Derek Dreyer
MPI-SWS, Germany

Research Statement

April 2017

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 strains 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 and how my recent and ongoing 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** [26]. 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 of "progress and preservation" [34]—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.

In building our semantic model of Rust, one major challenge we face is 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 scale existing work up to model features like Rust's novel "lifetime-and-borrowing" mechanism (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 [23, 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 [33], LRG [8], CAP [7], FCSL [21], and TaDA [6]) and, more recently, on *higher-order* CSLs (*e.g.,* CaReSL [31] and iCAP [29]), 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* [24], "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 [28], 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. **Sequential Consistency**: Existing CSLs assume 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, modern architectures execute memory operations speculatively or out of order, and they employ hierarchies of buffers to reduce memory latency, with the effect that there is no globally consistent view of memory shared by all threads. Languages like C/C++, Java, and Rust (which is built on top of LLVM) expose these *relaxed 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 weaker 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 finding the right logic for modeling Rust. These constitute ongoing collaborations between members of my group at MPI-SWS and those of Lars Birkedal's group at Aarhus University, Chung-Kil Hur's group at Seoul National University, and Viktor Vafeiadis's group at MPI-SWS.

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

In our **POPL'15** paper [12], 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* [10], 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 [31], iCAP [29], and TaDA [6], 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, 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 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.

Since developing the initial version of Iris (1.0), we have extended it to support *higher-order ghost state—i.e.,* the ability to express richer forms of ghost state that depend recursively on the language of Iris assertions—a useful feature not present in a general form in any existing CSL (**ICFP'16** [11]). We have also shown that the essence of Iris can be reduced to a very simple base logic comprising a mere handful of modalities, and that the more semantically complex mechanisms in Iris 1.0 (such as Hoare triples) can be derived completely within this base logic (**ESOP'17** [16]). This provides evidence that Iris is really as canonical a foundation for higher-order concurrent separation logic as we claim.

Moreover, we have invested significant effort in building up tool support for mechanizing Iris proofs in Coq. Most recently, our collaborator Robbert Krebbers (formerly a postdoc of Lars Birkedal and now an assistant professor at

TUDelft) has engineered a powerful new "Iris proof mode" for Coq, described in a POPL'17 paper [17], which provides Coq-style tactics for interactive proofs performed *within* Iris, embedded in Coq. We are making crucial use of both higher-order ghost state and the new Iris proof mode in our ongoing efforts to formalize a semantic model of Rust.

GPS: A Modern Concurrency Logic for C/C++11's Release-Acquire Semantics

In the first phase of the RustBelt project, we are building a semantic model of Rust in Iris, and we are—for simplicity—assuming a sequentially consistent (SC) model of concurrency. However, to account for Rust code in its full generality, we will need to eventually lift this assumption. Although the Rust memory model has yet to be precisely defined, the language is compiled to LLVM and inherits the basic features of its concurrency model from C/C++11. Thus, in an orthogonal line of work, we have been trying to figure out how to verify programs that make use of relaxed-memory operations under C/C++11 semantics.

In our **OOPSLA'14** paper [32], we developed **GPS**, a separation logic for the C/C++11 relaxed memory model, 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 protocol mechanisms in existing logics (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 per-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 per-location protocols, along with other useful mechanisms (logical synchronization via "escrows", and ghost state via PCMs as in Iris), all verified sound in Coq against Batty *et al.*'s formal axiomatic semantics of C/C++11 [2]. In a subsequent paper at **PLDI'15** [30], 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.

Popping back up to the big picture: Iris and GPS provide complementary benefits. Iris offers a simple foundation for higher-order concurrent separation logics, with support for machine-assisted verification, but it is restricted at present to reasoning about languages with an SC-style (interleaving) operational semantics. GPS supports useful reasoning principles for release-acquire accesses, but it is verified sound against a completely different axiomatic style of semantics. For verifying safety of real Rust libraries, we will need the capabilities of both logics.

Fortunately, in recent work published at POPL'16, Lahav *et al.* [18] demonstrated that release/acquire accesses can be given an alternative characterization via an interleaving operational semantics. By instantiating Iris with their operational semantics, we are currently exploring whether we can encode the GPS proof rules *within* Iris in the same way that we have already encoded other SC logics. If this approach succeeds, it will enable us to (1) combine Iris-style reasoning for SC accesses with GPS-style reasoning for release/acquire accesses in a single logic, and (2) exploit our existing Iris proof mode to mechanize proofs in the combined logic.

A "Promising" Semantics for Relaxed-Memory Concurrency

GPS marks a significant step forward in verification technology for realistic concurrent code. But in order to verify the safety of Rust libraries that use relaxed-memory operations, we will need to ultimately develop a logic that accounts for the full spectrum of C/C++11 features that these libraries use, which is not limited to just SC and release-acquire.

There is a big problem, however: the full C/C++11 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 full C/C++11 model (as it stands) is impossible! Furthermore, C/C++ is not alone in this problem: the Java memory model suffers from a complementary flaw, in that although it prohibits OOTA behaviors, it fails to validate some basic optimizations performed by mainstream Java

compilers. Over the past decade, a number of proposals have been put forth for how to fix the OOTA problem, but none have been proven to validate the full range of standard optimizations and instruction reorderings performed by Java and C++ compilers and by commodity hardware like Power and ARM. Furthermore, for most of the existing proposals, it is known that indeed they do *not* validate some important reorderings.

In our **POPL'17** paper [13], we propose a very "promising" way forward. In particular, we present 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. Crucially, to prevent OOTA behaviors, a promise step requires a thread-local certification that it will be possible to execute the promised write even in the absence of the promise.

To establish confidence in our model, we have 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 [3, 27], the standard compilation schemes for Power processors (as implemented in mainstream compilers) are unsound. In our forthcoming **PLDI'17** paper, we develop a fix to this problem for the official axiomatic semantics of C/C++11 [19], but it remains to be seen how to account properly for SC accesses in our "promising" semantics.

Taking GPS as a starting point, the natural next step for future work is to figure out how to develop a logic for concurrent code running under our "promising" semantics. We expect to use Rust libraries like `Arc`, `Channel`, and `Crossbeam`, which use a large feature set of C/C++11 concurrency, as a source of inspiration in determining what proof principles we must support in our logic.

Other Projects

I conclude by briefly describing some other recent and ongoing projects in which I am involved besides RustBelt.

Mtac [ICFP'13, JFP'15]: 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.

Mtac [36, 37], which was the central contribution of my student Beta Ziliani's PhD thesis [35], is a lightweight but powerful extension to Coq for supporting 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 received significant attention already from Coq users and developers. Since his PhD, Beta has continued to evolve Mtac into MetaCoq [38], a more full-featured language for realistic tactic programming, in the hope that it can eventually replace Ltac and OCaml as a general-purpose tactic language for Coq.

Backpack [POPL'14]: Retrofitting Haskell with Separate Modular Development Haskell is one of the most actively developed functional programming languages. It has served as a testbed for a great deal of research on type system design and has had a major impact on the design of newer languages like Rust. However, one important type system feature that Haskell has neglected is proper support for large-scale modular programming. Haskell does not even allow programmers to write down *interfaces* for program components, and thus does not allow modules to be developed separately from the specific modules on which they depend. Unfortunately, although there has been a huge amount of work on module system design, nearly all prior work has taken a clean-slate approach, ignoring the practical concern of how to integrate stronger support for modular programming into weakly modular languages like Haskell.

In this project, which is joint work with Simon Peyton Jones (lead developer of GHC, the leading Haskell compiler), we developed **Backpack** [15], a new approach to retrofitting a weak module system like Haskell's with separately typecheckable *packages*. The design of Backpack is inspired by the MixML module calculus that Andreas Rossberg and I developed in earlier work [25], but differs significantly in detail because it is motivated less by theoretical elegance and more by the practical problem of integration into the existing Haskell ecosystem. My student Scott Kilpatrick has laid the formal groundwork for Backpack in his POPL'14 paper and forthcoming PhD thesis, and Edward Yang (a student of David Mazières and John Mitchell at Stanford) has implemented a fully-realized design based on Backpack, which is on track to be incorporated into a future release of GHC. Edward's work, which has involved a two-year collaborative effort with our group, will be described in his forthcoming PhD thesis.

Pilsner [ICFP'15] and SepCompCert [POPL'16]: Compositional Compiler Verification In order to gain full confidence in the safety of a language like Rust, it does not suffice to only consider the high-level semantics of the language itself—one must also prove that the language's semantics is preserved by the Rust compiler when it generates machine code. This extremely difficult *compiler verification* problem is beyond the five-year scope of RustBelt. However, thinking long-term, we want to start building the foundations that would be needed to tackle it.

Ten years ago, Xavier Leroy demonstrated in his landmark CompCert project [20] that, for a simpler language like C, the notoriously challenging problem of compiler verification *is* actually within the capabilities of modern interactive theorem proving technology. He built a compiler for a significant subset of C, which is realistic in its range of optimizations and is fully verified in Coq. However, CompCert, like most prior work on the subject, only verifies correctness of compilation for *whole programs*. This does not correspond to how compilers are typically used, namely to compile one module at a time. To support separate compilation, it is important to develop a compositional notion of compiler correctness that is both *modular* (preserved under linking) and *transitive* (supports multi-pass compilation).

We have initiated two projects that make fundamental advances to the state of the art in *compositional compiler verification*. In the **Pilsner** project [22], which is the heart of my student Georg Neis's forthcoming PhD thesis, we built a verified compiler (in Coq) for an ML-like core language. Pilsner is not only modular and transitive, but also *flexible*, in the sense that it can be used to safely link the results of *different* compilers that use different intermediate representations, as well as hand-written assembly code that is proven to correctly implement some source module. To achieve flexibility, we relied on a new relational reasoning technique that we developed, called *parametric simulations*, which combines the benefits of simulations and logical relations [9].

Pilsner was groundbreaking, but required a significant verification effort (several person-years of work and around 50K lines of Coq to verify a relatively simple compiler). In the **SepCompCert** project [14], we showed that it is possible to verify separate compilation *much* more cheaply if one gives up on flexibility and only attempts to verify the linking of modules compiled by the *same* compiler. In particular, using a very simpleminded technique, we were able to adapt the existing verification of CompCert to one supporting separate compilation in only two person-months, and the resulting SepCompCert verification is only 3% larger than the original CompCert.

In future work, we aim to explore the problem of compositional compiler verification for *concurrent* languages. An obvious challenge problem in this domain would be to verify correctness of compilation from a concurrent C-like language—equipped with our "promising" semantics—down to one of the weaker architectures (*e.g.,* Power or ARM).

Acknowledgments

The work I have described in this research statement—which is supported in part by generous funding from the European Research Council, Google, and Microsoft—would not have been possible without the efforts of my past and present collaborators at MPI-SWS and elsewhere. I would like to thank Lars Birkedal, Aleš Bizjak, Hoang-Hai Dang, Chung-Kil Hur, Jacques-Henri Jourdan, Ralf Jung, Jan-Oliver Kaiser, Jeehoon Kang, Scott Kilpatrick, Yoonseung Kim, Robbert Krebbers, Neel Krishnaswami, Ori Lahav, Simon Marlow, Craig McLaughlin, Aleks Nanevski, Georg Neis, Simon Peyton Jones, Andreas Rossberg, Filip Sieczkowski, Kasper Svendsen, David Swasey, Joe Tassarotti, 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. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge, 2014.

[3] 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.

[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] 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.

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

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

[9] Chung-Kil Hur, **Derek Dreyer**, Georg Neis, and Viktor Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

[10] 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.

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

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

[13] 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.

[14] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, **Derek Dreyer**, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016.

[15] Scott Kilpatrick, **Derek Dreyer**, Simon Peyton Jones, and Simon Marlow. Backpack: Retrofitting Haskell with interfaces. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

[16] 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.

[17] 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.

[18] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016.

[19] 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.

[20] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009.

[21] 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.

[22] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, **Derek Dreyer**, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2015.

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

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

[25] Andreas Rossberg and **Derek Dreyer**. Mixin' up the ML module system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 35(1), Article 2, April 2013.

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

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

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

[35] Beta Ziliani. *Interactive Typed Tactic Programming in the Coq Proof Assistant*. PhD thesis, Saarland University, March 2015.

[36] 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.

[37] 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.

[38] Beta Ziliani, Yann Régis-Gianas, and Jan-Oliver Kaiser. The next 700 safe tactic languages. Submitted for publication, October 2016.