Research Statement

Oliver Bračevac

Fall 2022

I am broadly interested in **programming languages (PL)**, perhaps the most important tool for creating software. As a PL researcher, my goal is to make lofty ideas crisp and firmly grounded, and unearth the core concepts and abstractions of complex problem domains. **The overarching theme of my research is devising good building blocks for creating safe, efficient, extensible, and modular systems based on principled PL approaches**. I strive for foundational, mathematically solid tools and techniques that help solve practical problems and yield safe and correct systems with tangible performance benefits. **I want develop new classes of high-level, typed functional programming languages that permit safe reasoning about low-level aspects of computations in a developer-friendly way**. In this spirit, I have been working on the following projects:

- **Reachability Types [3]:** A novel type system for tracking aliasing, ownership, and low-level memory properties in impure functional languages (e.g., Scala, OCaml) based on separation logic. Compared to Rust's type system, it demonstrably supports higher-order functions more seamlessly. It also supports state-of-the-art lightweight polymorphism and programming with effects as capabilities.
- **Seamless Stack Allocation [2]:** Many programming languages (e.g., Swift, C♯, OCaml) are currently advocating for forms of stack-allocated values. My research offers the first comprehensive and provably sound solution based on a type system tracking modes of storage. Compared to other proposals, it seamlessly supports stack-allocated data types of statically unknown size, including function closures.
- **Easy and Efficient Symbolic Execution for Everyone [5, 4, 1]:** A simple and high-level approach for constructing high-performance symbolic execution compilers, based on generative programming and functional programming techniques.
- **Algebraic Effects & Handlers for Versatile Correlations [6, 7]:** I designed a modular and extensible framework for programming systems that correlate/join unbounded information flows. It is centered around effect handlers [22] being the fundamental building blocks for programming correlations.
- **Other Works:** I have contributed to the theory and practice of incremental type checking [11, 9], foundations of serverless computing [10], and mechanized possibilistic information flow security [8].

Reachability Types: Expressive Ownership-Style Reasoning for Higher-Order Programs with (Co)Effects

Ownership-style type systems have seen increasing mainstream adoption, spearheaded by Rust. While plenty of prior academic works in this space exists, they often do not allow full control over the finer aspects of first-class, and higher-order functions/closures, the core primitives of functional programming as in languages like Scala, OCaml, Racket, etc. Functions are often subject to severe restrictions by such type systems, and (much to our surprise!) little previous research exists towards ownership across *higher-order functions*. Reachability types [3] enable safe and idiomatic reasoning about memory properties of functional programs with effects.

The design of reachability types is inspired by Reynold's separation logic, which also serves a key role in establishing the formal foundations for Rust's type system (cf. RustBelt [17]). However, we expose and explore separation all the way to the user-facing type level. The type system tracks aliased variables in the surface types, and permits users to reason about their separation/overlap, which has many useful applications, e.g., programming with capabilities having context-sensitive lifetime and sharing properties.

The reachability type system has the property that it is implicitly polymorphic, because it is a dependently-typed system. The value of implicit polymorphism has been recently recognized for effect systems (e.g. Scala Capture Types [21]), and our work is no exception in this regard.

I am part of the core team behind reachability types, and have been leading the mechanization effort of the system's metatheory in Coq, including work under submission on extending the type theory [13] and novel applications of the type system in optimizing compilers for impure functional programming languages [12].

Seamless Stack Allocation

The call stack was first proposed by Dijkstra for the ALGOL language in the 1960s, and it was a breakthrough that enabled programming with recursive functions. Since then, "the stack" with its simple and yet very rigid automatic memory allocation/reclamation has been the backbone of most programming language implementations. That the stack immediately "pops" when a function returns is taught in every CS curriculum and unquestioningly accepted.

However, "what if we don't pop the stack"? This is a simple question with profound consequences [2]: if we let the stack frame live *just a little longer* than usual, we gain the ability to return data structures of variable size entirely on the stack! We can run a much larger class of computations purely on the stack, requiring no heap, no garbage collection (GC), and no manual memory management whatsoever. This is beneficial for a wide range of applications, especially those which process short-lived ephemeral data, e.g., reactive and dataflow computations, differential programming, HTTP servers, etc.

Several languages (e.g., C♯, Swift, and OCaml) have recently proposed adding support for stack-allocated values. Those proposals, while promising, have limitations, e.g., the size of stack-allocated data has to be fully known at compile time, which limits expressiveness of on-stack computations and inhibits uses of higher-order functions. In contrast, our approach is much more *seamless* and supports returning on-stack data of statically unknown size, including anonymous function closures.

To ensure delayed stack allocation is safe, we also contribute an expressive type system tracking the storage mode of expressions (i.e., heap and stack) and supporting forms of storage-mode polymorphic code. We proved type-and-memory safety of our type system in Coq, and have implemented a prototype compiler on top of Scala native. Our benchmarks show that the approach can reduce garbage collection overhead by up to 54% and can improve wall-clock time by up to 22%.

Easy and Efficient Symbolic Execution for Everyone

Complementing the purely static typing approaches of reachability types and storage modes, I have also been working on dynamic analyses, making symbolic execution easy, scalable, and performant using PL principles and code generation techniques [5, 4].

Symbolic execution (SE) is a popular software-testing technique from the 1970s, e.g., used for bug finding, security, verification, and program synthesis. The underlying idea is to simultaneously explore multiple execution paths in a given program, with some inputs being left symbolic rather than concrete, and generating constraints checked by SMT solvers. However, building an SE engine is often considered "the most difficult aspect of creating solver-aided tools". Our work drastically reduces this hardship by a streamlined process which is both easy to grasp and performant. It reconciles the strengths of prevailing construction approaches for SE without their downsides, i.e., those based on interpreters (high level, but slow) and instrumentation of programs (performant, but ad hoc, language-specific, and hard to generalize).

Our framework relies heavily on concepts from functional programming, i.e., the interpreters return a monadic representation of symbolic programs in terms of algebraic effects and handlers [22]. Effect handlers grant extensibility (new object-language effects can be added), and customizability (effects can be denoted differently on the fly). For instance, SMT solver interactions can be abstracted over with a dedicated algebraic effect, and concrete handlers implement the communication between the solver (e.g., Z3) and the SE engine. Exploring different execution paths is a nondeterminism effect, and concrete handlers

implement user-defined search strategies and heuristics. To eliminate all abstraction and interpretation overhead, we exploit an old discovery by Futamura [16] showing how a compiler can be mechanically derived from an interpreter using partial evaluation/multi-stage programming.

We demonstrate our framework by symbolic execution of LLVM IR programs. Remarkably, the first version [5] with a naive code generation backend already outperforms the interpreter-based KLEE [14] by up to $2x$. Since then, we have further improved performance by generating code for asynchronous path exploration [4], followed by compiling truly parallel explorations [1]. We now achieve average speedups of $4x$ over KLEE on real-world code bases (GNU Coreutils).

Algebraic Effects & Handlers for Versatile Correlations

As part of my dissertation research with Mira Mezini at TU Darmstadt, I have worked on PL support for correlating notions of information flows [6, 7], akin to database joins, but over diverse data sources, which may be unbounded, continuously and frequently change over time, and which are often concurrent, asynchronous, and distributed. An example of such a query is "the current GPS position of all delivery vehicles observed within a 20 mile radius of NYC over the last hour". Correlations, are broadly useful, e.g., in data science, machine learning, real-time monitoring, modern user interfaces, truth maintenance.

I proposed a new perspective on correlations, called the "computational interpretation (CI)" [7]. It states that correlations over n data sources are cartesian product computations, but some "force" is influencing how these computations unravel. The "force" here are computational effects manipulating and constraining the control flow of the computation, so that it does not materialize all the pairings *a priori*.

My CI bridges purely relational algebra perspectives on correlation and impure computational perspectives, with effects being the mediator. Concerning programming abstractions for realizing the CI, I found algebraic effect handlers [22] to be fitting, because they constitute modular and extensible snippets of denotational semantics, and they are first-class representations of control flow, which can thus be manipulated (the "force" acting on the computation).

Correlations can be uniformly programmed by effect handlers that interact with the generic cartesian product, in a manner akin to coroutines. The modularity and extensibility for algebraic effects carries over into the correlation setting, so that it is possible for programmers to define a custom vocabulary of new interactions with the computation. Correlation features from different systems domains can be cross-composed mix-and-match style, thus leading to a truly "à la carte"[1] correlation system.

My research at TU Darmstadt primarily focused on expressivity and high-level abstractions for correlations. At Purdue, I have been strengthening my expertise in compilation, code generation, and type systems for memory properties, in order to make correlations performant in future research.

Near-Term Vision

Reachability Types for Driving Compiler Optimizations Graph-based intermediate representations (IRs) are widely used for powerful compiler optimizations, either interprocedurally in pure functional languages, or intraprocedurally in imperative languages. Yet so far, no suitable graph IR exists for aggressive global optimizations in languages with both effects and higher-order functions (like Scala or OCaml): aliasing and indirect control transfers make it difficult to maintain sufficiently granular dependency information for optimizations to be effective. To close this long-standing gap, I proposed a novel typed graph IR combining reachability types [3] with an expressive effect system to compute precise and granular effect dependencies at an affordable cost while supporting local reasoning and separate compilation.

This IR is part of the next version of the Scala lightweight modular staging (LMS) compiler framework [23]. A first part of this work is already under submission [12], with initial results showing dramatic speedups (up to 21x) in functional machine learning DSLs.

[1] In homage to Swierstra's "Data types à la carte" [24].

Easy and Efficient Symbolic Execution for Everyone *and Everything* A natural follow-up extension to our work on generating symbolic compilers [5, 4, 1], which so far has focused on "all-path symbolic execution", is demonstrating the applicability to other kinds of symbolic execution, e.g., concolic execution, fuzzing, diverse verification tasks (like Saw [15]), and program synthesis (like Rosette [25]). I anticipate that our generative approach (1) will effortlessly set new performance records for these applications, (2) will set the standard on how to design, construct, and teach symbolic execution tools, (3) can be further automated, leading to new high-level tools and meta-DSLs, and (4) will lead to new classes of domain-specific optimizations that broadly apply to any kind of symbolic execution flavor and object language. To further increase performance, I plan to investigate generating the SMT solver's code itself and embedding it into the symbolic execution engine, and make use of the above graph IR based on reachability types [12].

Long-Term Vision

Towards Functional Systems Programming with Reachability Types I want to extend high-level typed impure functional languages (e.g., Scala, Swift, OCaml) with facilities that permit safe reasoning about low-level aspects of computations in a way that is unobtrusive and retains the high level of abstraction. One use case is rich support for capability-oriented systems programming. Our reachability type system [3] is well suited for this purpose, due to its support for higher-order functions front and center, its lightweight tracking of aliasing, and its notion of separation. An important question is how our foundational reachability system scales to a full language. A follow-up adding type polymorphism (generics) and data types is already under submission [13], and more extensions (e.g., higher-kinded types) are planned. These extensions will further require metatheoretical reasoning tools such as logical relations, a denotational semantics, and equational theory for the type system. In this context, I also intend to scale the graph IR work [12] to a full optimizing compiler for impure higher-order languages that rivals state-of-the-art compilers like MLton [20].

New Classes of Safe, Fast, and Modular Correlation Systems Moving forward, I want to fundamentally change the landscape for correlation systems. I envision new classes of systems that are simultaneously (1) modular, extensible, and customizable, and (2) safe, fast, and scalable. My PhD work [6, 7] has already laid the groundwork for (1), and I will tackle (2) with symbolic execution [5, 4, 1] and compiler optimizations based on reachability types and staging [3, 13, 12].

An important next step is eliminating sources of abstraction overhead, in particular effect handlers. That will lead to new fusion and deforestation techniques for effect handlers as well as staging-based optimizations thereof using Scala LMS. Sophisticated compiler optimizations for effect handlers (a fairly recent development) are still largely underexplored by language implementations (e.g., multicore OCaml). Thus, the outcomes of this research will also benefit the broader algebraic effects community, and contribute to the theory and practice of continuations and coroutines, since handlers are close cousins of these concepts. Using multi-stage programming is very likely to be effective, because correlations in my framework can be clearly separated into stages of execution, and related works on stream processing (e.g., [19, 18]) have greatly benefited from staging.

Certain correlations over asynchronous data source features negative constraints, e.g., an observation should *not* occur within a certain amount of time of another. Such constraints involving the passage of time and non-occurrences can be formulated in a way that is unsatisfiable, and make a continuously running correlation stuck or unproductive. I intend to generalize the work on symbolic execution [5, 4, 1] to notions of symbolic streams and dataflow for ensuring liveness of correlations even under negative constraints. The key idea is to symbolically execute the constraints in declarative correlation pattern expressions and check with SMT solvers whether there is a potential scenario in which such liveness properties can be violated. Another application is using symbolic execution on declarative correlation patterns to generate the underlying continuations and coordination logic of n-way asynchronous joins, by "simulating" the correlation with symbolic observations.

Publications

[1] Guannan Wei, Songlin Jia, Ruiqi Gao, Haotian Deng, Shangyin Tan, **Oliver Bračevac**, and Tiark Rompf. "Compiling Parallel Symbolic Execution with Continuations". In: *International Conference on Software Engineering (**ICSE**)*. **to appear**. 2023.

[2] Anxhelo Xhebraj, **Oliver Bračevac**, Guannan Wei, and Tiark Rompf. "What If We Don't Pop the Stack? The Return of 2nd-Class Values". In: ***ECOOP***. Vol. 222. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 15:1–15:29. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16243/.

[3] Yuyan Bao, Guannan Wei, **Oliver Bračevac**, Yuxuan Jiang, Qiyang He, and Tiark Rompf. "Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs". In: *Proc. ACM Program. Lang.* 5.**OOPSLA** (2021), pp. 1–32. URL: https://doi.org/10.1145/3485516.

[4] Guannan Wei, Shangyin Tan, **Oliver Bračevac**, and Tiark Rompf. "LLSC: A Parallel Symbolic Execution Compiler for LLVM IR". In: *ESEC/SIGSOFT **FSE***. ACM, 2021, pp. 1495–1499. URL: https://doi.org/10.1145/3468264.3473108.

[5] Guannan Wei, **Oliver Bračevac**, Shangyin Tan, and Tiark Rompf. "Compiling Symbolic Execution with Staging and Algebraic Effects". In: *Proc. ACM Program. Lang.* 4.**OOPSLA** (2020), 164:1–164:33. URL: https://doi.org/10.1145/3428232.

[6] **Oliver Bračevac**. "Event Correlation with Algebraic Effects - Theory, Design and Implementation". PhD thesis. Darmstadt, Germany: TU Darmstadt, Nov. 2019. URL: http://tuprints.ulb.tu-darmstadt.de/9258/.

[7] **Oliver Bračevac**, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. "Versatile Event Correlation with Algebraic Effects". In: *Proc. ACM Program. Lang.* 2.**ICFP** (2018), 67:1–67:31. URL: https://doi.org/10.1145/3236762.

[8] **Oliver Bračevac**, Richard Gay, Sylvia Grewe, Heiko Mantel, Henning Sudbrock, and Markus Tasch. "An Isabelle/HOL Formalization of the Modular Assembly Kit for Security Properties". In: ***Archive of Formal Proofs*** (2018). ISSN: 2150-914x. URL: https://isa-afp.org/entries/Modular_Assembly_Kit_Security.html.

[9] Edlira Kuci, Sebastian Erdweg, **Oliver Bračevac**, Andi Bejleri, and Mira Mezini. "A Co-contextual Type Checker for Featherweight Java". In: ***ECOOP***. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 18:1–18:26. URL: http://dx.doi.org/10.4230/LIPIcs.ECOOP.2017.18.

[10] **Oliver Bračevac**, Sebastian Erdweg, Guido Salvaneschi, and Mira Mezini. "CPL: A Core Language for Cloud Computing". In: *Proceedings of Conference on Modularity (**MODULARITY**)*. 2016. URL: https://doi.org/10.1145/2889443.2889452.

[11] Sebastian Erdweg, **Oliver Bračevac**, Edlira Kuci, Matthias Krebs, and Mira Mezini. "A Co-Contextual Formulation of Type Rules and its Application to Incremental Type Checking". In: ***OOPSLA***. ACM, 2015, pp. 880–897. URL: https://doi.org/10.1145/2814270.2814277.

Drafts Under Submission

[12] **Oliver Bračevac**, Guannan Wei, Yuxuan Jiang, Supun Abeyshinghe, Songlin Jia, Yuyan Bao, and Tiark Rompf. "From Effects and Reachability to Granular Dependencies: A new Graph IR for Impure Higher-Order Languages". 2022.

[13] Guannan Wei[†], **Oliver Bračevac**[†], Siyuan He, Yuyan Bao, and Tiark Rompf. "Tracking Aliasing and Separation in Polymorphic Higher-Order Programs". [†]*equal contributions*. 2022.

Other References

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

[15] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. "Constructing Semantic Models of Programs with the Software Analysis Workbench". In: *VSTTE*. Vol. 9971. Lecture Notes in Computer Science. 2016, pp. 56–72.

[16] Yoshihiko Futamura. "Partial evaluation of computation process-an approach to a compiler-compiler". In: *Systems, Computers, Controls* 25 (1971), pp. 45–50.

[17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: securing the foundations of the rust programming language". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 66:1–66:34.

[18] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. "Stream fusion, to completeness". In: *POPL*. ACM, 2017, pp. 285–299.

[19] Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. "i3QL: language-integrated live data views". In: *OOPSLA*. ACM, 2014, pp. 417–432.

[20] MLton. *The MLton Compiler and Runtime System*. 2012. URL: http://www.mlton.org.

[21] Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondrej Lhoták. "Safer exceptions for Scala". In: *SCALA/SPLASH*. ACM, 2021, pp. 1–11.

[22] Gordon D. Plotkin and Matija Pretnar. "Handlers of Algebraic Effects". In: *ESOP*. Vol. 5502. Lecture Notes in Computer Science. Springer, 2009, pp. 80–94.

[23] Tiark Rompf and Martin Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs". In: *GPCE*. ACM, 2010, pp. 127–136.

[24] Wouter Swierstra. "Data types à la carte". In: *J. Funct. Program.* 18.4 (2008), pp. 423–436.

[25] Emina Torlak and Rastislav Bodík. "Growing solver-aided languages with rosette". In: *Onward!* ACM, 2013, pp. 135–152.