Derek Dreyer
MPI-SWS, Germany

Research Statement

April 2021

My research is motivated by a simple goal: to provide *rigorous formal foundations* for establishing the safety and reliability of *realistic software systems*. How can we *design* large-scale systems that are efficient, easy to maintain, and safe to execute, and how can we develop *programming languages* that help us program such systems more effectively? Furthermore, how can we actually *verify* that complex, real-world systems written in these languages do what they are supposed to do, and how can we provide maximum assurance that our verification efforts are on solid ground? Given the central role that programming languages play in our software infrastructure, the above questions lie at the heart of computer science and have remained among its grand challenges since the 1960s.

My general strategy in tackling these challenges is to develop formal methods—such as *type systems*, *semantic models*, and *program logics*—that support *compositional verification*. In general, "compositional verification" refers to the ability to establish the safety or correctness of a program in a modular fashion—decomposing the reasoning about a whole, complex program into reasoning about its simpler, reusable component parts. Compositionality is widely understood to be crucial for scaling program verification to handle real-world software systems with millions of lines of code. As such, it has been a central concept in programming languages and verification research since the pioneering work of John Reynolds on type systems, Dana Scott and Christopher Strachey on denotational semantics, and Tony Hoare on program logics, half a century ago.

Unfortunately, the dream of compositional verification has traditionally failed to account for the messy reality of modern software systems, because in most prior work, compositional verification methods have made significant simplifying assumptions, rendering them inapplicable to real-world programs and programming languages. For example: (1) They ignored features that are common to nearly every programming language since the 1990s (such as higher-order mutable state) because they were difficult to reason about compositionally. (2) They assumed the use of simplified "sequentially consistent" memory models of multithreaded programming rather than the "relaxed" models exhibited by real multiprocessor architectures. (3) They avoided dark corners of real programming languages, such as "unsafe escape hatches", which make it challenging to even *state* safety guarantees for such languages, let alone prove them.

In my research, I have developed a range of new formal methods that avoid these kinds of simplifying assumptions, thus making it feasible to compositionally verify *realistic* software systems that were previously out of reach of any rigorous proof. At the moment, my work revolves primarily around an ERC Consolidator project I have led for the past five years called **RustBelt** (`plv.mpi-sws.org/rustbelt`), in which we have built the first formal foundations for the Rust programming language. This project ties together several strands of work that I have been (and will continue) pursuing in the areas of types, semantics, concurrent separation logic, interactive theorem proving, and relaxed memory models. In this research statement, I will first give a brief overview of the high-level problem motivating the RustBelt project, and then describe the various strands of my research that have contributed to solving it, before concluding with a discussion of several outstanding research challenges that I aim to address in ongoing and future work.

RustBelt: Securing the Foundations of the Rust Programming Language

One of the longstanding open problems in programming languages is to develop a language that ensures *safety* (*i.e.,* that programs don't crash your machine or exhibit undefined behavior) while also providing low-level *control* over system resources. Languages like Java have strong type systems for ensuring safety, but this safety comes at the expense of control. As a result, for many *systems programming* applications—*e.g.,* web browsers, operating systems, or game engines, where performance and resource constraints are primary concerns—the only option is to use an *unsafe* language like C or C++ that provides fine-grained control. However, this control comes at a steep cost. For example, Microsoft has reported that 70% of the security vulnerabilities they fix are due to memory safety violations, precisely the type of bugs that strong type systems were designed to rule out [40]. Google has recently reported the same 70% statistic for the Android operating system [39]. Likewise, Mozilla reports that the vast majority of critical bugs they find in Firefox are memory related [14]. If only there were a way to avoid these bugs and somehow get the best of both worlds: a systems programming language with both safety *and* control…

Enter **Rust** [35, 19]. Sponsored by Mozilla and developed actively over the past decade by a large and diverse community of contributors, Rust supports many common low-level programming idioms and APIs derived from modern C++, but at the same time it guarantees statically (at compile time) that these idioms are used safely. In fact, Rust goes even further than existing safe languages by defending programmers against other, more insidious anomalies that no other mainstream language can prevent. For example, consider *data races*: unsynchronized accesses to shared memory (at least one of which is a write). Even though data races effectively constitute undefined (or weakly defined) behavior for concurrent code, most "safe" languages (such as Java and Go) permit them, and they are a reliable source of concurrency bugs [41]. In contrast, Rust's type system rules out data races at compile time.

Rust has been steadily gaining in popularity, to the point that it has topped Stack Overflow's list of "most loved" programming languages for the past five years and is now being invested in by many major industrial software vendors (such as Amazon, Google, Huawei, and Microsoft, which came together with Mozilla in 2021 to form the Rust Foundation). Given the increasing adoption of Rust in industry, it is thus of essential importance—both to the success of the language and the safety of applications built using it—that we obtain higher assurance that the language is in fact as safe as claimed. In other words: **How can we *prove* that Rust is safe?**

As it turns out, the answer to this question is very far from obvious, because Rust's expressiveness relies crucially on a subtle interplay between safe and unsafe code. Specifically, at the core of Rust is a novel *ownership type system*, in which the "lifetimes" of aliases (references) to objects are carefully tracked so that one alias cannot be used to mutate and accidentally corrupt the view of another alias. This ownership type system serves to prohibit bad behaviors like memory safety violations and data races, *but* it is also quite conservative—*i.e.,* it rules out some perfectly safe programming patterns as well. This is to be expected: no automatically checkable type system will be capable of inferring whether arbitrarily clever low-level programming patterns are safe. As a result, though, to implement certain kinds of data structures or synchronization mechanisms, Rust programmers must sometimes resort to using *unsafe* "escape hatches", such as unchecked type casts or array accesses, which circumvent the safety restrictions of the core type system. Good practice dictates that uses of such unsafe features should be *encapsulated by safe APIs*, so that well-typed clients of such APIs will be guaranteed to be safe. But in reality, such APIs have repeatedly been shown to break the soundness of Rust [4, 16, 5], undermining confidence in its safety claims. Hence, we are left with the key question: **How can we *prove* that uses of unsafe features in Rust APIs have been safely encapsulated?**

In the RustBelt project, we have supplied the first rigorous answer to this question, building on a decade of breakthrough results in semantics, program logics, and concurrency. We now proceed to describe some of the key ingredients of our solution.

Semantic Models of Modern Type Systems

The first challenge we faced is that the standard technology for verifying safety properties for high-level programming languages—namely, Wright and Felleisen's method of syntactic type soundness [45] (aka "progress and preservation" [13])—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 fixed set of typing rules. This assumption is fundamentally violated by Rust's unsafe escape hatches, which are explicitly designed to circumvent the restrictions of its safe typing rules.

To tackle this challenge, we instead adopted the approach of *semantic type soundness*. Under this approach, we build a *semantic model* of Rust, which explains how to interpret each type of the language as a *verification condition*—*i.e.,* a logical specification that every term of that type must satisfy in order to be considered *semantically safe* to use at that type. With the semantic model in hand, proving safety of Rust programs breaks into two parts. First, we prove a meta-theorem stating that all well-typed Rust terms that are syntactically safe (*i.e.,* that do not use unsafe features) are also semantically safe *by construction*. Second, for any API whose implementation makes use of unsafe features, we must manually prove that the implementation satisfies the semantic model of the API. Once we have done so, we obtain an end-to-end result that every well-typed Rust program P is safe to execute so long as the only uses of unsafe features in P are in the implementations of APIs that have been proven semantically safe.

The essential idea of semantic type soundness is very old, dating back at least to Milner's seminal paper on type soundness [30]. Milner constructed his semantic soundness proof using *denotational semantics*, in which the semantic model was defined using suitably enriched algebraic structures called *domains* [37]. However, despite several decades of subsequent research on domains, denotational models were ultimately unable to provide effective methods for reasoning about the full panoply of features found in modern programming languages: features like higher-order functions, mutable state, recursive types, abstract data types, ownership types, control operators, and concurrency.

In a series of papers [1, 10, 27, 43], my collaborators and I developed and popularized a new class of semantic models called **step-indexed Kripke logical relations (or SKLRs)**, which accounted for all the aforementioned language features used in tandem (as they are in Rust), thus overcoming the limitations of prior denotational models. SKLRs were inspired directly by two prior lines of seminal work: the *step-indexed* models of Appel and McAllester [3] and Ahmed [2] for handling recursive types and higher-order state, and the *Kripke logical relations* models of Pitts and Stark [34] for establishing invariants about the private state of objects. In addition to marrying these prior approaches together, our SKLRs significantly expanded their reasoning power by enriching the Kripke structure of invariants to include state transition systems, describing how the private state of an object may evolve over time. These more flexible Kripke structures have provided a crucial foundation for verifying the safety of real Rust APIs.

Iris: A Unifying Foundation for Modern Concurrent Separation Logics

After my initial work on SKLR models, it became clear to me that, although the models were theoretically very powerful, there was a fundamental problem with actually deploying them in practice. In short, they were *too low-level*, making it painful to carry out proofs about even small toy programs, let alone the much more complex Rust APIs whose safety we aimed to verify in RustBelt. To address this problem, I undertook a line of work [9, 11, 42] to improve the usability and scalability of SKLR models by formalizing them at a much higher level of abstraction using *program logics*. This culminated in the development of **Iris**, my most directly impactful project to date [22, 20, 26, 21].

Iris is a framework for deriving advanced forms of *separation logic* [32]. Invented a little over 20 years ago as an extension of Hoare logic for compositional verification of pointer-manipulating programs, separation logic (SL) has been a major influence on modern verification and analysis tools like VeriFast [15] and Infer [12]. In the mid-2000s, Peter O'Hearn went a step further and developed *concurrent separation logic* (CSL), which showed how the principles of SL were just as relevant to reasoning about concurrent (multithreaded) programs as they were to sequential programs [31]. Like the original SL work, CSL was enormously influential, earning the 2016 Gödel Prize, and spawning a wide variety of follow-on program logics ("CSLs") that expanded its reach to ever more challenging concurrent programming idioms. However, this development also led to a fractured field in which, to quote Matthew Parkinson's position paper *The Next 700 Separation Logics* [33], "there is a disturbing trend for each new library or concurrency primitive to require a new separation logic."

Iris addresses Parkinson's concern head-on by reducing the essence of modern CSLs to a minimal core foundation. Specifically, its design was based on the key insight that only two logical mechanisms (*user-defined ghost state* and *invariants*) were needed to *derive* the most powerful reasoning principles from earlier CSLs within a unified logic [22]. In subsequent work [21], this foundation was reduced even further, as we showed how invariants and even Hoare triples themselves could be encoded in terms of simpler primitives. Furthermore, unlike prior CSLs, Iris is language-agnostic (*i.e.,* applicable to a range of different programming languages), and is implemented in the Coq proof assistant, making it possible for users of Iris to build *interactive, machine-checked proofs* about their programs [25]. Such support for machine-checked proof is essential for building confidence in large-scale verification artifacts like RustBelt.

In the past 6 years, Iris has transformed the landscape of research on CSLs, with more than 40 published papers from a range of contributors (including 17 POPL papers) studying or deploying Iris for verification of complex programs and programming language meta-theory in C, Rust, Go, OCaml, Scala, and other widely-used languages (`iris-project.org`). The startup BedRock Systems has adopted Iris as a central component of its systems verification technology, and the reach of Iris has extended into the systems community as well, with renowned systems researchers Frans Kaashoek and Nickolai Zeldovich of MIT re-orienting their crash-safe verification project around Iris [6, 7].

The most advanced application of Iris, though, has been in **RustBelt**, where we use it to encode a machine-checked SKLR model of Rust [18]. Iris is a great fit for encoding such a semantic model for two key reasons. First, by virtue of being a separation logic, Iris comes with built-in support for compositional, high-level reasoning about *ownership*, a central concept in the Rust type system. Second, the RustBelt semantic soundness 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 derive (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 soundly within the existing separating conjunction connective of the logic.

Separation Logic for Relaxed Memory Models

In the initial work on RustBelt [18], we assumed (for the sake of simplicity) a sequentially consistent (SC) model of concurrency, in which threads take turns interacting with a single, shared global state. This is a common assumption in work on concurrent program verification, but it does not reflect reality. In fact, Rust follows C and C++ in employing a *relaxed memory model*, which offers significantly weaker and more complex guarantees about the ordering of concurrent memory operations, as well as a wider range of options for programmers to choose from in deciding how strongly synchronizing they want their memory accesses to be.

In order to adapt RustBelt to more accurately account for the relaxed memory operations that real concurrent Rust APIs actually employ, our first step was thus to develop a sound program logic for the C++11 memory model (on which Rust is based). Our GPS logic [44] paved the way in this regard: it was the first CSL to support modern reasoning principles (such as shared-state "protocols") in a manner carefully restricted to be sound under C++11's weaker notion of *release-acquire* consistency. In our work on iGPS [23], we showed furthermore how GPS could be encoded in Iris, which in turn enabled us to begin building the first machine-checked proofs about relaxed-memory C++/Rust programs.

These developments fed directly into our subsequent work on **RustBelt Relaxed** [8], in which we showed how to adapt RustBelt to relaxed memory. The overarching challenge we faced 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 *synchronized ghost state*.

Independently of Rust, my collaborators and I have also made other fundamental contributions to the study of relaxed memory models. Notably, we developed a novel "promising" solution to C++11's longstanding "out-of-thin-air" problem, which has been the basis of much follow-on research [24]. We also uncovered and fixed a flaw in the official C++11 semantics of SC accesses, and our fix was subsequently incorporated into the language standard [28].

Ongoing and Future Work

Further exploration of the foundations of Rust. With RustBelt, we have taken a big leap forward, but it is only the first step. We now plan to use it as a springboard for further exploration of the actively evolving Rust language.

A natural direction for future work is to test the mettle of our current RustBelt model by using it—or adapting it as needed—to account for the safety of more "daring" Rust APIs which use unsafe features in subtle ways. In work under submission, we have already adapted RustBelt to prove soundness of `GhostCell`, a Rust API for safely implementing data structures with internal sharing [46]. In ongoing work, we are exploring how to use RustBelt to verify soundness of the "pinning" API, which underlies Rust's recently introduced support for asynchronous programming.

In another line of work, we have contributed to the *design* of Rust itself through our work on **Stacked Borrows** [17]. Stacked Borrows is a new semantics for Rust memory accesses, geared toward validating more aggressive compiler optimizations for Rust that exploit the strong aliasing information inherent in Rust types. As with RustBelt, the tricky part of developing the Stacked Borrows semantics was dealing with the interaction between safe and unsafe code, and ensuring that unsafe code does not break the type-based aliasing guarantees expected by safe code. My former PhD student Ralf Jung, who led the development of Stacked Borrows, has worked closely with the Rust Unsafe Code Guidelines working group (of which he is co-lead) to gain acceptance of the Stacked Borrows design and implement it in Miri, the experimental reference interpreter that ships with Rust. In addition to exploring further refinements of the Stacked Borrows semantics, our obvious next step for future work is to integrate Stacked Borrows into RustBelt.

Expanding the expressive power of Iris. Although Iris is already a mature verification tool, my collaborators and I continue to explore ways to expand its expressive power through further development of its foundations.

One key limitation of Iris is that, due to its *step-indexed* semantic foundation, it is restricted to proving *safety* properties (*i.e.,* that programs do not do anything bad); *liveness* properties (*i.e.,* that programs eventually do something good) have been out of reach. In recent work [38], we have argued that the main problem with supporting liveness verification is that the step-indexed model of Iris fails to validate the *existential property*, which says that existential quantification inside the logic corresponds to existential quantification outside the logic. To address this problem, we developed **Transfinite Iris**, which replaces Iris's step-indexed model with one based on ordinals so that it validates the existential property. We have demonstrated that Transfinite Iris supports liveness reasoning for higher-order, *sequential* programs, and in future work, we are exploring how to support liveness reasoning about *concurrent* programs as well.

Independently of liveness, we are studying several promising alternatives to the traditional step-indexed foundations of Iris in the interest of avoiding the oft-perceived complexity of step-indexed reasoning. First, we are developing a variant of Iris in which proofs that require tricky step-indexed reasoning (or which are not presently doable at all) can be simplified (or made possible) by treating step-indices themselves as a kind of *resource*. Second, we are investigating how to disentangle Iris's support for step-indexed reasoning from its support for ownership reasoning. Specifically, we are building a framework for coinductive simulation proofs, which inherits Iris's strong support for compositional reasoning about ghost state and ownership while avoiding the complexities of its step-indexed model altogether.

Automating the foundational verification of realistic systems code. In our work so far on RustBelt and Iris, the main focus has been on *foundational verification—i.e.,* verification performed completely in an interactive theorem prover (Coq), whose logical consistency rests on a small trusted foundation. Iris enables one to derive high-level logics in which to conduct very tricky proofs about realistic systems code, concerning deep correctness properties that are presently beyond the reach of any other rigorous formal methods. However, the proofs still require a large amount of manual human effort. Thus, in order to scale these tools to large code bases and put them eventually in the hands of programmers (rather than merely Coq experts), we need much better support for *automated proof*.

Recently, my collaborators and I have developed **RefinedC** [36], the first tool for automated *and* foundational verification of functional correctness for C programs. The key idea of RefinedC is to facilitate automated, goal-directed proof search using a combination of ownership types (à la Rust) and *refinement types*, which annotate C types with data structure invariants. These types are then given semantics—in the style of the RustBelt model—by interpreting them as Iris predicates and proving the RefinedC typing rules foundationally sound using Iris in Coq. In this way, RefinedC supports the automation afforded by tools like VeriFast [15], while also reducing the trusted code base of those tools by outputting foundational proofs in Coq. To drive further research on RefinedC, we are currently deploying it as part of a large multi-institution project, sponsored by Google, to verify functional correctness of the pKVM hypervisor.

We are also exploring automated foundational verification for *Rust* code. In one project, we are studying how to adapt the RefinedC approach to work for Rust instead of C. This appears to require a significant rethink of the system, since RefinedC does not support anything like the "borrowing" pattern of ownership transfer so prevalent in Rust code. In another project, we are investigating how to use the RustBelt infrastructure to build a semantic soundness proof for Matsushita *et al.*'s recently developed RustHorn tool [29], so that we can then link automated RustHorn verification of safe Rust code together with Iris verification of unsafe Rust APIs and obtain an end-to-end correctness guarantee.

Acknowledgments

I would like to thank my past and present advisees and collaborators (at MPI-SWS and around the world) for all their essential contributions to the work I have described here. I would also like to thank the Max Planck Society, the European Research Council, Google, Microsoft, and Mozilla for their generous support.

References

[1] Amal Ahmed, **Derek Dreyer**, and Andreas Rossberg. State-dependent representation independence. In *POPL*, 2009.

[2] Amal Jamil Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

[3] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683, 2001.

[4] Ariel Ben-Yehuda. std::thread::JoinGuard (and scoped) are unsound because of reference cycles, 2015. Rust issue #24292. `https://github.com/rust-lang/rust/issues/24292`.

[5] Christophe Biocca. std::vec::IntoIter::as_mut_slice borrows &self, returns &mut of contents, 2017. Rust issue #39465. `https://github.com/rust-lang/rust/issues/39465`.

[6] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *SOSP*, 2019.

[7] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a concurrent, crash-safe journaling system using JrnlCert. In *OSDI*, 2021.

[8] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and **Derek Dreyer**. RustBelt meets relaxed memory. *PACMPL*, 4(POPL), 2020.

[9] **Derek Dreyer**, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2:16):1–37, June 2011. An earlier version appeared in *LICS*, 2009.

[10] **Derek Dreyer**, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming (JFP)*, 22(4&5):477–528, 2012. An earlier version appeared in *ICFP*, 2010.

[11] **Derek Dreyer**, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful ADTs. In *POPL*, 2010.

[12] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at Facebook. *Commun. ACM*, 62(8):62–70, 2019.

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

[14] Diane Hosfelt. Implications of rewriting a browser component in Rust, 2019. Blog post. `https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/`.

[15] 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.

[16] Ralf Jung. MutexGuard<Cell<i32>> must not be Sync, 2017. Rust issue #41622. `https://github.com/rust-lang/rust/issues/41622`.

[17] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and **Derek Dreyer**. Stacked borrows: An aliasing model for Rust. *PACMPL*, 4(POPL), 2020.

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

[19] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and **Derek Dreyer**. Safe systems programming in Rust. *Commun. ACM*, 64(4):144–152, April 2021.

[20] Ralf Jung, Robbert Krebbers, Lars Birkedal, and **Derek Dreyer**. Higher-order ghost state. In *ICFP*, 2016.

[21] 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*, 28(e20):1–73, November 2018.

[22] 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.

[23] 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*, 2017.

[24] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and **Derek Dreyer**. A promising semantics for relaxed-memory concurrency. In *POPL*, 2017.

[25] 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), 2018.

[26] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, **Derek Dreyer**, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *ESOP*, 2017.

[27] Neelakantan R. Krishnaswami, Aaron Turon, **Derek Dreyer**, and Deepak Garg. Superficially substructural types. In *ICFP*, 2012.

[28] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and **Derek Dreyer**. Repairing sequential consistency in C/C++11. In *PLDI*, 2017.

[29] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. RustHorn: CHC-based verification for Rust programs. In *ESOP*, 2020.

[30] Robin Milner. A theory of type polymorphism in programming. *JCSS*, 17(3):348–375, 1978.

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

[32] Peter W. O'Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, 2019.

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

[34] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *Higher-Order Operational Techniques in Semantics (HOOTS)*, 1998.

[35] The Rust team. The Rust programming language, 2020. `http://rust-lang.org/`.

[36] 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*, 2021.

[37] Dana S. Scott. Domains for denotational semantics. In *ICALP*, 1982.

[38] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, **Derek Dreyer**, and Lars Birkedal. Transfinite Iris: Resolving an existential dilemma of step-indexed separation logic. In *PLDI*, 2021.

[39] Jeff Vander Stoep and Stephen Hines. Rust in the Android platform, 2021. Blog post. `https://security.googleblog.com/2021/04/rust-in-android-platform.html`.

[40] Gavin Thomas. A proactive approach to more secure code, 2019. Blog post. `https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/`.

[41] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in Go. In *ASPLOS*, 2019.

[42] Aaron Turon, **Derek Dreyer**, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, 2013.

[43] Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and **Derek Dreyer**. Logical relations for fine-grained concurrency. In *ICFP*, 2013.

[44] Aaron Turon, Viktor Vafeiadis, and **Derek Dreyer**. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, 2014.

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

[46] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and **Derek Dreyer**. GhostCell: Separating permissions from data in Rust, March 2021. Submitted for publication.