Research Statement

Ralf Jung

I am developing *formal foundations and tools* that can establish *machine-checked guarantees* for *real-world software systems*. To achieve this, my work spans all the way from foundational and deeply theoretical to applied, from proving theorems to developing tools used by other researchers and software developers.

One key challenge in modern software is that fully utilizing today's processors forces software to be highly *concurrent*. Concurrent software requires the programmer to think about all the many ways in which parallel computations could interact, making it hard to get right. At the same time, testing is often insufficient for finding concurrency bugs, since bugs might arise only in specific circumstances that happen to arise in production but do not occur during testing. I am working on *formal methods* that are able to exclude bugs in concurrent software.

Another important challenge in modern software is to reconcile safety with performance. Many languages use type systems to guarantee that entire classes of bugs cannot occur. However, following the rules of the type system regularly imposes run-time costs, e.g. through extra copies or bounds checks. In performance-critical code, programmers can avoid that cost by using *unsafe features*, resulting in code that achieves safety for sophisticated application-specific reasons that are difficult for the compiler to understand. Most type-safe languages have such unsafe mechanisms, but these language features are largely ignored by type system formalizations. *Rust* is an example of a language that uses unsafe features to great effect; I am using formal methods to capture why this works and how we can formally reason about this situation.

Foundations for reasoning about modern software. Most modern software is both imperative and highly concurrent, often leading to non-trivial interactions between multiple threads running in parallel while sharing memory. *Concurrent Separation Logic* (recently awarded with the Gödel prize) makes the tantalizing promise that despite all these interactions, we should be able to verify such concurrent imperative programs *locally*, one function at a time, without explicitly enumerating all the exponentially many possible executions of such a program.

Over the past decade, many variants of concurrent separation logic were developed to prove correctness of increasingly intricate concurrent algorithms and data structures. However, these logics typically came with some *built-in* notion of how to reason about the interaction of multiple threads acting on shared state, e.g., they fix a particular style of state-transition systems. This has led to the issue that a new logic had to be developed each time a new interaction pattern had to be reasoned about.

My work on **Iris** [14, 12, 16, 13] aims to remedy this situation. The core design principle of Iris is to focus on a small, expressive core logic that enables its user to *derive* new reasoning principles at a high level of abstraction. In particular, the reasoning principles of many previous logics can be encoded inside Iris, unifying and consolidating prior work. On top of that, Iris permits these proofs to be carried out in a machine-checked manner through its implementation in the Coq Proof Assistant: it is the first separation logic to provide the usual user experience of interactive proof assistants [17, 15].

Iris thus makes the challenging task of formally reasoning about concurrent software a lot more approachable. This has led to a success story: Iris forms the foundation of more than 40 verification projects[1] establishing a wide range of formal guarantees about concurrent systems, ranging from functional correctness to type safety to security properties such as non-interference. To name two examples, the Perennial project at MIT has built an Iris-based toolchain for verifying Go code [2] (I have since joined this team to continue this exciting line of research [3]), and BedRock Systems is using Iris for commercial verification of a hypervisor written in C++.

I have led the development of Iris from the start, combining the usual academic approach to collaboration (the Iris papers cited above altogether have 11 authors) with my experience in how open-source projects are managed: Iris employs standard open-source development practices such as public issue tracking, continuous integration, and a chat where users and developers meet. The Iris community helps maintain the Coq library and also provides support to new users. This open-source spirit is a key ingredient to the continued success of Iris.

Foundations for Rust. *Rust* is a promising newcomer in the field of programming languages that is recently gaining notable traction both in academia and industry: initiated at Mozilla and deployed in the Firefox browser, Rust is now being used and developed by most of the "big tech" companies (e.g., Amazon, Google, Microsoft,

[1] See https://iris-project.org for a full list of publications using Iris.

and Facebook). Its key selling point is combining *type safety* with a programming model that enables optimizing for *performance* like one could in languages such as C or C++. Rust ensures memory safety and data-race freedom, ruling out entire classes of bugs that plague C and C++ programs and regularly cause security vulnerabilities.

However, because a type checker will never be able to accept all correct programs, Rust also features `unsafe` blocks that unlock non-type-safe operations. To avoid losing all the benefits of memory safety, these unsafe blocks are generally *safely encapsulated* behind an API surface, i.e., the author of the library promises that well-typed code cannot cause memory errors or data races by interacting with this library. This leads to a compositional ecosystem of high-performance libraries that can be safely used to build reliable applications.

That is the claim, anyway. I have built **RustBelt** [10, 11] to formally verify these claims. RustBelt is the first formal proof of correctness of the Rust type system that accounts for `unsafe` code. As part of this, we verified the soundness of some of the most important data structures in the standard library, meaning we showed that the `unsafe` code they use internally is safely encapsulated by their API: *any* well-typed code using these libraries is still memory safe and data-race free. What makes these data structures particularly interesting and challenging to verify is that they "violate" a key property of the Rust type system—that all data is shared or mutable, never both—to provide safe access to shared mutable state.

Moreover, the approach we chose is compositional: even though the individual abstractions were verified independently of each other, the safety result scales to safe programs that combine these abstractions in arbitrary ways. The RustBelt proof is carried out using Iris and mechanized in the Coq Proof Assistant, crucially relying on the versatility of Iris and its ability to define and verify new logical reasoning principles at a high level of abstraction.

Our work has led to the discovery of a bug in `Mutex` [6] (implementing mutual exclusion and used for lock-based concurrent programming), and follow-on work by collaborators that incorporates weak memory into RustBelt [4] has found another bug in Rust's atomically reference-counted pointer (`Arc`) [8]. Based on this formal model, I have closely interacted with the Rust language team during the development of a new feature for *pinning* [5, 7], a cornerstone of Rust's growing support for asynchronous programming.

In follow-on work, we have established safety of `GhostCell` [20], a library that enables Rust programmers to split permissions from data to enable new safe ways of using shared mutable state in Rust. This safety proof was particularly tricky since GhostCell employs the type system in non-conventional ways by relying on "branding": type-level identifiers that are used as purely logical indicators to track which permission belongs to which data.

While this line of research is specific to Rust, the general approach is not: unsafe escape hatches similar to `unsafe` blocks exist in most programming languages (e.g., `Obj.magic` in OCaml, `unsafePerformIO` in Haskell), but they are usually neglected in formal studies of these languages. The same methodology can also be applied to reason about the interaction of a type-safe language with a non-type-safe language, which is common in real systems (e.g., Java code calling a C library). In fact, RustBelt has already inspired people to apply a similar approach for verifying OCaml code [19].

Specification and Tooling for Rust. RustBelt established that the Rust type system is useful to ensure that programs run correctly. However, there is another benefit the type system could bring to the table: the compiler should be able to exploit type information for *performance*. In particular, Rust's reference types provide very strong alias information, which otherwise would be hard or impossible for the compiler to obtain. If Rust compilers can exploit that alias information, that could make Rust not just a safer language than C or C++, but also one that can be optimized better.

Unfortunately, this story becomes tricky once we consider unsafe code. Unsafe Rust is to some extent allowed to circumvent the language's aliasing restrictions. How can we ensure that optimizations remain correct even in the presence of unsafe code? My work on **Stacked Borrows** [9] provides a possible answer to this question. Stacked Borrows specifies a precise set of rules that unsafe code has to follow, and we formally prove in the Coq Proof Assistant that under this assumption some powerful optimizations are correct.

However, enabling optimizations is only one side of the coin. We must also take care that Stacked Borrows is *realistic*, i.e., that it still permits programmers to do what they need to do with unsafe Rust. To this end, I have implemented the Stacked Borrows rules in **Miri**, an interpreter for Rust, which means that Miri is able to tell if real Rust code is compatible with the requirements of Stacked Borrows. This enabled me to define Stacked Borrows in a way that the Rust standard library test suite is fully compliant. This work was done in close collaboration with the Rust language team. The process of officially standardizing an aliasing model is still ongoing, but in the mean time, Stacked Borrows has become the de-facto baseline aliasing model in Rust.

In fact, I have turned Miri into a tool that can check Rust code for compliance not only with Stacked Borrows but also with many other requirements that Rust imposes on unsafe code (e.g., not performing out-of-bounds memory accesses). It has become a standard tool in the Rust community: Miri is available as part of the official distribution of the nightly Rust compiler, developers use Miri in their continuous integration to gain confidence in the correctness of their unsafe code [1], and Miri has also been integrated into the Rust Playground,[2] enabling Rust programmers to test small snippets of code directly in the browser.

Future research

In my past research, I developed a suite of influential and groundbreaking tools for reasoning modularly about the correctness of complex concurrent programs (Iris) and the safety of Rust, a popular and actively developed systems programming language (RustBelt, Miri, Stacked Borrows). My plans for future work build on that foundation. Iris has already been demonstrated to be applicable to a wide range of problem domains, so I aim to use Iris as a stepping stone for branching off into new research directions. On the Rust side, my existing expertise and my connections within the community provide a unique opportunity to identify and tackle challenging research problems that will impact the future development of the language.

Distributed systems. Distributed systems (*i.e.,* services that are distributed across many machines in a network) form the backbone of our internet infrastructure. They are also notoriously hard to get right: networks are unreliable, so distributed systems have to contend with network messages being lost, reordered, or duplicated. Testing all possible cases is not usually feasible, making verification a core tool for ensuring reliability of distributed systems. This becomes even more true when one considers distributed systems that also exploit per-machine concurrency, as is often the case today.

In ongoing work, I am applying Iris to the verification of distributed systems. The goal is to verify distributed systems *locally*, one component at a time. This thinking in terms of reusable and replaceable components reflects the way distributed systems are typically built, but is not yet applied in existing high-profile verification efforts (like Verdi and IronFleet) which have focused on establishing strong correctness properties of *complete systems*. By using a powerful concurrent separation logic, we can state and prove specifications of distributed systems components, and we can compose specifications of these components to verify a complete distributed system.

Another critical aspect of realistic distributed systems is *fault tolerance*: recovering gracefully when nodes of the network crash or become inaccessible due to network outages. However, the state of the art in distributed systems verification largely ignores the difficulties that arise when a node recovers and rejoins the system based on whatever durable state it still has on disk. In the MIT work on Perennial, Iris has already been shown expressive enough to reason about crash recovery in concurrent programs running on a single machine; in future work, I will investigate how to scale this approach up to account for crash recovery in a distributed system. The key challenge is that unlike in Perennial, distributed systems have to consider *partial failures*, where some machines crash and reboot while others keep running without even knowing that a failure occurred.

Verifying large-scale realistic systems will also require innovation within Iris itself. While Iris comes with a carefully engineered "proof mode" that provides excellent support for doing machine-checked interactive separation-logic proofs in Coq, Iris proofs nonetheless require a great deal of manual effort. I believe that improving automation for Iris proofs will be essential for deploying it in the verification of large-scale systems, and that there are clear opportunities for improvement. This will be challenging because most existing approaches to automation in proof assistants are geared towards standard (non-separation) logics, and thus cannot be applied to automating Iris proofs. Finally, I plan to investigate whether we can soundly combine Iris with existing fully automated solvers for separation logic like GRASShopper.

Towards a Rust Standard. Any attempt to formally reason about programs requires a solid specification of the language they are written in. However, such a specification does not yet exist for Rust (so RustBelt had to make do with an approximation). Developing a Rust specification is thus an important open problem—notwithstanding Stacked Borrows, which only makes a first stab at specifying one particular aspect of the language semantics.

At the same time, it is a major challenge: like C and C++, Rust combines a low-level machine model with aggressive optimizations that make assumptions the programmer needs to be aware of: when writing

[2]The playground can be found at https://play.rust-lang.org/.

`unsafe` Rust, the programmer has to guarantee that the program does not have *undefined behavior*. However, determining if a given program has undefined behavior is very tricky, and even the hundreds of pages of specification that C and C++ come with are often not enough to answer that question. This affects both programmers that accidentally write code with undefined behavior, and compiler writers that accidentally make inconsistent assumptions about undefined behavior in different parts of the compiler, leading to incorrect compilation results.

I am planning to use Miri as a key tool in developing a standard for Rust that avoids this kind of ambiguity. Specifying undefined behavior by means of an interpreter like Miri has several key advantages: first of all, it is not hard to extract a formal mathematical specification from the interpreter—but interpreter source code is much more accessible to programmers than formal specifications, making it easier to involve the Rust community in the discussion of what should and should not be undefined behavior in Rust. Secondly, we can avoid the pitfall of putting unrealistic requirements on unsafe code by simply running Miri on a corpus of unsafe code and testing if any of that code violates the requirements we want to impose on unsafe Rust authors. This is crucial to raise acceptance of an eventual official specification of undefined behavior: if there is a common sentiment that avoiding undefined behavior is near impossible and almost all programs have undefined behavior anyway (and that sentiment is not uncommon in the C/C++ communities), then it becomes very hard to convince people to rid their code of undefined behavior—which would entirely subvert any attempt to put Rust's safety story on solid footing!

I am already in contact with the Rust language team, as well as several teams in academia and industry that are interested in Rust verification, on the topic of developing a specification. I plan to use my experience with Stacked Borrows and my connections with these key stakeholders to work towards making the official specification of Rust precise, accessible, and realistic.

Low-barrier safety proofs for unsafe code. The RustBelt project leverages Iris's support for modular reasoning about low-level concurrent code to achieve a landmark result: an iron-clad guarantee of safety for a large subset of the Rust programming language, as well as for many Rust libraries which internally rely on unsafe features of the language. However, RustBelt is just the first step in a long-term research program with the goal of enabling programmers to ensure that their unsafe code is upholding Rust's safety standards.

In particular, in the version of RustBelt in my thesis, some aspects of Rust were modeled in an idealized manner or not at all. Notably, RustBelt does not account for Rust's *trait* mechanism, which is used pervasively in production code. Some unsafely implemented libraries crucially rely on there being no more than one implementation of a trait for any given type, or they may even rely on the *absence* of an implementation. This leads to a subtle interplay of reasoning about unsafe code and trait implementations, making a proof of correctness for them even more challenging than for the libraries covered by RustBelt so far. Other missing features of Rust include automatic destructors (which are challenging when considering recursive data structures) and pinning (which enables unsafe code to rely on the fixed location of some data in memory). I expect that establishing the soundness of these crucial features will require significant extensions to the RustBelt framework.

That said, the biggest hurdle is the effort required to perform a safety proof for an unsafely implemented library. In the RustBelt part of my thesis, I proved safety of a number of representative Rust libraries. However, these proofs all required significant manual input, both to translate the library into an Iris proof obligation and to make the resulting proof of that obligation go through. To achieve maximum impact on Rust programming practice, I plan to lower the barrier of entry to verification of unsafe code, by building *automatic tool support* to enable developers of Rust libraries to verify the safety of those libraries themselves. This is inspired by RefinedC [18], an Iris-based tool for automated, annotation-based verification of real C code.

One key advantage in unsafe code verification is that the types programmers put on the public API of their library already determine the specification they need to satisfy. However, those specifications are often a lot more complicated than what automated tools can handle today (in particular when they involve the Rust mechanisms of borrowing and lifetimes), so this project will require breakthrough results in automated program verification. For particularly tricky libraries, I expect some guidance from the programmer will be crucial; this requires new approaches to appropriately combine interactive and automated verification strategies without expecting the user to be a verification expert. Such an automatic tool will allow us to establish machine-checked safety guarantees for a much larger subset of real-world Rust programs, thus significantly improving the overall safety of the Rust ecosystem. Putting such a tool into the hands of programmers would *make unsafe Rust just as safe as the rest of Rust*, finally fully realizing the dream of safe systems programming in practice.

Summary

The modern world is full of complicated software systems that are prone to bugs, sometimes with severe consequences. Verification can ensure the absence of bugs even in cases where testing is infeasible since there are too many ways the system can behave. At the same time, real-world systems are typically not monolithic; they consist of a multitude of components developed by separate teams. I work on developing formal foundations that enable verification to follow the component-wise structure of large-scale software systems, and on tools for verification researchers and software engineers that bring those foundations to fruition. My general research approach could be described as striving for elegance and simplicity in the theoretical foundations, and then carrying those foundational insights all the way to attack problems that matter in practice.

References

[1] James Bornholt et al. "Using lightweight formal methods to validate a key-value storage node in Amazon S3". In: *SOSP*. ACM, 2021, pp. 836–850. DOI: 10.1145/3477132.3483540.

[2] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying concurrent, crash-safe systems with Perennial". In: *SOSP*. ACM, 2019, pp. 243–258. DOI: 10.1145/3341301.3359632.

[3] Tej Chajed, Joseph Tassarotti, Mark Theng, **Ralf Jung**, M. Frans Kaashoek, and Nickolai Zeldovich. "Gojournal: A verified, concurrent, crash-safe journaling system". In: *OSDI*. USENIX Association, 2021. URL: https://www.usenix.org/system/files/osdi21-chajed.pdf.

[4] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. "RustBelt meets relaxed memory". In: *PACMPL* 4.POPL (2020), 34:1–34:29. DOI: 10.1145/3371102.

[5] **Ralf Jung**. "A formal look at pinning". Blog post. 2018. URL: https://www.ralfj.de/blog/2018/04/05/a-formal-look-at-pinning.html.

[6] **Ralf Jung**. "How MutexGuard was Sync when it should not have been". Blog post. 2017. URL: https://www.ralfj.de/blog/2017/06/09/mutexguard-sync.html.

[7] **Ralf Jung**. "Safe intrusive collections with pinning". Blog post. 2018. URL: https://www.ralfj.de/blog/2018/04/10/safe-intrusive-collections-with-pinning.html.

[8] **Ralf Jung**. "The tale of a bug in Arc: Synchronization and data races". Blog post. 2018. URL: https://www.ralfj.de/blog/2018/07/13/arc-synchronization.html.

[9] **Ralf Jung**, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. "Stacked Borrows: An aliasing model for Rust". In: *PACMPL* 4.POPL (2020). DOI: 10.1145/3371109.

[10] **Ralf Jung**, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the foundations of the Rust programming language". In: *PACMPL* 2.POPL (2018). DOI: 10.1145/3158154.

[11] **Ralf Jung**, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "Safe systems programming in Rust: The promise and the challenge". In: *CACM* (2021). DOI: 10.1145/3418295.

[12] **Ralf Jung**, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. "Higher-order ghost state". In: *ICFP*. 2016. DOI: 10.1145/2951913.2951943.

[13] **Ralf Jung**, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018). DOI: 10.1017/S0956796818000151.

[14] **Ralf Jung**, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning". In: *POPL*. 2015. DOI: 10.1145/2676726.2676980.

[15] Robbert Krebbers, Jacques-Henri Jourdan, **Ralf Jung**, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. "MoSeL: A general, extensible modal framework for interactive proofs in separation logic". In: *PACMPL* 2.ICFP (2018). DOI: 10.1145/3236772.

[16] Robbert Krebbers, **Ralf Jung**, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. "The essence of higher-order concurrent separation logic". In: *ESOP*. Vol. 10201. LNCS. 2017. DOI: 10.1007/978-3-662-54434-1_26.

[17] Robbert Krebbers, Amin Timany, and Lars Birkedal. “Interactive proofs in higher-order concurrent separation logic”. In: *POPL*. 2017. DOI: 10.1145/3093333.3009855.

[18] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. “Refinedc: Automating the foundational verification of C code with refined ownership types”. In: *PLDI*. ACM, 2021, pp. 158–174. DOI: 10.1145/3453483.3454036.

[19] Matthieu Sozeau. “PhD position on certified extraction in Nantes”. 2020. URL: https://coq.discourse.group/t/phd-position-on-certified-extraction-in-nantes/1044.

[20] Joshua Yanovski, Hoang-Hai Dang, **Ralf Jung**, and Derek Dreyer. “GhostCell: Separating permissions from data in Rust”. In: *PACMPL* 2.ICFP (2021). DOI: 10.1145/3473597.