Research Statement

GUANNAN WEI

My research interests lie in the scientific and engineering aspects of programming. I study programming languages and compiler techniques to help programmers build correct, safe, and performant software systems. In pursuit of these goals, I seek elegant and general solutions to pragmatic problems. With a keen eye on adjacent areas, I apply my expertise to tackle pressing and nascent issues in security, smart contracts, and quantum programming. In short, my work focuses on designing *better programming languages, compilers, and program analyzers* and their applications in existing and emerging domains:

- **Language Design & Type Systems**: Programmers are increasingly adopting low-level languages with richer type systems. For example, Rust provides memory safety in a low-level language by statically tracking ownership and lifetimes. However, adapting successful techniques from Rust to higher-order languages is quite challenging, due to the tension between Rust's "shared XOR mutable" principle and the pervasive capturing/escaping from functional and type-level abstractions.

  My research [1–3, 7, 17] aims to reconcile this tension by designing type systems for higher-order languages that can track rich program properties, such as lifetime, aliasing, and side-effects. Instead of restricting sharing, we developed *reachability types* [1] with the core mechanism that *tracks sharing/separation in higher-order languages*. On top of that, I have further developed effect systems [1], precise reachability polymorphism [7], aggressive optimizations [3, 17], and the Diamond prototype language [12].
- **Metaprogramming Program Analyzers**: Program analyzers are vital tools to reason about program behaviors without executing these programs. However, despite their usefulness in bug finding and security analytics, it is still challenging to build *correct, performant, and flexible analyzers*, due to the gap between high-level semantic specifications and their corresponding implementations.

  My dissertation work [6] investigated the construction of program analyzers using (meta)programs, treating *program analyzers as data objects to be analyzed, transformed, and generated using metaprogramming techniques*. I developed compilation [8, 11, 14, 16], derivation [10], and transformation [4] techniques to build correct-by-construction, performant, and flexible analyzers, enabling the use of higher-level abstractions without compromising performance or correctness.
- **Applications**: Using metaprogramming techniques, I developed next generation static analyzers [11, 18] for more efficient and effective bug finding. Borrowing "design-by-contract" ideas from PL community, my ongoing work is to design Solidity extensions [15] for safer and expressive smart contract programming.

  Building on my prior work on reachability types [1, 7], I have started investigating designing flexible and safe languages for quantum computing, exploiting the connection between quantum semantics (e.g., entanglement/measurement) and traditional semantics (e.g., aliasing/effects). Additionally, building on my prior work on compiler optimizations [3], I am now exploring compiler optimizations for quantum computing [13].

My contributions have been published at flagship programming languages (PL) and software engineering (SE) venues, including POPL [7], OOPSLA [1, 3, 4, 8, 9, 18], ICFP [10], ECOOP [17], ICSE [11], and ESEC/FSE [14]. I am also the 2022 recipient of the Maurice H. Halstead Memorial Award in Software Engineering Research.

1 TRACKING LIFETIMES, SHARING, AND EFFECTS IN HIGH-LEVEL LANGUAGES

My work in language design is centered around *reachability types*, a foundation for tracking lifetimes, sharing, and separation in higher-order languages. Ultimately, my goal is to integrate flexible, fine-grained resource control into mainstream functional programming. On top of the core reachability types (OOPSLA '21 [1]), I have further developed effect systems [1], polymorphic reachability types (POPL '24 [7]), effect-and-dependency guided optimizations (OOPSLA '23 [3]), and a type system tracking storage modes (ECOOP '22 [17]).

Tracking Contexts, Aliases, and Effects. Functions in higher-order languages serve as both data and control abstractions, making it challenging to adopt existing successful ownership-style techniques, such as those from Rust. To smoothly introduce resource control into higher-order languages, our work on reachability types and effects [1] offers a unified approach that tracks information across multiple orthogonal dimensions: *space* (e.g. heap topology), *context* (e.g. environment capturing), and *time*, (e.g. order of effectful computation).

To track *spatial* heap structure, reachability types qualify types with a set of variables reachable (i.e., aliased) from the evaluation result. Programmers can enforce *spatial* constraints by requiring disjoint qualifiers, implying that their values are separate/non-aliased. To track *contextual* information, qualifiers of function types include the free variables observed by the function from its lexical scope, consistent with the closure interpretation of functions. Tracking aliases enables precise and sound reasoning about side effects, such as the read/write of memory cells. This is already useful for ensuring safe parallelism. Now, what about Rust-style ownership transfer and move semantics? We showed that these operations can be modeled in a flow-sensitive effect system [1] *that takes both aliases and execution order (i.e., temporal) into account.* To model move semantics, our key idea is to designate *destructive effects* that "kill" any further access to a value and all its aliases. Such alias-ware flow-sensitive effect systems open a new avenue for statically checking a wide range of temporal use patterns, such as use-once capabilities, one-shot continuations, etc.

Compared to existing Rust-style approaches that impose restrictive invariants, which are then selectively relaxed through borrowing, reachability types start with a liberal foundation that tracks sharing first. Then, aliasing patterns can be selectively restricted through enforced separation or flow-sensitive effect systems.

Freshness and Polymorphism. The original reachability type system [1] works in a simply-typed setting, which is too limited for real-world polymorphic languages. The approach deploys the same notion to represent resources that are tracked but not yet bound (i.e., freshly allocated), and those that are untracked. This conflation leads to unfortunate imprecision and further prevents sound and precise extensions with reachability polymorphism.

My POPL '24 work [7] addresses this issue by introducing another dimension to reachability types. The new system offers lightweight, precise, and sound reachability polymorphism with a new designated *freshness qualifier.* This design results in a new reachability calculus providing both quantifier-free reachability polymorphism and $\mathsf{F}_{<:}$-style bounded reachability abstraction, paving the way to *integrate reachability types into realistic languages.* I have implemented the new polymorphic reachability type system in the Diamond prototype language [12].

Optimization and Memory Management for Higher-Order Programs. My works at OOPSLA '23 [3] and ECOOP '22 [17] demonstrate the practical use of rich type systems in optimization and memory management.

While graph-based intermediate representations (IRs) have been used in optimizing pure or local program blocks, their application to effectful higher-order programs is challenging due to aliasing and indirect control transfers. Augmented by reachability types, my OOPSLA '23 work [3] designs a graph-based IR that infers fine-grained effect dependencies, enabling the implementation of sound [2] and aggressive optimizations such as dead code elimination, rewriting informed by effects, and code motion across function boundaries. This IR has been implemented in the LMS compiler framework [26] and serves as the foundation for constructing program-analysis compilers in my other works [8, 9, 11, 14, 16].

As a close cousin to reachability types, my ECOOP '22 [17] work develops a modal type system to track storage lifetimes (e.g. heap or stack), enabling *efficient stack allocation of data with statically unknown sizes (e.g. closures).* Similar efforts to incorporate stack-allocated data in OCaml are currently underway, led by Jane Street. Our evaluation shows reduced heap/GC pressure and 10~25% improvements in end-to-end execution time.

Future Direction: Practical Languages and Under-Approximation Reasoning

My future agenda is to build practical programming systems around reachability types. Furthermore, I am on an intellectual quest to explore the dual concept of over-approximating reachability types.

Growing a Practical Language with Reachability Types. To study the potential of reachability types in a realistic setting, I lead the development of the Diamond language [12], implementing a type-checking algorithm for the polymorphic reachability types [7] and flow-sensitive effects. The next research question that I plan to study is the *inference algorithm for both reachability and effects in a polymorphic setting*, which is crucial to improving usability. I envision Diamond as a research vehicle to explore new ideas that complement reachability types, such as effect polymorphism, algebraic effects, and verification based on refinement types.

Moreover, I plan to explore a human-centered approach to language design. Diamond serves as an excellent playground to develop interactive language tools (e.g., visualizers, language server protocols, etc.) that assist programmers in writing correct programs by interacting with context, separation, and aliasing tracking.

Under-Approximation of Reachability. In addition to sound reasoning principles that over-approximate, recently, reasoning techniques for completeness (e.g., incorrectness logic) have gained more attention. A new avenue of my work is to study under-approximated reachability, which is the dual of my prior work that tracks over-approximated reachability. By combining these two complementary notions, types $T^{l..u}$ are augmented with lower-bound l and upper-bound u reachability. This approach would be beneficial in ensuring safety and improving expressiveness in various scenarios.

In object-oriented programming, must-reachability can be used to ensure safe object initialization when combined with a flow-sensitive effect system (complementing "kill" with its dual "gen" effects). In dependently-typed languages, tracking must-reachability can significantly improve expressiveness without sacrificing consistency (termination). The key idea is to identify “semantic subterms” with must-reachability and allow recursion over them, in contrast to merely tracking “syntactic subterms” as used in Agda or Coq.

2 METAPROGRAMMING PROGRAM ANALYZERS

Besides language design, my dissertation work addresses performance, flexibility, and specification issues in constructing program analyzers, all centered around metaprogramming approaches. Specifically, I devise compilation techniques for dynamic symbolic execution and abstract interpretation (ICSE ’23 [11], PEPM ’22 [16], ESEC/FSE ’21 [14], OOPSLA ’20 [8], OOPSLA ’19 [9]), and inter-derivation of abstract interpreters (ICFP ’18 [10]). I also develop verification techniques driven by program transformation (OOPSLA ’19 [4]).

Compiling Analysis for Free. To unlock the full potential of program analysis, analyzers must be efficient and scalable. Implementing analyzers as interpreter is not hard; however, this approach leads to significant overhead in the traversal and dispatch over static program representations (e.g. AST). To mitigate such interpretation overhead, my thesis posits that *compilation is the inevitable path towards high-performance and scalable program analysis*, as evidenced by similar implementations for concrete languages. A program-analysis compiler generates the target program that performs the desired analysis following a non-concrete semantics of the input language. Compilation eliminates the interpretation overhead and can further optimize the target program by other means.

Unfortunately, most existing program analyzers are not yet implemented in this way. Complex non-concrete semantics (richer value domains, fixed-point computation, nondeterminism, etc.) make code generation a nontrivial task. My work [8, 9, 11, 14, 16] makes this vision feasible by leveraging *Futamura’s old ideas* [21] and *modern programming abstractions*. More concretely, by writing analyzers as high-level interpreters that are amenable to manipulation and specialization [23], program-analysis compilers can be obtained for free and correct by construction. My work generalizes Futamura’s idea to symbolic execution and abstract interpretation, *offering a practical way to improve performance/efficiency without sacrificing precision/soundness.*

Controlling Symbolic Execution. Symbolic execution is an analysis technique used to systematically generate test cases or examine invariants in programs. In contrast to concrete semantics, symbolic execution explores *both* branches when the condition is symbolic, inducing nondeterministic (forking) behaviors. Therefore, compiling such semantics requires an IR to effectively execute multiple non-interference execution paths.

My ICSE ’23 work [11] (on top of my prior work [8] at OOPSLA ’20) addresses the issue of compiling multi-path symbolic execution using continuation-passing style (CPS), a compiler IR that explicitly represents the rest of the execution as first-class functions. The key insight is to use continuations to *control* the execution à la *cooperative concurrency*, allowing for *pausing, resuming, and switching the execution of multiple paths in the compiled analysis*. The execution further coordinates with a search heuristic that dispatches paths running in *parallel*, guided by runtime information (e.g., coverage). We have realized this idea with an optimizing compiler GenSym [11] for parallel symbolic execution, which significantly outperforms existing tools (e.g. 4× vs. KLEE [19]).

Achieving Flexibility via Abstractions. By developing analysis compilers using high-level programming abstractions, my work strives to achieve high performance without sacrificing flexibility and precision/soundness. One example is my OOPSLA ’19 work [9], which develops *a generic interpreter* that can be flexibly instantiated in *four modes*: (concrete + abstract) × (interpreter + compiler), enabling a higher degree of code reuse and confidence in correctness. In addition, my OOPSLA ’20 work [8] makes use of algebraic effects/handlers to specify path exploration strategies in symbolic execution. My ongoing work aims to further explore realizing complex path exploration strategies (e.g. concolic execution and state merging) using algebraic effects and continuations. In this way, concolic execution can be modeled as interleaving concrete and symbolic continuations, and state-merging as the synchronization of two concurrent paths.

Relating Small-Step and Big-Step Abstract Interpretation. Static analysis designers often need to choose a suitable form to specify the semantics. However, not every formulation leads to an executable, performance-oriented artifact, especially one that is amenable to code generation. How can we bridge the gap between diverse specifications of abstract interpretation by deriving one from another? My ICFP ’18 work [10] answers this question by constructing a functional correspondence between small-step abstract abstract machines [22] and big-step abstract definitional interpreters [20], thus filling this intellectual gap by applying mechanical transformations (i.e. refunctionalization, etc.) to meta-functions of static analyses.

Future Direction: Scaling up via Metaprogramming

I plan to investigate how metaprogramming can make program analyzers more *compositional* and *declarative*, two key aspects to engineer scalable program analysis and to support continuous software development.

Deriving Compositional Analysis for Free. A compositional analysis examines modules of a large program separately and produces "summaries" for these modules, which are then "composed" to obtain final results. However, static analyses are usually not compositional and only work for whole programs, where the major obstacle is to design suitable notions of summaries and the mechanics of their composition.

I am interested in devising mechanical *transformations to turn a whole-program analyzer into a compositional analyzer*. Specifically, I propose representing summaries as "programs" instead of data structures (inspired by an old idea of Reynolds [25]), and developing a *generative approach that compiles program units to summaries*. The composition of summaries is achieved by "linking" these generated programs. I have been exploring these ideas with prototype implementations and plan to further develop the theory and tools in the future.

Raising the Level of Abstraction in Meta-Languages. My prior work has applied high-level programming abstractions to express nondeterministic semantics. A promising direction is to further elevate the *level of meta-languages* and consider those with built-in nondeterminism (e.g. Datalog). An underappreciated yet powerful downstream application enabled by this approach is goal-directed/backward symbolic reasoning.

However, existing attempts (e.g., Flix, Formulog, etc.) encounter performance or flexibility limitations in scaling to real-world analyses. My objective is to design and build Datalog-based meta-languages that are highly *efficient* via compilation and specialization, *expressive* to support SMT and user-defined lattices, and *flexible* enough to use domain-specific evaluation heuristics. Borrowing techniques from query compilation (e.g. [27]), I aim to develop such meta-languages for versatile, efficient, and declarative symbolic reasoning.

3 APPLICATIONS IN SECURITY, SMART CONTRACTS, AND QUANTUM COMPUTING

As part of my intellectual quest as a scholar, I appreciate understanding problems, methods, and disciplines from fields other than my core areas. In particular, I apply my expertise in languages and compilers to address issues in software testing/security, smart contracts, and quantum programming.

Analyzing Low-level Code for Bug Finding & Security. The GenSym (ICSE '23 [11]) analyzer presents a radical reconstruction of parallel symbolic execution engines atop code generation and continuations. Benchmarked on real-world CoreUtils programs in LLVM IR, GenSym's single-thread execution is already 4× faster than state-of-the-art symbolic interpreter KLEE [19] thanks to compilation. With 12 threads, GenSym is 9.4× faster compared to its own sequential execution, thanks to CPS-based parallelism. The ongoing development of a concolic-execution compiler for WebAssembly (based on preliminary work [16]) would further leverage continuations to enable efficient snapshot reuse in concolic execution.

By repurposing program optimizations to enhance *clarity*, my OOPSLA '19 work [4] develops verification techniques for C-like languages. We use optimization techniques to *recover heap-and-time structures represented as collective operations*. Then low-level programs can be mechanically verified more easily with these higher-level collective operations. We have built a formal model in Coq to establish soundness and a tool SIGMA [4], which outperforms state-of-the-art tools (CPAChecker and SeaHorn) on SV-COMP benchmarks.

Further lowering the level of analyzed IRs, I have worked on binary dependence analysis (OOPSLA '19, *distinguished paper award* [18]) based on abstract interpretation, which significantly outperforms existing value-set analyses. I have also worked on (1) verified lifting techniques [5] from binary to LLVM, providing a higher degree of assurance for decompilation correctness, and (2) an abstract interpreter for JVM bytecode to detect vulnerabilities related to space/time complexity attacks.

Ongoing Work: Behavioral/Temporal Contracts for Smart Contracts. My recent work [15] (submitted to PLDI) argues that *behavioral contracts* [24] *and smart contracts should complement each other*. We develop ConSol that allows specifying and enforcing rich pre- and post-conditions in the Solidity language. Inspired by contracts for higher-order functions, ConSol enables programmers to modularly specify and enforce conditions for latent address calls while maintaining gas efficiency, without worrying about how first-class address values propagate. The effectiveness of ConSol has been examined as a defense against known real-world attacks, which have resulted in a total loss of $8.1M.

Additionally, many critical conditions in smart contracts are *temporal*, e.g., preventing the reentrancy of a critical function. I plan to develop expressive and developer-friendly temporal specification systems for smart contracts. These systems would enable numerous downstream tasks, including but not limited to runtime validation, static verification (e.g., as flow-sensitive effect systems [1]), property-based testing, etc.

Future Direction: Languages and Compilers for Quantum Computing

The potential of quantum computing to revolutionize computation is increasingly evident. To unlock such potential, I am interested in developing quantum languages and compilers, which are, however, still in their infancy.

First, I plan to design expressive quantum languages on top of my prior work on reachability types/effects [1, 7]. Specifically, *reachability can track entanglement between qubits*, and *destructive effects can track measurements on entangled (i.e. "aliased") qubits*. By statically tracking entanglement, separation, and measurement, well-typed quantum programs are ensured not violating physical laws. Compared to existing proposals (e.g. those based on linear types), reachability types would permit more flexibility and expressiveness.

Apart from language design, I am interested in developing *modular and reusable compilation and optimization techniques* for quantum programs using metaprogramming [6] and graph-based compiler IRs [3]. Collaborating with Kirshanthan Sundararajah at Virginia Tech, we are working on optimizing classical simulation of quantum circuit programs via compilation (a preliminary work submitted to PLanQC [13]). The key insight is to notice that the matrix computation for simulation is neither entirely dense nor randomly sparse, but *structurally sparse* with *repetitiveness patterns*. We are developing compiler optimizations to generate performant code that exploits the structural sparsity and repetitiveness (e.g. by representing matrices as trees to enable sharing). Additionally, I would like to explore how reasoning techniques for quantum programs such as ZX-calculus can be integrated with my prior work on graph IRs [3], which would enable optimizations that exploit both high-level quantum structures and lower-level classical computation.

REFERENCES

[1] Yuyan Bao, **Guannan Wei**, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021).

[2] Yuyan Bao, **Guannan Wei**, Oliver Bračevac, and Tiark Rompf. 2023. Modeling Reachability Types with Logical Relations. *CoRR* abs/2309.05885 (2023).

[3] Oliver Bračevac, **Guannan Wei**, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages – Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA (2023).

[4] Gregory Essertel, **Guannan Wei**, and Tiark Rompf. 2019. Precise Reasoning with Structured Heaps and Collective Operations. *Proc. ACM Program. Lang.* 3 , OOPSLA (2019).

[5] Joe Hendrix, **Guannan Wei**, and Simon Winwood. 2019. Towards Verified Binary Raising. *Workshop on Instruction Set Architecture Specification (SpISA), co-located with ITP* (2019).

[6] **Guannan Wei**. 2023. *Metaprogramming Program Analyzers.* Ph.D. Dissertation. Purdue University.

[7] **Guannan Wei**, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *To Appear at POPL* (2024).

[8] **Guannan Wei**, Oliver Bračevac, Shangyin Tan, and Tiark Rompf. 2020. Compiling Symbolic Execution with Staging and Algebraic Effects. *Proc. ACM Program. Lang.* 4 , OOPSLA (2020).

[9] **Guannan Wei**, Yuxuan Chen, and Tiark Rompf. 2019. Staged Abstract Interpreter: Fast and Modular Whole-Program Analysis via Meta-Programming. *Proc. ACM Program. Lang.* 3 , OOPSLA (2019).

[10] **Guannan Wei**, James Decker, and Tiark Rompf. 2018. Refunctionalization of Abstract Abstract Machines: Bridging the Gap Between Abstract Abstract Machines and Abstract Definitional Interpreters (Functional Pearl). *Proc. ACM Program. Lang.* 2 , ICFP (2018).

[11] **Guannan Wei**, Songlin Jia, Ruiqi Gao, Haotian Deng, Shangyin Tan, Oliver Bračevac, and Tiark Rompf. 2023. Compiling Parallel Symbolic Execution with Continuations. In *ICSE 2023*. IEEE, 1316–1328.

[12] **Guannan Wei**, Songlin Jia, and Tiark Rompf. 2023. The Diamond Language. https://continuation.passing.style/Diamond.

[13] **Guannan Wei** and Kirshanthan Sundararajah. 2024. Towards Compiling Quantum Circuit Simulation via Decomposing Sparse Tensors (Extended Abstract). *Submitted to the Fourth International Workshop on Programming Languages for Quantum Computing* (2024).

[14] **Guannan Wei**, Shangyin Tan, Oliver Bračevac, and Tiark Rompf. 2021. LLSC: A Parallel Symbolic Execution Compiler for LLVM IR (Demonstration). In *ESEC/SIGSOFT FSE 2021*. ACM, 1495–1499.

[15] **Guannan Wei**, Danning Xie, Wuqi Zhang, Yongwei Yuan, and Zhuo Zhang. 2024. Consolidating Smart Contracts with Behavioral Contracts. *Submitted to PLDI* (2024).

[16] Shangyin Tan, **Guannan Wei**, and Tiark Rompf. 2022. Towards Partially Evaluating Symbolic Execution for All (Short Paper). *Workshop on Partial Evaluation and Program Manipulation (PEPM), co-located with POPL* (2022).

[17] Anxhelo Xhebraj, Oliver Bračevac, **Guannan Wei**, and Tiark Rompf. 2022. What If We Don't Pop the Stack? The Return of Second-Class Values. *The 36th European Conference on Object-Oriented Programming (ECOOP)* (2022).

[18] Zhuo Zhang, Wei You, Guanhong Tao, **Guannan Wei**, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: Practical Dependence Analysis for Binary Executables by Unbiased Whole-Program Path Sampling and Per-Path Abstract Interpretation. *Proc. ACM Program. Lang.* 3 , OOPSLA (2019).

OTHER REFERENCES

[19] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. USENIX Association, 209–224.

[20] David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP (2017), 12:1–12:25.

[21] Yoshihiko Futamura. 1971. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Systems, Computers, Controls* 25 (1971), 45–50.

[22] David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *ICFP*. ACM, 51–62.

[23] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice Hall.

[24] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51.

[25] John C. Reynolds. 1978. *User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction.* Springer New York, New York, NY, 309–317.

[26] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *GPCE*. ACM, 127–136.

[27] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD*. ACM, 307–322.