Research Statement

Ralf Jung

Improving the reliability of software is a critical problem. My research is about developing *compositional formal methods* to increase software reliability. Formal methods complement other techniques, such as testing, which has a lower bar of entry but does not work well in the face of high degrees of non-determinism (such as in concurrency) or malicious adversaries (such as in security). In order to be able to handle massive, real systems such as OS kernels or web browsers, I am convinced that we have to develop compositional formal methods. Instead of analyzing a large system all at once, a compositional analysis proceeds by determining *abstraction boundaries*, verifying that the desired property holds true for each component, and showing that composing components preserves the property.

Probably the most widespread form of compositional analyses are *type systems*, which are commonly used to show that programs are memory safe, but can also scale to other properties. I work on verifying the correctness of type systems by considering the types on a library interface as describing a *semantic contract* between the library and its client. The contract is set up such that it is automatically satisfied by well-typed code, but the key strength of this approach is that non-well-typed code can still be shown to follow the contract. Non-well-typed code can arise when working with data structures that are type-safe for reasons too subtle for the type system to understand, like a low-level implementation of a data structure that is heavily optimized to reduce memory consumption and increase performance. The semantic approach enables the compositional integration of well-typed and non-well-typed components: only the non-well-typed pieces need to be carefully analyzed to show that the entire program enjoys the type system guarantees.

Defining the precise contract between a library and its (typed) clients can require a sophisticated program logic. This connects to another focus of my research: I work on a logic that is able to give modular specifications to algorithms and data structures in a higher-order, concurrent, imperative language. Beyond the scope of typical program logics, this logic is also suited to verify type systems (using the approach described in the previous paragraph) and it can handle relational properties, such as compiler correctness.

The confidence derived from a correctness proof (be it of a type system, a logic, or some program component) entirely rests on that proof being carefully checked for errors. To increase that confidence and reduce the amount of burdensome manual proof checking, all of my work is formalized in the Coq Proof Assistant.

Ongoing research

Rust type system

Recently, the main focus of my research has been the Rust programming language. Rust is a young programming language (released in 2015) developed by Mozilla, and used not just in Mozilla's Firefox browser, but also by companies such as

Dropbox, Google and Amazon. Its key selling point is combining *type safety* properties such as memory safety and data-race freedom with a programming model that enables optimizing for *performance* like one would in languages such as C or C++. 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. These unsafe blocks are typically hidden behind an API surface with the promise 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 in higher-level code without worrying about incorrect usage of the library's interface.

We have built the first formal proof of correctness of the Rust type system [9] as part of the *RustBelt*[1] research project. We also verified the correctness 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 interacting with these libraries is still memory safe and data-race free. 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. To achieve this, we built a *semantic model* of Rust's type system, formalizing the contract that well-typed functions must satisfy to be safely callable.

Our work has led to the discovery of a bug in `Mutex` [5] (implementing mutual exclusion and used for lock-based concurrent programming), and follow-on work by collaborators that incorporates weak memory into RustBelt [2] 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* [4, 6], a cornerstone in Rust's growing support for supporting asynchronous programming.

This research is specific to Rust, but the general approach is not: unsafe escape hatches similar to `unsafe` blocks are a common feature even in type-safe languages (such as `Obj.magic` in OCaml, `unsafePerformIO` in Haskell), and the same methodology can also be applied to reason about the interaction of a type-safe language with a non-type-safe language (such as calls from/to code written in C, which most languages allow).

Key challenge: Interior mutability. One particularly interesting class of unsafely implemented libraries are those that exhibit *interior mutability*. These libraries violate what is otherwise a core type system invariant: data is either mutable or aliased, but never both. In contrast to that, types like `Mutex` enable mutation in the presence of aliasing. The reason such types can still be used safely is that they only permit mutation of aliased state in a highly controlled manner. For example, `Mutex` performs run-time checks to make sure only one thread ever actually gets access to the data and can mutate it. The key to handling interior mutability was to not only let library types pick their own invariant as part of their contract with the environment, but also to let them pick their own mechanism for *sharing*.

[1] Project website: https://plv.mpi-sws.org/rustbelt/

Iris

To give a semantic model of Rust's type system requires a logic that is expressive enough to reason about ownership and borrowing. How to handle ownership using *separation logic* has been explored in great depths over the last decades, and the last couple of years have seen a significant boost in expressive power enabling these logics to reason about subtle interactions of concurrent algorithms and data structures. All of these logics typically come 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.

The core design principle of *Iris*[2] [12, 10, 15, 11] is to focus on a small, expressive core logic that enables the user of the logic to *derive* new reasoning principles without having to extend the core logic. In particular, the reasoning principles of many previous logics can be encoded inside Iris, not only giving a higher-level justification for their correctness, but also enabling the composition of program components that were verified with different techniques. I have led the development of Iris from the start, and at this point Iris has been used in several other verification projects [17, 20, 13, 21, 19, 3, 1, 9] and has more than 130 citations (according to Google Scholar).

For example, Iris' ability to incorporate new reasoning principles was crucial in verifying Rust's type system, where we developed the *lifetime logic* as a way to reason about borrowing. Proving the correctness of the lifetime logic required combining many of Iris' features in a novel way, but we did not have to extend Iris itself—the existing mechanisms provided by Iris were flexible enough to also handle the new concepts introduced by the lifetime logic. This logic is thus just a set of lemmas proven in Iris, and can be composed with any other Iris proof.

Iris is being developed in the Coq Proof Assistant. Thanks to a proof mode developed mainly by my collaborator Robbert Krebbers, it is the first separation logic to provide the usual user experience of interactive proof assistants, including support for higher-order quantification and magic wand [16, 14]. This has been instrumental not only to increase confidence in our results, but also to enable the continued development of the foundations of Iris. Proof assistants enable fearless refactoring and avoid tedious manual re-checking of existing proofs.

Logical atomicity. Having a powerful and expressive program logic is just one ingredient to verifying a library. The other important aspect is to figure out *which specification to prove*. Even seemingly simple questions like "What is a stack?" are not easy to answer when concurrency comes into play. One common approach is to use linearizability or contextual refinement, which means that the complex real implementation can be replaced by a much simpler, "obviously correct" implementation for the purposes of reasoning. That simple implementation effectively serves as specification.

The problem with this approach is that linearizability is external to a program logic and thus cannot be directly applied when verifying, *inside* the logic, clients that use the data structure. For example, concurrent separation logics including Iris typically give special power to *atomic* operations: they cannot be interrupted by other threads, and hence they can get exclusive access to shared state for an atomic instant. The `push` operation of a linearizable stack, however, is not

[2]Project website: https://iris-project.org/

an atomic operation from the perspective of the program logic. Even though linearizability means that it is *observably atomic*, it internally consists of many program steps, and hence does not get the same special power that "physically" atomic operations have.

A promising approach to mitigate this problem is to define a *logical notion of atomicity*, in the form of Hoare triples that not only specify pre- and postconditions, but also indicate that the operation happens atomically. Such operations are robust to interference from other threads and hence permit exclusive access to shared state in a way that is otherwise only permitted for "physically" atomic operations.

Unlike prior approaches to logical atomicity that bake in such a notion as a logical primitive, we have followed the Iris methodology and defined logical atomicity *inside* Iris. We can thus use any reasoning principle encoded in Iris to verify a logically atomic specification, and compose proofs carried out using different techniques. We have used this to verify the correctness of an elimination stack, a subtle concurrent data structure that relies on one thread completing the action of another. In ongoing research, we are showing that this approach even scales to handling prophecy variables, where the order in which multiple atomic actions seem to occur is not fixed until after the fact.

Specifying programming language semantics

The verification of a program relies on a precise specification of the language the program is written in. Unfortunately, for low-level languages such as Rust, developing a specification that is precise, suited for reasoning about programs, and matches real program behavior is still an open problem. I have been working on this problem from two angles.

Developing a specification for Rust. I have worked closely with Mozilla and the Rust community during two summer projects (2017 and 2018) to help flesh out a specification for unsafe code in Rust. (RustBelt made some simplifying assumptions to avoid having to deal with the open questions in the semantics of Rust.) One challenge here is that the Rust compiler developers would like to have optimizations that exploit the fact that data is never both aliased and mutable. However, unsafe code is able to subvert these type system guarantees.

We would like the language specification to declare this an illegal operation in unsafe code (making it *undefined behavior*), such that it is the responsibility of the unsafe code author to make sure it does not subvert assumptions the compiler is making. I have worked on a set of rules that achieves this, and implemented these rules in an interpreter that can be used to check if real code complies with the new requirements [7]. This work has already found some bugs in the standard library, but it is still very much in progress.

Moreover, I am an active member of the Rust Unsafe Code Guidelines Working Group[3], which has the goal of developing a set of guidelines for unsafe code authors to follow in order to ensure that their code does not violate Rust's safety guarantees. At this stage, we are determining the current set of *de-facto* rules for unsafe code, and starting to build a framework for an eventual proper specification of undefined behavior in Rust.

[3]Project website: https://github.com/rust-rfcs/unsafe-code-guidelines/

Clarifying the semantics of LLVM IR. There still is a big gap between the semantics of Rust and what actually gets executed on the machine: Rust is built on top of the LLVM library, which comes with its own intermediate representation (IR). To argue that Rust's type system guarantees are preserved by compilation, we need to first understand the semantics of the LLVM IR. LLVM IR is heavily optimized, and one challenge in defining the semantics of the LLVM IR is making sure that all these optimizations are actually correct under the proposed semantics—or else demonstrating that some of them are incompatible with each other and cannot soundly coexist on the same language. I have worked with collaborators on a formal model of the LLVM IR that justifies significantly more transformations than any previous proposal [18].

Future research

Expanding RustBelt

The RustBelt model of Rust does not model some important aspects of the language such as panics (which is a mechanism similar to exceptions), and automatic destructors (which have a special kind of unsafe escape hatch). These are obvious candidates for future work.

Moreover, my summer projects with Mozilla on determining rules for aliasing in unsafe code have not been incorporated into RustBelt yet. I would like to formalize these rules in Coq and show that the Rust type system and the libraries we verified are sound with respect to the aliasing model. Moreover, I would also like to prove that the desired transformations can indeed soundly be carried out if all code is assumed to follow these rules.

Verifying a Rust library currently requires a manual translation from the Rust source code to the formal language we use in RustBelt, called λ_{Rust}. I would like to adjust λ_{Rust} to be closer to one of the intermediate languages in the Rust compiler (MIR, the mid-level intermediate language that the interesting analyses like checking borrowing operate on) so that we can have an automatic translation from the Rust source code to the formal Coq development.

Such a formal model of MIR would also enable a direct verification of the next-generation Rust borrow checker, dubbed "Polonius" (the borrow checker is the part of the Rust compiler that enforces the no-aliased-mutable-state property). Polonius is described as a set of clauses in the spirit of logic programming, so a formal verification of the actual algorithm used by the compiler should be feasible. The Rust developers are always interested in relaxing the borrow checker to accept more programs, and have expressed interest in collaborating to ensure that this does not introduce unsoundness.

Another interesting avenue is to incorporate my work on a formal semantics of LLVM: I would like to show that Rust's translation strategy from MIR to the LLVM IR is actually correct. This will require modeling of LLVM features that have not been formally specified yet, like the `noalias` attribute.

Beyond linearizability

As described above, I have used Iris to explore a notion of logical atomicity, a program-logic equivalent to linearizability, in order to specify correctness of

concurrent data structures such as stacks. Logical atomicity works great for specifying operations that can be described as a single action, but falls short when talking about operations that consist of multiple observably separate actions, such as map/fold on collections, or traversal of a tree (e.g., file system lookup). I would like to research methods to scale logically atomic specifications to those situations.

Moreover, so far, logical atomicity has only been studied in the context of sequential consistency. When moving away to more realistic models of concurrent memory, it is not clear whether linearizability-like approaches such as logical atomicity are still the best way to specify components of a concurrent system. Other notions of library correctness have been proposed, and I would like to research how they can be integrated with the existing work on weak memory in Iris [13, 2].

More automation for Rust verification

So far, all my verification efforts have been entirely manual. I would like to collaborate with experts in automated verification techniques to work towards a tool that can automatically or at least semi-automatically verify unsafe Rust code against the RustBelt model. This is a very long-term project. The ideal outcome would be a tool that can translate a library's typed API and code into proof obligations, discharge as many of them as possible automatically, and in case the automation gets stuck let the user provide manual guidance in some high-level logic geared towards reasoning about Rust types. The tool would run as part of regular automated testing to ensure that the library and its proof remain in sync. The tool could translate successful proofs into certificates that can be verified by Coq, enabling a manual proof in Coq as a (sound and compositional) fall-back mechanism and resulting in a proof from first principles that the library satisfies its API contract.

References

[1] Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. *Iron: Managing Obligations in Higher-Order Concurrent Separation Logic*. To appear in POPL. 2019.

[2] Hoang-Hai Dang, Jacues-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. *RustBelt Relaxed*. Submitted for publication. 2018.

[3] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency". In: *LICS*. 2018, pp. 442–451.

[4] **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.

[5] **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.

[6] **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.

[7] **Ralf Jung**. *Stacked Borrows Implemented*. Blog post. 2018. URL: https://www.ralfj.de/blog/2018/11/16/stacked-borrows-implementation.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**, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *PACMPL* 2.POPL (2018), 66:1–66:34.

[10] **Ralf Jung**, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. "Higher-order ghost state". In: *ICFP*. 2016, pp. 256–269.

[11] **Ralf Jung**, Robbert Krebbers, Jacues-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), e20.

[12] **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, pp. 637–650.

[13] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris". In: *ECOOP*. Vol. 74. LIPIcs. 2017, 17:1–17:29.

[14] 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), 77:1–16:30.

[15] 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, pp. 696–723.

[16] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive Proofs in Higher-Order Concurrent Separation Logic". In: *POPL*. 2017, pp. 205–217.

[17] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. "A Relational Model of Types-and-Effects in Higher-Order Concurrent Separation Logic". In: *POPL*. 2017, pp. 218–231.

[18] Juneyoung Lee, Chung-Kil Hur, **Ralf Jung**, Zhengyang Liu, John Regehr, and Nuno P. Lopes. "Reconciling High-level Optimizations and Low-level Code in LLVM". In: *PACMPL* 2.OOPSLA (Oct. 2018), 125:1–125:28. ISSN: 2475-1421.

[19] David Swasey, Deepak Garg, and Derek Dreyer. "Robust and Compositional Verification of Object Capability Patterns". In: *PACMPL* 1.OOPSLA (2017), 89:1–89:26.

[20] Joseph Tassarotti, **Ralf Jung**, and Robert Harper. "A Higher-Order Logic for Concurrent Termination-Preserving Refinement". In: *ESOP*. Vol. 10201. LNCS. 2017, pp. 909–936.

[21] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. "A Logical Relation for Monadic Encapsulation of State: Proving contextual equivalences in the presence of `runST`". In: *PACMPL* 2.POPL (2018), 64:1–64:28.