL: https://doi.org/10.1145/3776673.

[6] Andrew Zhou. *Formally Verified Cost of the Parallel Prefix Sum Algorithm*. Tech. rep. Carnegie Mellon University, 2025. URL: https://www.cs.cmu.edu/~runmingl/student/zhou-scan.pdf.

[7] Lukas Kebuladze. *Amortized Analysis of Splay Trees via a Lax Homomorphism*. Tech. rep. Carnegie Mellon University, 2025. URL: https://www.cs.cmu.edu/~runmingl/student/kebuladze-splay.pdf.