Derek Dreyer

MPI-SWS, Germany

Research Statement

December 2018

My research is motivated by a simple goal: to provide *rigorous formal foundations* for establishing the safety and reliability of *realistic software systems*. My general strategy in attacking this goal is to develop formal methods—such as *type systems* and *separation logics*—that, due to their support for *compositional* reasoning, are ideally suited to exploit the inherently compositional way in which complex software systems are built. The main challenge I face is that most prior work on such compositional methods makes big simplifying assumptions—for example, that languages can be proven type-safe using simple syntactic methods, or that concurrent programs obey sequentially consistent semantics—assumptions which are not valid for, and thus do not scale to handle, realistic systems. In my research, I re-examine these assumptions and build new formal foundations that take real-world complexity seriously.

At the moment, my work revolves primarily around a project I am leading called **RustBelt**, which has been funded (beginning in April 2016) by a 5-year, 2-million-euro ERC Consolidator Grant. This project ties together several strands of work that I have been (and will continue) pursuing in the areas of types, concurrent separation logic, and relaxed memory models. In this research statement, I will describe the high-level objectives of the project, as well as how my recent, ongoing, and planned future research addresses them.

RustBelt: Logical Foundations for the Future of Safe Systems Programming

The RustBelt project is motivated by a longstanding open question in the design of languages for programming complex software systems: **How can we provide programmers with both *safety* and *control*?**

On the one hand, languages like C and C++ give programmers low-level control over resource management, which is necessary for systems programming tasks, but this comes at the expense of safety. On the other hand, languages like Java give programmers safe high-level abstractions and automatic resource management, but this comes at the expense of control. This apparent tradeoff between safety and control is deeply unsatisfactory and has inspired a vast amount of research in programming language design on how to marry high-level safety with low-level control. On the academic side, there have been many proposals to use *types* to enforce various "ownership disciplines" for safe manual resource management, but these disciplines tend to be too restrictive in practice and thus none have made it out into mainstream industrial use. Rather, the *lingua franca* of real-world systems programming is still C++. Although "modern C++" (since the 2011 standard) includes several features (like smart pointers, move semantics, and RAII-style APIs) that encourage safer programming idioms, the type system of C++ is too weak to enforce these idioms statically, so it is still easy to write catastrophically unsafe code using these features.

The language that many believe holds the greatest promise as an industrial-strength answer to this challenge is **Rust** [33]. Rust is a new language developed at Mozilla Research that has been garnering a significant amount of buzz among systems programmers and researchers alike. (In a recent Stack Overflow developer survey, it was voted the "most loved" programming language of 2016.) The reason for the excitement is simple: Rust was deliberately designed from the start to support a style of systems programming close to modern C++ in expressiveness and efficiency, *together with* a state-of-the-art "ownership" type system guaranteeing that Rust programs are type-safe, memory-safe, and data-race-free. Rust is the only industry-supported systems programming language to claim such safety guarantees. As such, Rust has the potential to revolutionize systems programming by overcoming the safety/control tradeoff, making it possible to build software systems that are safe by construction, without having to give up low-level control over performance and resource management.

Unfortunately, there is a major problem: none of Rust's safety claims have been formally investigated, it is not at all clear whether they actually hold, and we don't know how to verify them using existing technology.

The root of the problem is that the Rust type system is deliberately—and *necessarily*—conservative. It cannot account for the safety of every advanced form of low-level programming that one finds in the wild, because there is no practical way to do so while retaining automatic typechecking. Rather, in order to be effectively implementable, Rust's type system focuses on providing the basic building blocks for common safe systems programming idioms. As a consequence, though, if the programmer wants to write some code that does not fit into these idioms, they must

employ `unsafe` blocks, an escape hatch which allows them to use potentially unsafe C-style features like the ability to dereference a "raw pointer". Ideally, `unsafe` code should be used only sparingly and *encapsulated* inside safe abstractions, so that the vast majority of Rust code is still written in the safe fragment of the language. However, the sheer number of `unsafe` blocks in Rust's standard library (over 1000 at present) calls its safety guarantees into question. Furthermore, `unsafe` is not limited to existing libraries: Rust programmers will need to occasionally use `unsafe` code in the development of future Rust libraries as well. Currently, the advice given to such programmers is essentially to "be careful," because even the Rust developers lack a clear understanding of what it means to write `unsafe` code in a "safe" way, as evidenced by a variety of embarrassing safety violations that have been uncovered in the Rust standard library in the past few years. This state of affairs leads us to the following key question:

How can we ensure that existing uses of `unsafe` do not fatally undermine Rust's safety guarantees, and how can we provide programmers with formal tools to help them use `unsafe` in a "safe" way?

Answering this question for Rust will have far-reaching impact on safe systems programming in general because `unsafe` code is simply a fact of life—any realistic languages targeting this domain in the future *must* provide an escape hatch like `unsafe` to afford systems programmers the low-level control they expect. And yet, we have no tools for reasoning about it! The standard technology for verifying safety properties for high-level programming languages—namely, Wright and Felleisen's syntactic method [42] (aka "progress and preservation" [12])—does not apply to languages in which one can mix safe and unsafe code. (Progress and preservation is a *closed-world* method, which assumes the use of a closed set of typing rules. This assumption is fundamentally violated by `unsafe` blocks, which explicitly circumvent the typing rules.) So, to account for safe-unsafe interaction, we need a way to specify formally what we are obliged to *prove* if we want to encapsulate `unsafe` code safely behind a Rust interface, and we need verification tools to help us prove it. Currently, despite decades of semantics and verification research, we are still not to the point of having logical foundations suitable to this task. It is the goal of the RustBelt project to build them.

Finding the Right Logic for Modeling Rust

Our general approach to proving safety of Rust is to build a *semantic model* of the language. Using this semantic model, we will be able to establish formally that, even if a Rust library employs potentially unsafe features internally, it can still be "semantically safe", in the sense that linking it with other safe Rust code will never result in a program with observably unsafe behavior. Concretely, the semantic model will interpret the interface of a Rust library as a *verification condition*, a logical specification that the implementation of the library must satisfy in order to be considered semantically safe. For those libraries that are syntactically safe (*i.e.,* that do not use `unsafe` blocks), we will prove a meta-theorem establishing that such libraries satisfy their verification conditions by construction. For any libraries that employ `unsafe` blocks, we will have to prove manually that their implementations satisfy their verification conditions—but at least we will have formal safety specifications against which to verify them.

Although the general idea of building semantic models of type systems is not new, a major challenge we face is that that the type system of Rust is much more sophisticated than that of any toy λ-calculus for which prior semantic models have been developed. It is not at all obvious, for instance, how to apply existing techniques from program semantics to model features like Rust's novel "lifetimes" and "borrowing" mechanisms (for controlling aliasing), or its `Send` and `Sync` traits (which track whether a type is "thread-safe").

But before we can even attempt to address that challenge, there is a fundamental question we need to answer: when our semantic model translates interfaces into verification conditions, what is the right *logic* in which to express those verification conditions? Since the Rust type system relies crucially on the notion of resource ownership, and is designed to guarantee safety in the presence of fine-grained concurrency, the natural choice is to use some kind of *concurrent separation logic* (CSL), which builds in concurrency-aware ownership reasoning as a primitive notion. However, there is not just one CSL—there are many. Since O'Hearn and Brookes's seminal development of the original CSL [31, 5], which won the 2016 Gödel Prize, the last decade has seen a flurry of work on ever more expressive and powerful CSLs (*e.g.,* RGSep [41], LRG [11], CAP [10], FCSL [30], and TaDA [8]) and, more recently, on *higher-order* CSLs (*e.g.,* CaReSL [39] and iCAP [37]), which include the kind of "impredicative" quantification needed to model higher-order state (pointers to arbitrary objects—another essential feature of Rust).

There are two main problems with the CSLs developed so far, which prevent them from serving as an appropriate logic for modeling Rust:

1. **No Canonical Logic**: As Parkinson noted in *The Next 700 Separation Logics* [32], "there is a disturbing trend for each new library or concurrency primitive to require a new separation logic." And as these new CSLs become ever more expressive, they bake in increasingly baroque and bespoke proof rules as primitive, with the relationships and compatibility between different proof rules (*e.g.,* whether they can be soundly combined in one logic) remaining unclear. The complexity of existing CSLs is exacerbated by the fact that, until very recently [35], they only supported manual and error-prone "pencil-and-paper" proofs. To verify realistic Rust libraries and build confidence in the results, we need a simple logic that supports a wide variety of reasoning principles for concurrent code, and one that supports machine-assisted and machine-checked proof.

2. **The Assumption of Sequential Consistency**: Until recently, all CSLs have assumed that concurrent programs obey a *sequentially consistent* (SC) semantics. Under SC semantics, there is a single, global view of memory shared by all threads, and operations of different threads are nondeterministically interleaved, with each operation taking immediate effect on the global memory. Unfortunately, although SC semantics is simple and clear, it is not reflective of reality. For performance reasons, compilers reorder and merge memory accesses, and modern architectures employ hierarchies of caches to reduce memory latency; as a result, there is no globally consistent view of memory shared by all threads. One can regain SC semantics if needed, but this comes at the steep cost of disabling important optimizations and inserting expensive fence instructions in the compiled code. Consequently, languages like C/C++, Java, and Rust expose so-called *relaxed* (or *weak*) *memory models* to programmers so that they may take advantage of cheaper memory operations when they do not require strong consistency, and indeed several Rust libraries make use of such relaxed memory operations. Thus, to verify the safety of Rust, we need a logic that supports reasoning about relaxed memory.

I will now describe several lines of work we have been pursuing in the interest of remedying the above problems and building a semantic model for Rust. These constitute ongoing collaborations between members of my group at MPI-SWS and our international collaborators, including Lars Birkedal's group at Aarhus University, Chung-Kil Hur's group at Seoul National University, Viktor Vafeiadis's group at MPI-SWS, and Robbert Krebbers' group at TU Delft.

Iris: A Simple, Unifying Foundation for Higher-Order Concurrent Separation Logics

Across a series of papers (**POPL'15** [19], **ICFP'16** [17], **ESOP'17** [24], **JFP'18** [18]), we attacked the first problem mentioned above by developing **Iris**: a simple, unifying foundation for higher-order concurrent separation logics. The core idea of Iris is to provide a uniform way of accounting for the central source of complexity in existing CSLs, namely how they control *interference* between threads accessing shared shate. The best-known interference-control mechanism is Jones's *rely-guarantee* [14], which uses binary relations to describe the state transitions a thread may perform (the guarantee) vs. those its environment may perform (the rely). But recent advanced CSLs, like CaReSL [39], iCAP [37], and TaDA [8], provide much more sophisticated and elaborate *protocol* mechanisms, which use state transition systems of various forms to control interference more abstractly, modularly, and concisely than rely-guarantee does.

With Iris (`iris-project.org`), we showed that even the fanciest of these interference-control mechanisms could be expressed by a combination of two orthogonal "off-the-shelf" ingredients: (1) *partial commutative monoids* (PCMs) for expressing protocols on shared state, and (2) *invariants* for enforcing them. Invariants are an old and ubiquitous concept in program verification, and PCMs have been used in a number of prior logics to represent different kinds of *ghost state* (*i.e.,* logical state that is manipulated as part of the proof of a program but is not manipulated directly by the program itself). Our key observation was that in fact these two simple mechanisms are all you need! Just using PCMs and invariants, we were able to easily *derive* a number of the most powerful forms of protocol-based reasoning from prior logics *within* Iris, and moreover to develop new reasoning mechanisms (in particular, a notion of *logically atomic* specification) that went beyond the expressive power of any prior CSL.

Furthermore, we have invested significant effort in building up tool support for mechanizing Iris proofs in Coq. Initially, our project collaborator Robbert Krebbers engineered IPM (Iris Proof Mode), a tactic library for Coq that makes it possible to perform interactive proofs in Iris in much the same style as standard interactive proofs in Coq [25]. Following on IPM, we developed **MoSeL**, a Coq framework which decouples the tactics of IPM from the specifics of the Iris logic, thus bringing the benefits of IPM to a much broader class of separation logics (**ICFP'18** [23]). Using MoSeL, we are now able to support machine-checked interactive proofs not just in the Iris logic itself, but also in logics *derived within* Iris (such as iGPS [20] and Iron [3]), as well as logics developed independently [6, 7].

RustBelt: A Semantic Soundness Proof for Rust in Iris

In our **POPL'18** paper [16], we used Iris to formalize **RustBelt**, the first (machine-checked) validation of the safety of the Rust programming language. RustBelt is groundbreaking in several ways. First, it defines a core typed calculus for Rust (called λ_{Rust}), which encapsulates the central features of Rust and can serve as a basis for further formal investigation of the language. Second, it offers a semantic soundness proof that encompasses both the λ_{Rust} type system *and* a number of widely-used Rust libraries that internally employ unsafe features.

The proof makes critical use of Iris's facility for deriving new domain-specific logics. In particular, in order to give a simple and direct model of Rust's reference types, we derived (within Iris) a new Rust-oriented logic called the *lifetime logic*. Unlike traditional separation logic, which is based on the idea that resources can be divided *in space*, the lifetime logic introduces the novel ability for resources to be divided *in time*. This facility is essential for modeling what happens when an object `o` in Rust is "borrowed", since borrowing effectively splits the ownership of `o` between the borrower (who owns `o` for the duration of some "lifetime") and the original owner (who reclaims ownership of `o` once the lifetime has ended). Although "splitting ownership in time" is not a standard notion in separation logic, Iris makes it easy to derive such non-standard notions of separation and embed them in the separating conjunction connective.

iGPS: Towards a Separation Logic for the C/C++/Rust Relaxed Memory Model

In our initial work on RustBelt [16], we assumed (for the sake of simplicity) a sequentially consistent (SC) model of concurrency. However, in reality, Rust employs a relaxed memory model, inherited directly from C/C++11, and over the past several years we have been developing separation logic foundations suitable for reasoning about programs under such a relaxed memory model.

In our **OOPSLA'14** paper [40], we proposed **GPS**, a logic for C/C++11 specifically focusing on its *release/acquire* mode of memory accesses. Release/acquire is weaker (and more efficient) than SC but strong enough to support synchronization via a common "message passing" idiom. Though not the first logic for relaxed memory, GPS is the first logic to take the *modern* protocol-based reasoning supported by advanced CSLs (such as Iris) and adapt it to be sound under C/C++11's release-acquire semantics.

The core idea of GPS is to observe that the problem with taking protocol mechanisms from SC logics and trying to use them in a relaxed-memory setting is that they assume the ability to place invariants on the contents of multiple memory locations simultaneously. In a relaxed-memory setting, this is generally unsound, since threads do not share a common view of memory. What *is* sound, however, is to restrict protocols to only govern the contents of a single location at a time. Formally, these *single-location protocols* are justified semantically by the *per-location coherence* (aka "SC per location") conditions guaranteed both by modern architectures and by language models like C/C++11's.

In GPS, we showed how to develop formal reasoning principles for single-location protocols, along with other useful mechanisms (logical synchronization via "escrows", and ghost state via PCMs as in Iris). In a subsequent paper at **PLDI'15** [38], we applied GPS to a challenging case study: the first formal verification of an implementation of the RCU (read-copy-update) synchronization mechanism (used widely in the Linux kernel) under relaxed-memory assumptions. A key limitation of GPS, however, was its lack of support for interactive, machine-checked proofs.

In our **ECOOP'17** paper [20], we addressed this limitation by deriving a variant of GPS—called **iGPS**—within our Iris logic in Coq. In order to pull this off, we had to overcome a central point of tension between Iris and C/C++11, namely: C/C++11 is defined using an *axiomatic* "event-graph" semantics, whereas Iris is only applicable to languages defined by an *operational* "interleaving" semantics. We resolved this tension by devising an operational-semantics formulation of the release-acquire fragment of C/C++11, instantiating Iris with that semantics, and then deriving the rules of GPS from the basic axioms of Iris. Our work on iGPS received the best paper award at ECOOP'17.

RustBelt Relaxed

In recent work (presently under submission [9]), we weave together all the strands of research described above to build **RustBelt Relaxed** ($\mathbf{RB}_{\mathtt{rlx}}$), the first formal validation of the soundness of Rust under relaxed memory. Although based closely on the original RustBelt, $\mathbf{RB}_{\mathtt{rlx}}$ takes a significant step forward by accounting for the safety of the more weakly consistent memory operations that real concurrent Rust libraries actually use. For the most part, we were able to verify Rust's uses of relaxed-memory operations as is. Only in the implementation of one Rust library (`Arc`) did we need to

strengthen the consistency level of two memory reads (from relaxed to acquire) in order to make our verification go through. And in one of these cases, our attempt to verify the original (more relaxed) access led us to expose it as the source of a previously undetected data race. Our fix for this race has since been merged into the Rust codebase.

The overarching challenge in developing $\mathbf{RB}_{\texttt{rlx}}$ was porting the RustBelt verification from being built on top of Iris (under SC semantics) to being built on top of (an extension of) iGPS (under C/C++11 semantics). That such a porting would be possible at all was far from obvious. In particular, the most subtle aspect of porting to iGPS concerned the lifetime logic, which played such a crucial role in the original RustBelt verification. As it turns out, *most* of RustBelt's lifetime logic—with the exception of its "atomic borrows" mechanism—remains sound under relaxed memory. As a result, much of the original RustBelt—*e.g.,* the large parts that did not depend on atomic borrows—did not have to be changed at all! However, *proving* that the lifetime logic remains mostly sound under relaxed memory—and fixing the rules for atomic borrows so that they are—required us to develop a novel concept of *view-dependent ghost state*, which we have not seen in any prior work on relaxed-memory separation logic.

Other Lines of Work in the RustBelt Project

A "promising" semantics for relaxed-memory concurrency. The C/C++11 relaxed memory model suffers from a severe flaw, known as the *out-of-thin-air* (OOTA) problem [4]. Roughly speaking, the problem is that—in trying to balance the conflicting desiderata of compiler writers, architecture vendors, and programmers—C/C++11 semantics ends up allowing certain "bad" program behaviors that break fundamental properties desired of a memory model, such as the "DRF" property [1] and the soundness of basic invariant-based reasoning. So building a logic on top of the C/C++11 model (as it stands) is impossible!

In our work on iGPS and $\mathbf{RB}_{\texttt{rlx}}$, we sidestepped this issue by targeting a strengthened version of C/C++11 in which the OOTA problem is solved by prohibiting load-store reorderings on relaxed accesses. Although this strengthening easily fixes the problem, it incurs a potential performance cost on weaker architectures like Power and ARM.

In our **POPL'17** paper [22], we proposed a very "promising" approach to solving the OOTA problem head-on. In particular, we presented the first relaxed memory model that (1) accounts for nearly all the features of the C/C++11 concurrency model, (2) provably validates a number of standard compiler optimizations, as well as a wide range of memory access reorderings that commodity hardware may perform, (3) avoids bad OOTA behaviors that break invariant-based reasoning, (4) supports "DRF" guarantees, ensuring that programmers who use sufficient synchronization need not understand the full complexities of relaxed-memory semantics, and (5) defines the semantics of racy programs without relying on undefined behaviors, which is a prerequisite for broader applicability to type-safe languages like Java. The key novel idea behind our model is the notion of *promises*: a thread may promise to execute a write in the future, thus enabling other threads to read from that write out of order.

To establish confidence in our "promising" model, we formalized most of our key results in Coq. In the course of doing so, we uncovered—much to our surprise—a previously unknown flaw in the *official* semantics of SC accesses in C/C++11: contrary to published results [2, 34], the standard compilation schemes for Power processors (as implemented in mainstream compilers) are unsound. In our **PLDI'17** paper [26], which received a distinguished paper award, we developed a fix to this problem which will be incorporated into the next C++ standard.

Mtac: A language for typed tactic programming in Coq. Effective support for custom proof automation is essential for large-scale interactive proof development of the sort we are carrying out in the RustBelt verification. However, existing languages for automation via *tactics* either (a) provide no way to specify the behavior of tactics statically within the logic of the theorem prover or (b) rely on advanced type-theoretic machinery that is not easily integrated into established theorem provers.

Developed by my student Beta Ziliani in his **ICFP'13** and **JFP'15** papers [43, 44] and subsequently by my student Jan-Oliver Kaiser in his **ICFP'18** paper [21], **Mtac** is a lightweight but powerful extension to Coq for dependently-typed tactic programming. Mtac tactics have access to all the features of ordinary Coq programming, as well as a new set of typed tactical primitives. We avoid the need to touch the trusted kernel typechecker of Coq by encapsulating uses of the new tactical primitives in a monad, and instrumenting Coq so that it executes monadic tactics during type inference. Mtac has already received significant attention from Coq users and developers, and we are aiming to evolve it to the extent that it can eventually replace Ltac and Ocaml as a more robust general-purpose tactic language for Coq.

Future Work

Further exploration of Rust. As a researcher in programming languages and formal methods, I often find it somewhat frustrating to see how long it takes for good ideas from academic PL research to make their way into mainstream practice. In contrast, one of the most satisfying aspects of working on the RustBelt project has been witnessing the reverse: the Rust developers (and the broader Rust community with which they engage) are not only deeply informed by the state of the art in PL research, they are pushing well beyond it into exciting, uncharted territory, and they are actively seeking help from academic researchers in exploring this frontier. With RustBelt, we have the rare opportunity to guide the groundbreaking exploration of a major, actively developed language by putting its design on a sound formal footing.

As part of his thesis work, my student Ralf Jung has been exploring how to define undefined behavior for Rust. The problem is that, thus far in RustBelt, we have assumed a memory model in which the only forms of undefined behavior are data races and memory safety violations. However, this is too simplistic. The Rust developers would like to support more aggressive compiler optimizations that exploit non-aliasing assumptions derived from Rust's ownership types, but in order for such optimizations to be proven sound, undefined behavior must be expanded to include unsafe code that violates such non-aliasing assumptions. Ralf has proposed a solution to this problem based on *stacked borrows* [15], for which he is implementing a dynamic checker to test the model on real Rust code. Once we obtain empirical evidence that it is a good model, we plan to develop the formal theory of stacked borrows and integrate it into RustBelt.

Going forward, we would like to enrich our formal model of Rust, both to better reflect the reality of Rust and to help guide the Rust developers as they evolve the language. First of all, we want to extend λ_{Rust} (and RustBelt) to account for Rust's *trait* system, which plays a central organizing role in the language (much as type classes do in Haskell). The Rust developers have already been experimenting with Chalk [28], a model of the trait system combining Prolog-style logic programming and modal logic. We hope to collaborate with the Rust team on the development of Chalk and to incorporate a formalization of it into RustBelt. In addition, we plan to examine other major language extensions that the Rust community is investigating, such as higher-kinded abstractions, value-dependent types, and improvements to Rust's borrow checker. Lastly, we plan to use RustBelt to verify the safety of more sophisticated libraries built atop `unsafe` code, particularly those that seem like they might demand revisions to our current semantic model of Rust. Such libraries include `crossbeam` (epoch-based memory reclamation for building lock-free data structures), `indexing` (sound unchecked indexing via type generativity), and `shifgrethor` (precise tracing garbage collection as a library).

Static analysis for C++ based on Rust. Rust has the potential to revolutionize the future of systems programming, but of course at present the *lingua franca* of systems programming remains C++. In collaboration with researchers from the Infer team at Facebook, we are exploring how to use insights from Rust to develop a compositional static analysis for finding safety bugs in C++ programs. The hope is that we can use the existing Infer infrastructure to catch violations of a Rust-like ownership discipline (in the spirit of Ralf's stacked borrows [15]).

Connecting foundational and automated verification. RustBelt and Iris sit squarely in the realm of foundational verification—*i.e.,* verification performed completely in an interactive theorem prover, whose logical consistency rests on a small trusted foundation. Our MoSeL framework [23] makes such interactive verification feasible, and our Mtac tactic language [21] is designed to help programmers build their own custom proof automation in a more reliable way. But ultimately, proofs in Iris/MoSeL involve significant manual verification effort. Fortunately, there is a great deal of complementary work on verification of concurrent and imperative programs using tools like Chalice [27], VeriFast [13], and Viper [29], which provide a much higher degree of automation, but whose formal foundation is less clearly developed. In future work, we hope to connect these complementary lines of work by using Iris to provide formal soundness proofs for logics encoded in Viper.

Applications of separation logic to distributed computing. Separation logic has thus far been applied primarily to the verification of imperative, pointer-manipulating programs. At heart, however, it is merely a logic of resource ownership, which should be relevant to the modular verification of any kind of stateful programs, including, for example, *distributed programs* that employ asynchronous communication. (Indeed, separation logics for relaxed memory models already must cope with such asynchronous communication, since threads can witness writes in different orders.) There has recently been preliminary work on applying separation logic to the verification of distributed systems [36], but we believe that the field is ripe for a more systematic account and that Iris is in a good position to provide one.

Acknowledgments

The work I have described in this research statement—which has been supported by generous funding from the Max Planck Society, the European Research Council, Google, and Microsoft—would not have been possible without the efforts of my past and present advisees and collaborators at MPI-SWS and elsewhere. I would like to thank Lars Birkedal, Aleš Bizjak, Manuel Chakravarty, Arthur Charguéraud, Karl Crary, Hoang-Hai Dang, Deepak Garg, Bob Harper, Chung-Kil Hur, Jacques-Henri Jourdan, Ralf Jung, Jan-Oliver Kaiser, Jeehoon Kang, Gabriele Keller, Scott Kilpatrick, Yoonseung Kim, Robbert Krebbers, Neel Krishnaswami, Ori Lahav, Simon Marlow, Jan Menz, Craig McLaughlin, Aleks Nanevski, Georg Neis, Gaurav Parthasarathy, Simon Peyton Jones, Marianna Rapoport, Yann Régis-Gianas, Andreas Rossberg, Filip Sieczkowski, Kasper Svendsen, David Swasey, Joe Tassarotti, Amin Timany, Aaron Turon, Viktor Vafeiadis, Edward Yang, Zhen Zhang, and Beta Ziliani for their essential contributions.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. In *International Symposium on Computer Architecture (ISCA)*, 1990.

[2] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

[3] Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. Iron: Managing obligations in higher-order concurrent separation logic. *PACMPL*, 3(POPL):65:1–65:30, January 2019.

[4] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Workshop on Memory Systems Performance and Correctness (MSPC)*, 2014.

[5] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1–3):227–270, 2007.

[6] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2011.

[7] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[8] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014.

[9] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and **Derek Dreyer**. RustBelt relaxed. Submitted for publication, November 2018.

[10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[11] Xinyu Feng. Local rely-guarantee reasoning. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009.

[12] Robert Harper. *Practical Foundations for Programming Languages (Second Edition)*. Cambridge University Press, New York, NY, USA, 2016.

[13] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods (NFM)*, 2011. Invited paper.

[14] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.

[15] Ralf Jung. Stacked borrows: An aliasing model for Rust, August 2018. Published on the author's blog: `https://www.ralfj.de/blog/2018/08/07/stacked-borrows.html`.

[16] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and **Derek Dreyer**. RustBelt: Securing the foundations of the Rust programming language. *PACMPL*, 2(POPL):66:1–66:34, January 2018.

[17] Ralf Jung, Robbert Krebbers, Lars Birkedal, and **Derek Dreyer**. Higher-order ghost state. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2016.

[18] 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. *Journal of Functional Programming (JFP)*, 28(e20), November 2018.

[19] 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 *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2015.

[20] 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*, LIPIcs, pages 17:1–17:29, 2017.

[21] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and **Derek Dreyer**. Mtac2: Typed tactics for backward reasoning in Coq. *PACMPL*, 2(ICFP):78:1–78:31, September 2018.

[22] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and **Derek Dreyer**. A promising semantics for relaxed-memory concurrency. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.

[23] 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. *PACMPL*, 2(ICFP):77:1–77:30, September 2018.

[24] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, **Derek Dreyer**, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *European Symposium on Programming (ESOP)*, 2017.

[25] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.

[26] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and **Derek Dreyer**. Repairing sequential consistency in C/C++11. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[27] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *FOSAD*, volume 5705 of *LNCS*, pages 195–222, 2009.

[28] Niko Matsakis. Chalk. `https://github.com/rust-lang-nursery/chalk`.

[29] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, volume 9583 of *LNCS*, pages 41–62, 2016.

[30] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *European Symposium on Programming (ESOP)*, 2014.

[31] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[32] Matthew Parkinson. The next 700 separation logics. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, 2010. Invited paper.

[33] The Rust programming language. Mozilla Research. http://www.rust-lang.org/.

[34] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[35] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[36] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *PACMPL*, 2(POPL):28:1–28:30, January 2018.

[37] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *European Symposium on Programming (ESOP)*, 2014.

[38] Joseph Tassarotti, **Derek Dreyer**, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[39] Aaron Turon, **Derek Dreyer**, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2013.

[40] Aaron Turon, Viktor Vafeiadis, and **Derek Dreyer**. GPS: Navigating weak memory with ghosts, protocols, and separation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2014.

[41] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *International Conference on Concurrency Theory (CONCUR)*, 2007.

[42] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[43] Beta Ziliani, **Derek Dreyer**, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in Coq. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2013.

[44] Beta Ziliani, **Derek Dreyer**, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in Coq. *Journal of Functional Programming (JFP)*, 25, e12, July 2015. Special issue devoted to selected papers from ICFP 2013.