RustBelt Relaxed

Hoang-Hai Dang
Max Planck Institute for Software Systems (MPI-SWS)
Saarland Informatics Campus, Germany
haidang@mpi-sws.org

Jacques-Henri Jourdan
LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay
France
jacques-henri.jourdan@lri.fr

Jan-Oliver Kaiser
Max Planck Institute for Software Systems (MPI-SWS)
Saarland Informatics Campus, Germany
janno@mpi-sws.org

Derek Dreyer
Max Planck Institute for Software Systems (MPI-SWS)
Saarland Informatics Campus, Germany
dreyer@mpi-sws.org

Abstract

The Rust programming language aims to support safe systems programming by means of a strong ownership-tracking type system. In their prior work on RustBelt, Jung et al. began the task of setting Rust's safety claims on a more rigorous formal foundation. Specifically, they used Iris, a Coq-based separation logic framework, to build a machine-checked proof of semantic soundness for a λ-calculus model of Rust, as well as for a number of widely-used Rust libraries that internally employ unsafe language features. However, they also made the significant simplifying assumption that the language is sequentially consistent.

In this paper, we adapt RustBelt to account for the relaxed-memory operations that concurrent Rust libraries actually use, in the process uncovering a data race in the `Arc` library. We focus primarily on the most interesting technical problem: how to adapt the "lifetime logic" (an essential component of the RustBelt proof) to be sound under relaxed memory.

ACM Reference Format:
Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt Relaxed. In *Proceedings of Some ACM SIGPLAN Conference on Programming Languages (PL'19).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Rust [14] is a new programming language—sponsored by Mozilla and developed actively over the past decade by a diverse community of contributors—that aims to bring safety to the world of systems programming. Specifically, Rust provides low-level control over data layout and resource management à la modern C++, while at the same time offering strong high-level guarantees (such as type and memory safety) that are traditionally associated with safe languages like Java. In fact, Rust takes a step further, statically preventing anomalies like data races and iterator invalidation that safe languages typically permit. Rust strikes its delicate balance between safety and control using a *substructural* type system, in which types not only classify data but also represent *ownership* of resources, such as the right to read, write, or deallocate a piece of memory. By tracking ownership in the types, Rust is able to prohibit dangerous combinations of mutation and aliasing, a well-known source of programming pitfalls and security vulnerabilities in both C/C++ and Java.

Only recently has Rust begun to receive attention from the programming languages community. Notably, the RustBelt project [7] was launched in 2016 in order to set the safety claims of Rust on a more rigorous formal foundation. The initial work on RustBelt by Jung et al. [10] made two main contributions. First, they proposed a formal definition of a core typed calculus called λ_{Rust}, which encapsulates the central features of the Rust language. Second, they used the Coq proof assistant to verify formally that Rust's aforementioned safety guarantees do in fact hold for λ_{Rust}.

However, the initial work on RustBelt also made a significant simplifying assumption: It assumed a sequentially consistent model for concurrent memory accesses. On the one hand, sequential consistency [17]—*i.e.*, an interleaving semantics in which threads take turns accessing the global state, and all threads share the same view of that state—has long been the standard memory model assumed by research on concurrency verification. On the other hand, this assumption does not match the reality of modern multicore programming languages, Rust included.

In reality, following C/C++11 (hereafter, C11), Rust provides a *relaxed* (or *weak*) memory model, supporting a variety of different consistency levels for shared-memory accesses [2]. For programmers who demand strong synchronization, *sequentially consistent* (SC) accesses are available, but this strength comes at the cost of disabling standard compiler optimizations and inserting expensive memory fences into the compiled code. The weaker consistency levels of

PL'19, February 29, 2019, New York, NY, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

release/acquire and *relaxed* allow one to trade off synchronization strength in return for more efficient compiled code. Rust employs a variety of these different consistency levels in several of its widely-used concurrency libraries, such as `Arc`, `Mutex`, and `RwLock`. But in the initial RustBelt verification effort, all atomic (*i.e.*, potentially racy) memory accesses were treated as having the strongest consistency level, SC.

In this paper, we present **RustBelt Relaxed** (or **RB$_{\texttt{rlx}}$**, for short), the first formal validation of the soundness of Rust under relaxed memory. Although based closely on the original RustBelt, **RB$_{\texttt{rlx}}$** 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 in the library. Our fix for this race has since been merged into the Rust codebase.

Before we can describe **RB$_{\texttt{rlx}}$** in greater detail, let us begin by briefly reviewing the structure of the original RustBelt verification on which it is based.

RustBelt. The structure of RustBelt is motivated primarily by the need to account for the *extensible* nature of "safety" in the Rust language. Specifically, at the heart of Rust is an ownership-based type system, which rules out bad combinations of mutation and aliasing, yet is expressive enough to typecheck many common systems programming idioms. Nonetheless, certain kinds of functionality (*e.g.*, some pointer-based data structures, synchronization abstractions, garbage collection mechanisms) cannot be implemented within the strictures of Rust's type system. Rust provides these abstractions instead via *libraries* whose implementations internally utilize **`unsafe`** features (*e.g.*, accessing raw pointers whose aliasing is unchecked). These libraries are *claimed* to be safe extensions to Rust because they encapsulate their uses of **`unsafe`** code in "safe APIs". However, given that the set of such extensions is far from fixed—new and surprising "safe APIs" are being developed all the time—there is a pressing need to understand what property an internally-**`unsafe`** library ought to satisfy to be deemed a safe extension to Rust.

To formalize Rust's extensible notion of safety, RustBelt employs a so-called *semantic soundness* proof [8]. (This is in contrast to the usual *syntactic*, "progress-and-preservation" proof [24, 9], which only applies to programs that do not use **`unsafe`** features.) The high-level idea of a semantic soundness proof is simple: (1) You define a *semantic model* of types: a mapping from types `T` to logical predicates on terms $\Phi(e)$, which asserts what it means for the term e to *behave* safely at type `T` (even if internally e uses **`unsafe`** code). (2) You prove that the syntactic typing rules of the language respect this semantic model. (3) For any library that makes use of **`unsafe`** code, you verify manually that the library's implementation satisfies the semantic model of its API. Put together, these imply that if a program is well-typed and does not use **`unsafe`** features except in the libraries verified in (3), then it is safe to execute—*i.e.*, its behavior is well-defined.

In RustBelt, Jung et al. [10] developed a semantic soundness proof for λ_{Rust}, and they instantiated point (3) by manually verifying the semantic safety of a number of widely-used Rust libraries that internally use **`unsafe`** features, including `Arc`, `Rc`, `Cell`, `RefCell`, `Mutex`, and `RwLock`.

To carry out such a proof for a language as complex as Rust, and to make the manual verification of tricky Rust libraries feasible, Jung et al. relied on *separation logic* [19]—in particular, a state-of-the-art higher-order concurrent separation logic framework called *Iris* [11]. It is easy to see why separation logic is a good fit for Rust: It is designed around the same notion of *ownership* as Rust's type system, and thus provides built-in support for ownership-based reasoning. Furthermore, the Iris framework was designed to make it easy to derive the soundness of domain-specific logics, a facility which Jung et al. exploited in order to derive a new Rust-oriented logic they called the *lifetime logic*. The lifetime logic proved crucial in enabling a relatively simple and direct semantic model of Rust's "reference types", along with high-level reasoning principles for the associated Rust mechanisms of *lifetimes* and *borrowing* [14, §4.2, §10.3].

Adapting RustBelt to relaxed memory. The overarching challenge in developing **RB$_{\texttt{rlx}}$** is that the logical foundation on which RustBelt is built is unsound for relaxed memory. More precisely, the Iris framework is parameterized by an operational semantics for the language under consideration, and depending on how this parameter is instantiated, Iris can be used to derive inference rules of varying strength. In the case of RustBelt, this parameter was instantiated with a sequentially consistent (SC) semantics for λ_{Rust}; and this "standard" instantiation (call it Iris-SC) supported a strong logical mechanism for placing *invariants* on arbitrary regions of shared memory. Iris-SC's invariants were gainfully employed in the RustBelt proof, both in the verification of concurrent libraries and in establishing the soundness of the aforementioned lifetime logic. Unfortunately, they were *too strong* to be sound under a relaxed memory model—under relaxed memory, different threads can observe writes to different locations in different orders, so one cannot in general maintain an invariant on multiple locations simultaneously.

Fortunately, there is nothing forcing us to instantiate Iris with an SC semantics; indeed, in prior work, Kaiser et al. [12] have already shown how one can just as well instantiate Iris with a relaxed memory model. The only catch is that one must come up with an operational-semantics formulation of the memory model (in contrast to the "axiomatic" style

of memory models like C11). In particular, Kaiser et al. instantiated Iris with an operational version of RA+NA—the fragment of C11 featuring release-acquire atomic (RA) and non-atomic (NA) accesses. They then used Iris to derive a separation logic for RA+NA called iGPS, which—although weaker than Iris-SC—was sufficiently powerful to verify several challenging relaxed-memory data structures (in Coq).

iGPS offers a natural starting point for us—to a first approximation, we are going to rebuild RustBelt on top of it. In so doing, however, we must tackle several technical challenges:

1. iGPS only targets RA+NA. For $\mathbf{RB_{rlx}}$, we must adapt iGPS to account for the larger fragment of C11 that the Rust libraries we are verifying actually depend on (including relaxed accesses and release/acquire fences).
2. The lifetime logic, which played such an essential role in RustBelt, was previously proven sound in Iris-SC. For $\mathbf{RB_{rlx}}$, we must adapt the lifetime logic to be sound under relaxed memory, and then prove that it is.
3. We must now verify the real, relaxed-memory implementations of concurrent Rust libraries, whose correctness is in some cases significantly more subtle than under the assumption of SC.

We will focus the majority of the paper on how we tackle the second of these challenges, since it is our most novel and (we believe) interesting technical contribution. In particular, it turns out that *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 verification—*e.g.*, the large parts that do not depend on atomic borrows—need not 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.

The remainder of the paper is structured as follows. First, we use a simple running example (message-passing between two threads) to review the basics of RustBelt's lifetime logic in §2 and separation logic for relaxed memory in §3. Next, in §4 and §5, we articulate the problem with naively combining the lifetime logic and relaxed memory, and we show how view-dependent ghost state solves the problem. Finally, in §6, we discuss other contributions of $\mathbf{RB_{rlx}}$, and we conclude in §7 with related work.

2 RustBelt's Lifetime Logic

Separation logic enriches traditional Hoare logic so that logical assertions denote not only facts about the program state but also *ownership of resources*. And as with ownership in real life, it is essential in separation logic to be able to *transfer* ownership from one piece of a program (*e.g.*, one thread) to another. In §2.1, we begin by reviewing the traditional *direct* style in which such ownership transfer is achieved in

$$\{X \mapsto \lightning * Y \mapsto \lightning\}\ X := 0; Y := 0;\ \{X \mapsto 0 * Y \mapsto 0\}$$
$$\{X \mapsto 0 * \mathsf{Send}_Y(X \mapsto -) * \mathsf{Recv}_Y(X \mapsto -)\}$$

$\{X \mapsto 0 * \mathsf{Send}_Y(X \mapsto -)\}$	$\{\mathsf{Recv}_Y(X \mapsto -)\}$
$X := 42;$	$\mathbf{if}\,({}^{*\mathbf{sc}}Y\ != 0)$
$\{X \mapsto 42 * \mathsf{Send}_Y(X \mapsto -)\}$	$\{X \mapsto -\}$
$Y :=_{\mathbf{sc}} 1;\ \{\mathsf{True}\}$	$X := 57;\ \{X \mapsto 57\}$

Figure 1. Message-Passing verified with direct transfer.

SendRecv-Create
$$Y \mapsto 0 \Rrightarrow \mathsf{Send}_Y(P) * \mathsf{Recv}_Y(P)$$

SC-Send
$$\{\mathsf{Send}_Y(P) * P\}\ Y :=_{\mathbf{sc}} 1\ \{\mathsf{True}\}$$

SC-Recv
$$\{\mathsf{Recv}_Y(P)\}\ {}^{*\mathbf{sc}}Y\ \{v.\ v = 0 \lor P\}$$

Figure 2. Separation-logic specification for SendRecv.

separation logic. Then, in §2.2, we motivate the more *indirect* style of ownership transfer exhibited by Rust's borrowing mechanism, and in §2.3, we review how RustBelt's lifetime logic supports formal reasoning about it. Throughout this section, we assume a strong SC semantics for concurrency.

2.1 Direct Ownership Transfer

Consider the Message-Passing (MP) example in Fig. 1. The example assumes two memory locations X and Y and starts by initializing X and Y to 0 before spawning two threads. Thread 1 writes 42 to X, then writes 1 to Y to send a message that thread 2 may now access X. Thread 2 tries to read Y, and if it reads a nonzero value, it knows it can safely write to X. Crucially, the accesses to Y are *atomic* (SC)—in C11 lingo—while the accesses to X are *non-atomic*. The difference is that it is safe for atomic accesses to race, while it is unsafe for non-atomic accesses to race. In particular, it is considered undefined behavior in C11/Rust for a program to contain two accesses to the same location, one (or both) of which is non-atomic and one (or both) of which is a write, which are not synchronized (*i.e.*, in a "happens-before" ordering).

Intuitively, MP is safe because the atomic accesses to Y ensure that thread 1's non-atomic write to X happens before thread 2's, so the writes to X do not race. To establish this formally, we introduce the separation-logic specification SendRecv shown in Fig. 2, which enables ownership of some resource—described by the assertion P—to be *directly* transferred from one thread to another via message-passing.[1] In reality, separation logics like Iris provide more general logical mechanisms than SendRecv; we use this bespoke spec here merely to streamline the presentation.

The first rule, SendRecv-Create, transforms the *points-to* assertion $Y \mapsto 0$ (representing ownership of the location

[1] We refer the reader to Kaiser et al. [12] for an implementation of SendRecv and a more detailed proof of MP in Iris-SC.

Y, together with the knowledge that Y currently points to 0) into two assertions $\mathsf{Send}_Y(P)$ and $\mathsf{Recv}_Y(P)$. These assertions are joined by the *separating conjunction* ($*$) connective and can thus be passed to two different threads to allow concurrent accesses to Y. (Note the use of the fearsome-looking *logical update* connective $\Rrightarrow\!\!\!\!\!\!\diagup$ to transform $Y \mapsto 0$ into $\mathsf{Send}_Y(P) * \mathsf{Recv}_Y(P)$. This connective, which is an essential feature of the Iris framework, is basically a much more flexible version of the magic wand (or linear implication) connective $-\!\!*$ from standard separation logic.[2] But to a first approximation, the reader can pretend that $\Rrightarrow\!\!\!\!\!\!\diagup$ is $-\!\!*$.)

The next rule, SC-Send, says that a thread owning $\mathsf{Send}_Y(P)$ has permission to write 1 to Y if it also owns P; and in so doing, it gives up its ownership of P. Finally, SC-Recv says that a thread owning $\mathsf{Recv}_Y(P)$ has permission to read Y, thereby losing its ownership of $\mathsf{Recv}_Y(P)$, but if it reads any nonzero value v, it also *gains* ownership of P.

We can now use SendRecv to verify MP as follows. After X and Y are initialized, ownership of Y is transformed (using SendRecv-Create) into $\mathsf{Send}_Y(P) * \mathsf{Recv}_Y(P)$, with P chosen to be $X \mapsto -$ (*i.e.*, $\exists v.\ X \mapsto v$), so that Y can be used to directly transfer ownership of $X \mapsto -$ from thread 1 to thread 2. Upon forking, the Send predicate is then passed (along with $X \mapsto 0$) to thread 1, while the Recv predicate is passed to thread 2. After the first thread writes 42 to X, it sends ownership of X to thread 2 using SC-Send. Dually, thread 2 uses SC-Recv to reason that if it reads a nonzero value from Y, it knows that it has received ownership of $X \mapsto -$, so it can safely write (non-atomically) 57 to X. □

2.2 Borrowing in Rust

Although direct ownership transfer is pleasantly simple, it is unfortunately not sufficient to explain a key feature of the Rust language, namely its *borrowing* mechanism. We first explain what borrowing is, and why direct ownership transfer does not easily account for it, before explaining in §2.3 how RustBelt's lifetime logic comes to the rescue.

To create a mutable reference to an object `o : T` in Rust, one *borrows* `o` for the duration of some *lifetime* `'a`, with the result being a reference value `r` of type `&'a mut T`. Borrowing causes the ownership of `o` to be *split in time*: While the lifetime `'a` is alive, the borrower controls the object and can use `r` to mutate it; but once `'a` is dead, the original owner of `o` can reclaim ownership of it. The reclamation that occurs once the lifetime `'a` is over is essentially a form of ownership transfer from the borrower to the original owner of `o`. And the natural question that arises when proving the safety of Rust is: How do we know that this reclamation is sound?

One might think that there is an obvious way of modeling this reclamation using direct ownership transfer: When the lifetime `'a` is over, the borrower just needs to hand ownership of the borrowed reference back to the original owner. The trouble is that this is not always possible.

For example, consider the `index_mut` function from Rust's `Vec` library, whose type is

```
fn(&'a mut Vec<i32>, usize) -> &'a mut i32.
```

This function takes a mutable reference `r` to an integer vector, along with an index `n`, and returns an interior mutable reference `e` to the `n`-th element of the vector. Crucially, thanks to Rust's substructural type system, the caller of this function *gives up* ownership of the argument `r` in exchange for the result `e`. The surrendering of `r` is quite important here because otherwise `r` could be used to subsequently mutate the object in a way that would invalidate the interior pointer `e`.

Now suppose that the argument `r` to `index_mut` had been obtained by borrowing an object `o : Vec<i32>`, *i.e.*,

```
let r = &mut o; let e = index_mut(r,n); ...
```

When `index_mut` is called here, ownership of `r` is lost, so if the borrower of `o` were required to return ownership of `r` at the end of lifetime `'a`, they would be unable to do so.

2.3 The Lifetime Logic

Given that it is not clear how to model borrow reclamation with direct ownership transfer, Jung et al. [10] took a different approach: They developed the *lifetime logic*. At a high level, the idea of the lifetime logic is to formalize the intuition mentioned above: Borrowing an object `o` for a lifetime `'a` splits ownership of `o` *in time*, between a "borrow" assertion, which the borrower can use to access `o` while `'a` is alive, and an "inheritance" assertion, which the original owner can use to reclaim ownership of `o` once `'a` is dead. Although "splitting ownership in time" is not a standard notion in separation logic, the Iris framework is designed to enable one to derive such non-standard notions of separation and embed them in the separating conjunction connective, and that is precisely what Jung et al. did.

The lifetime logic introduces several *abstract predicates* representing a variety of capabilities and permissions related to lifetimes and borrowing, together with axioms (proven sound in Iris-SC) for manipulating them, as shown in Fig. 3.[3] Let us begin with an overview of the new predicates:

- The *full borrow* $\&^{\kappa}_{\mathsf{full}} P$ asserts *temporary* ownership of resource P, *while the lifetime κ is alive*. It provides a direct means of modeling the semantics of Rust's mutable reference types.
- The *lifetime token* $[\kappa]_q$ serves as a witness that the lifetime κ is still alive. Here, q is a fraction in $(0, 1]$. If $q = 1$, we say that this is the *full token* for κ. The use

[2] As the name suggests, logical updates can *update* resources, whereas magic wands cannot. For further details, see Jung et al. [11].

[3] In Fig. 3, and throughout the paper, the proof rules are slightly simplified so as to omit occurrences of the ▷ modality [11]. This modality is an artifact of the step-indexed model of Iris, and is used to ensure consistency of the logic, but there is nothing new about our use of ▷, so we factor it out of the presentation. See the appendix for the rules with the ▷'s added back in.

LftL-begin
$\text{True} \Rrightarrow \exists \kappa.\, [\kappa]_1 * \text{Kill}(\kappa)$

LftL-kill
$[\kappa]_1 * \text{Kill}(\kappa) \Rrightarrow [\dagger\kappa]$

LftL-tok-fract
$[\kappa]_{q+q'} \Leftrightarrow [\kappa]_q * [\kappa]_{q'}$

LftL-full-bor
$P \Rrightarrow \&^\kappa_{\text{full}} P * \text{Inh}(\kappa, P)$

LftL-full-inh
$[\dagger\kappa] * \text{Inh}(\kappa, P) \Rrightarrow P$

LftL-full-split
$\&^\kappa_{\text{full}}(P * Q) \Rrightarrow \&^\kappa_{\text{full}} P * \&^\kappa_{\text{full}} Q$

LftL-full-acc
$\&^\kappa_{\text{full}} P * [\kappa]_q \Rrightarrow P * \text{Ret}(\kappa, P, q)$

LftL-full-ret
$P * \text{Ret}(\kappa, P, q) \Rrightarrow \&^\kappa_{\text{full}} P * [\kappa]_q$

Figure 3. Selected rules of RustBelt's lifetime logic (slightly simplified to omit details concerning the ▷ modality).

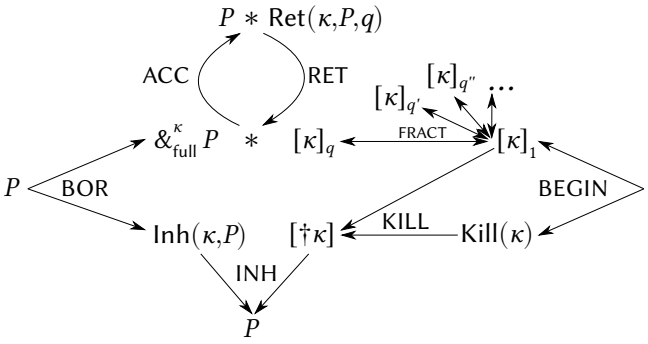

Figure 4. The life cycle of borrows and lifetimes.

of fractions allows one to share the knowledge that a lifetime is alive with multiple parties (see below).
- The *killer permission* $\text{Kill}(\kappa)$ is a unique permission needed to kill the lifetime κ.
- The *dead token* $[\dagger\kappa]$ is used to witness the knowledge that lifetime κ is dead.
- The *inheritance* $\text{Inh}(\kappa, P)$ asserts the right to reclaim the ownership of borrowed resource P once κ is dead.
- The *return policy* $\text{Ret}(\kappa, P, q)$ is used as part of the protocol for accessing the contents of a full borrow.

We briefly explain the rules in Fig. 3 with the help of Fig. 4, which depicts the life cycle of a lifetime and a full borrow. We start from the right of Fig. 4, where we create a new lifetime using LftL-begin (BEGIN in Fig. 4). This yields the full token $[\kappa]_1$ for a new lifetime κ, as well as the corresponding $\text{Kill}(\kappa)$ permission. Lifetime tokens are *fractional* (LftL-tok-fract, FRACT), so that they can be split into (and joined back from) smaller pieces which enable multiple threads to simultaneously witness that κ is still alive.

Next, on the left of Fig. 4, we see the "flagship" rule of the lifetime logic: Given ownership of any assertion P, and any lifetime κ, we can use the *borrowing rule* LftL-full-bor (BOR in Fig. 4) to create a borrow of P for κ. The rule splits ownership of P in time between two separately ownable assertions: (1) a full borrow $\&^\kappa_{\text{full}} P$ that represents ownership of P while κ is alive; and (2) an inheritance $\text{Inh}(\kappa, P)$ that can be used to reclaim P after κ dies. Intuitively, this rule directly models what happens when an object is borrowed in Rust, with the full borrow then being given to the borrower and the inheritance given to the object's original owner.

A thread owning *both* the full borrow $\&^\kappa_{\text{full}} P$ *and* a token $[\kappa]_q$ (proving κ is alive) can trade them to obtain P using the *accessing rule* LftL-full-acc (ACC in Fig. 4). As part of the trade, the thread is also given the return policy $\text{Ret}(\kappa, P, q)$. Once the thread is done using P, it trades P and $\text{Ret}(\kappa, P, q)$ to get back $\&^\kappa_{\text{full}} P$ and $[\kappa]_q$ (LftL-full-ret, RET in Fig. 4).

Once all accesses to borrows at lifetime κ are done, we can recollect the full token $[\kappa]_1$ and use the killer permission $\text{Kill}(\kappa)$ with LftL-kill (KILL) to end the lifetime. This yields the dead token $[\dagger\kappa]$. Since κ is now dead, the content P in $\&^\kappa_{\text{full}} P$ cannot be accessed any more and can thus be reclaimed. Anyone owning $[\dagger\kappa]$ and the inheritance $\text{Inh}(\kappa, P)$ can use LftL-full-inh (INH in Fig. 4) to reclaim P.

Although not depicted in Fig. 4, another crucial rule of the lifetime logic is LftL-full-split, which lets one split a borrow of $P * Q$ into separate borrows of P and Q. This rule is essential in verifying the soundness of Rust functions like `index_mut` (§2.2) that split a reference to an object into references to its subcomponents.

Finally, let us note an important safety property of the lifetime logic: *The inheritance of a borrow can only be used after all accesses to the borrowed content have finished.* The key to ensuring this is that, during an access of the borrow $\&^\kappa_{\text{full}} P$ via the accessing rule LftL-full-acc, the lifetime token $[\kappa]_q$ and the borrow assertion are "kept" by the return policy and are only returned in exchange for the borrowed content P. By withholding $[\kappa]_q$ and only returning it after the access finishes, the rule ensures that no party can have the full token $[\kappa]_1$ needed to kill κ while others are still accessing borrows associated with κ. Consequently, the inheritance can only be used *after* all accesses have finished.

However, as we will see in §4, maintaining this safety property under relaxed memory is not so straightforward.

Message-Passing in the lifetime logic. Let us now quickly demonstrate how the lifetime logic can support a somewhat different verification of the MP example than the one given in §2.1. Here, instead of transferring the location X from thread 1 to thread 2 directly, we transfer a lifetime token, which thread 2 then uses to reclaim ownership of X. The use of the lifetime logic here is clearly overkill since direct ownership transfer of X already suffices, but it will nonetheless give the reader a concrete feel for the lifetime logic in action.

In Fig. 5a, we start by creating a lifetime κ (LftL-begin). Then, with the ownership of $X \mapsto 0$, we create a borrow $\&^\kappa_{\text{full}}(X \mapsto -)$ using LftL-full-bor. Next, we instantiate

$\{[\kappa]_1 * \mathsf{Kill}(\kappa) * X \mapsto 0 * Y \mapsto 0\}$
$\{[\kappa]_1 * \mathsf{Kill}(\kappa) * \&^{\kappa}_{\mathbf{full}}(X \mapsto -) * \mathsf{Inh}(\kappa, X \mapsto -) * Y \mapsto 0\}$
$\{[\kappa]_1 * \mathsf{Kill}(\kappa) * \ldots * \mathsf{Send}_Y([\kappa]_{1/2}) * \mathsf{Recv}_Y([\kappa]_{1/2})\}$

(a) Proof of initialization.

$\{[\kappa]_{1/2} * \&^{\kappa}_{\mathbf{full}}(X \mapsto -) * \mathsf{Send}_Y([\kappa]_{1/2})\}$
$\quad\{X \mapsto - * \mathsf{Ret}(\kappa, X \mapsto -, 1/2) * \mathsf{Send}_Y([\kappa]_{1/2})\}$
$\quad X := 42; \{X \mapsto - * \mathsf{Ret}(\kappa, X \mapsto -, 1/2) * \mathsf{Send}_Y([\kappa]_{1/2})\}$
$\{[\kappa]_{1/2} * \mathsf{Send}_Y([\kappa]_{1/2})\}\ Y :=_{\mathtt{sc}} 1; \{\mathsf{True}\}$

(b) Proof of thread 1.

$\{[\kappa]_{1/2} * \mathsf{Kill}(\kappa) * \mathsf{Inh}(\kappa, X \mapsto -) * \mathsf{Recv}_Y([\kappa]_{1/2})\}$
$\mathtt{if}\,(^{*\mathtt{sc}}Y\ !\!= 0)$
$\quad\{[\kappa]_{1/2} * \mathsf{Kill}(\kappa) * \mathsf{Inh}(\kappa, X \mapsto -) * [\kappa]_{1/2}\}$
$\quad\{[\kappa]_1 * \mathsf{Kill}(\kappa) * \mathsf{Inh}(\kappa, X \mapsto -)\}$
$\quad\{[\dagger\kappa] * \mathsf{Inh}(\kappa, X \mapsto -)\}\ \{X \mapsto -\}\ X := 57; \{X \mapsto 57\}$

(c) Proof of thread 2.

Figure 5. MP verified with the lifetime logic in Iris-SC.

SendRecv for Y where the content to be sent is $[\kappa]_{1/2}$. Finally, we spawn two threads. We give a half token $[\kappa]_{1/2}$, the borrow $\&^{\kappa}_{\mathbf{full}}(X \mapsto -)$, and Send to the thread 1. We give the other half of the κ token, the killer, the inheritance, and Recv to thread 2.

In Fig. 5b, thread 1 trades the token and the borrow to access $X \mapsto -$ with LftL-full-acc and writes to X. After that, with LftL-full-ret, thread 1 trades the return policy and $X \mapsto -$ to get back the token and the borrow. Finally, thread 1 writes to Y and sends the token $[\kappa]_{1/2}$ to thread 2.

In Fig. 5c, thread 2 uses Recv to get back the full token. Owning $\mathsf{Kill}(\kappa)$, it ends the lifetime and earns the dead token $[\dagger\kappa]$ (LftL-kill). Combining that with the inheritance, thread 2 reclaims the ownership of $X \mapsto -$ (LftL-full-inh) and can safely write (non-atomically) to X. □

3 Reasoning About Relaxed Memory

Before explaining the problems of porting the lifetime logic to relaxed memory, we now briefly review the basics of relaxed memory models and their separation logics.

3.1 Understanding Relaxed Memory with Views

In the C11 relaxed memory model [2], memory accesses are not limited to SC, but can be picked from several consistency modes with varying synchronization power. At the bottom of the synchronization hierarchy sit non-atomic accesses (NA), which do not provide any synchronization and are not to be accessed concurrently. Data races on NA accesses in C11 are *undefined behavior*. In contrast, other memory access modes in C11 permit races: SC (**sc**), release (**rel**), acquire (**acq**), and relaxed (**rlx**). SC accesses always synchronize with one another, while acquire reads synchronize with release writes that they read from. Relaxed accesses do not synchronize, *unless* with the help of *memory fences* (see below).

$\{X \mapsto 0 * \mathsf{Send}_Y(X \mapsto -) * \mathsf{Recv}_Y(X \mapsto -)\}$

$\{X \mapsto 0 * \mathsf{Send}_Y(X \mapsto -)\}$	$\{\mathsf{Recv}_Y(X \mapsto -)\}$
$X := 42;$	$\mathtt{if}^{*\mathtt{rlx}}Y\ !\!= 0$
$\{X \mapsto - * \mathsf{Send}_Y(X \mapsto -)\}$	$\quad\{\nabla(X \mapsto -)\}$
$\mathtt{fence}_{\mathtt{rel}};$	$\quad\mathtt{fence}_{\mathtt{acq}};$
$\{\Delta(X \mapsto -) * \mathsf{Send}_Y(X \mapsto -)\}$	$\quad\{X \mapsto -\}$
$Y :=_{\mathtt{rlx}} 1; \{\mathsf{True}\}$	$\quad X := 57; \{X \mapsto 57\}$

Figure 6. Message-Passing with **rlx** accesses and fences.

As relaxed accesses do not synchronize, it would be unsafe to replace the **sc** accesses to Y in MP with **rlx** ones, since then the two non-atomic accesses to X would constitute a race. In order to understand why, we need to understand a bit of the relaxed-memory semantics.

Following [13] and [12], we opt for an operational explanation whose core notion is *views*. Intuitively, each thread in the program has its own *local view* which represents its local, subjective perspective of the state of memory. If thread 1 modifies the memory, it is not necessary that thread 2 observes that modification immediately. In the terminology of views, we say that thread 2's local view V_2 does not include the modification by thread 1. In order to observe the modification, thread 2 needs to perform physical synchronization with thread 1, so that thread 1's local view V_1 is incorporated or *joined* into V_2. Then, V_1 is *included* in V_2: $V_1 \sqsubseteq V_2$. After that, the thread 2 has observed the modifications by thread 1. The *view inclusion* relation implies synchronization or, in different terms, the *happens-before* relation. As threads execute, and their local views grow over time, they occasionally synchronize with one another by sending their local views to other threads.

Consider the MP example with **sc** accesses replaced by **rlx** accesses. In that hypothetical case, we are not guaranteed a happens-before relation between the non-atomic writes X of thread 1 and thread 2—leaving us with a race. In the language of views, this can be explained as follows: thread 1's local view V_1 after the write has not been joined into thread 2's local view V_2 when it starts writing X, so V_2 is not guaranteed to include the write in thread 1. In the operational semantics, performing a non-atomic write without having observed *all* writes constitutes a race. In order to avoid the race, we need sufficient synchronization, which we can achieve using release and acquire fences, as shown in Fig. 6.[4]

The synchronization is guaranteed physically through the chain of "release fence → relaxed write → relaxed read → acquire fence". That is, the synchronization is only between what is *before* the release fence and what is *after* the acquire fence. Since the non-atomic write to X is before the release fence, and the non-atomic read of X is after the acquire fence, they are happens-before and thus there is no race. If,

[4] One can also use a pair of release and acquire *accesses*, but here we use fences for the sake of the exposition.

however, we were to remove the fences or reorder them with the accesses to Y, we would once again have a race.

The explanation in terms of views is as follows. The **rlx** write of thread 1 to Y only sends to thread 2 the view V_{BF} that is thread 1's local view *before* the release fence, and the **rlx** read of thread 2 receives V_{BF} but only joins V_{BF} into thread 2's local view *after* the acquire fence. So only after the acquire fence would thread 2 have observed all writes to X, so that it may safely write to X non-atomically.

3.2 Separation Logic for Relaxed Memory

Several separation logics have been developed to reason modularly about relaxed-memory programs under variants of the C11 memory model. These logics—which include RSL [23], GPS [22], iGPS [12], and FSL [5, 6]—use views (or equivalent descriptions of a thread's local perspective) in their model of separation-logic assertions. Assertions thus become predicates not only on resources, but also on views. The reason for this may be illustrated by the points-to assertion of separation logic. If a thread owns $X \mapsto v$, it should be guaranteed (among other things) that a read from X will return v. In the relaxed-memory setting, ownership of $X \mapsto v$ must therefore say something about the local view V of the thread asserting it: V should contain the latest write to X, and it should have value v. Otherwise, reading from X could yield an older value and thus render at least one guarantee of the points-to assertion moot.

Furthermore, it is crucial that these predicates be *view-monotone*, *i.e.*, that assertions remain valid when the thread witnesses additional memory events. Formally, monotonicity means that if $[\![P]\!]$ represents the model of an assertion as a view predicate, and $V_1 \sqsubseteq V_2$, then $[\![P]\!](V_1)$ implies $[\![P]\!](V_2)$. This requirement stems from separation logic's "frame rule". Intuitively, a thread owning $(X \mapsto v) * P$ must be able to *frame* P around accesses to X—*i.e.*, retaining ownership of P throughout—*even though* such accesses will grow the thread's view. For this, the validity of P must be monotone in the thread's view.

To support **rlx** accesses, the model of assertions is built around a more fine-grained notion of a thread's view in which we distinguish between (1) the thread's current view, (2) its *release* view (*i.e.*, the view the thread had at the most recent **rel** fence), and (3) its *acquire* view (*i.e.*, the view the thread will have after the next **acq** fence).[5]

Based on this notion of views, we derive a separation logic for relaxed memory in Iris—called GPFSL—which brings together the essential features of iGPS [12] and FSL [5, 6]. The details of GPFSL are beyond the scope of this paper, but for now we focus our attention on the release and acquire modalities Δ and ∇ that GPFSL inherits from FSL.

[5]Views also include an additional component used to track per-location **rel** writes which essentially act as mini **rel** fences for that location.

$$\textsc{Rel-fence}\quad \{P\}\ \mathbf{fence_{rel}}\ \{\Delta P\} \qquad \textsc{Acq-fence}\quad \{\nabla P\}\ \mathbf{fence_{acq}}\ \{P\}$$

$$\textsc{Rlx-Send}\quad \{\mathrm{Send}_Y(P) * \Delta P\}\ Y :=_{\mathbf{rlx}} 1\ \{\mathrm{True}\}$$

$$\textsc{Rlx-Recv}\quad \{\mathrm{Recv}_Y(P)\}\ {}^{*\mathbf{rlx}}Y\ \{v.v = 0 \vee \nabla P\}$$

Figure 7. Rules for fences and SendRecv with **rlx**.

The logic for fences. The release modality ΔP asserts ownership of P at the thread's release view, thereby witnessing that P has been true for the current thread since (at least) its most recent **rel** fence. Consequently, ΔP allows P to be transferred through **rlx** writes as demonstrated by Rlx-Send (Fig. 7). The introduction rule Rel-fence moves propositions into the release modality when a **rel** fence occurs.

The acquire modality ∇P asserts ownership of P at the thread's acquire view, guaranteeing that P will hold for the current thread at (and after) its next **acq** fence. The intended use of the acquire modality is to record the effect of **rlx** reads, which constitute the modality's introduction rule as demonstrated by rule Rlx-Recv. Conversely, executing an acquire fence eliminates the modality (rule Acq-fence).[6]

With these rules for fences and relaxed SendRecv (Fig. 7), we can now make sense of the verification of relaxed MP in Fig. 6. The proof is very similar to the SC proof (Fig. 1), and one only needs to be careful about introducing the release modality and eliminating the acquire modality.

Ghost state and view-agnosticism. Following iGPS and FSL++, GPFSL provides support for *user-defined ghost state*. Ghost state is logical state that is used to track additional information in the verification, but is not part of physical state. In the aforementioned logics, the user of the logic is free to define the particular structure of ghost state that is appropriate for their proof, a facility that is useful in deriving domain-specific logics like the lifetime logic. The Send and Recv assertions, as well as the various abstract predicates that comprise the lifetime logic, are all instances of assertions that are constructed with the help of ghost state.

Although it has proven indispensable for verifying intricate concurrent data structures in traditional SC separation logics, ghost state has an additional "special power" in the relaxed-memory setting, namely that it is *view-agnostic*. In other words, because ghost state is purely logical, the ghost state assertion $\boxed{a}^{\gamma}$, which asserts the ownership of *ghost resource* a stored at *ghost location* γ, does not care about the thread view at which it is interpreted.

As a result, ownership of $\boxed{a}^{\gamma}$ can be transferred from one thread to another *without the need for physical synchronization*. In particular, not being tied to any view, ghost state can

[6]For more details on the logic, please consult [6] and our appendix.

soundly be moved in and out of the fence modalities:

$$\boxed{a}^{\gamma} \Leftrightarrow \Delta\boxed{a}^{\gamma} \Leftrightarrow \nabla\boxed{a}^{\gamma} \qquad \text{(FSL-Ghost-mod)}$$

Recall that ΔP asserts that P holds at the thread's release view, and ∇P asserts that P holds at the thread's acquire view. Since ownership of ghost state holds regardless of any view, it is easy to see that owning $\boxed{a}^{\gamma}$ is equivalent to owning $\Delta\boxed{a}^{\gamma}$ or $\nabla\boxed{a}^{\gamma}$.

FSL-Ghost-mod is essential in enabling threads to agree on the *global* state of a data structure. Regardless of what each thread has observed about the data structure, it is useful to be able to express globally consistent properties of the data structure. For example, we may want to record the history of all `push`'s and `pop`'s of a concurrent stack, and require that the number of `pop`'s cannot surpass the number of `push`'s. Ghost state is able to encode such objective, view-agnostic information, and in their paper on FSL++, Doko and Vafeiadis [6] noted the importance of ghost state being view-agnostic, writing: "The most important feature of ghost state from the perspective of the verification of `Arc` is [the] ability to transfer ownership of ghosts without the need for synchronization. This is achieved by having the ghost state be agnostic with respect to the Δ and ∇ modalities."[7]

While ghost state in general clearly benefits from being view-agnostic, we will see in the next section that sometimes it is useful for ghost state assertions to *not* be view-agnostic. This leads us to propose the apparently novel concept of *view-dependent ghost state*.

4 Lifetime Logic Meets Relaxed Memory

Now that we have reviewed the basics of the lifetime logic and relaxed memory, we turn our attention to the problem of combining the two and present our solution to it. In RustBelt, lifetime tokens are simply ghost state that are not tied to physical state. This model suffices for the soundness of RustBelt's lifetime logic under the SC assumption. However, in the relaxed-memory setting, modeling lifetime tokens as view-agnostic ghost state is problematic because then lifetime tokens can be transferred across threads without physical synchronization. In §4.1, we show a counterexample where this naive model of lifetime tokens leads to unsound reasoning under relaxed memory. In §4.2, we discuss how to remedy the problem by modeling lifetime tokens instead using view-dependent ghost state, and in §4.3, we explain how to prove soundness of full borrows under this more sophisticated model.

4.1 Deriving Unsoundness in a Naive Model

If in the relaxed-memory setting we continue to model lifetime tokens as view-agnostic ghost state, then by using the FSL-Ghost-mod rule we can provide a *spurious* verification of the buggy MP example given in Fig. 8.

The initialization is similar to the verification in Fig. 5: We create a lifetime κ and a borrow for X, and instantiate SendRecv for Y before giving them to the two threads. In thread 1 (Fig. 8b), we access the borrow and write to X. Then, to send $[\kappa]_{1/2}$ (via a **`rlx`** write to Y), we use FSL-Ghost-mod to obtain $\Delta[\kappa]_{1/2}$. *Note that this proof step is only possible because we assume view-agnostic lifetime tokens.*

In thread 2 (Fig. 8c), after receiving $\nabla[\kappa]_{1/2}$, we apply FSL-Ghost-mod again to strip off the acquire modality, thus obtaining the missing half of the token. Combining both halves, we kill κ and apply the inheritance to obtain $X \mapsto -$. This, in turn, licenses the following non-atomic write to X, which is *not* happens-after thread 1's write to X and thus constitutes a data race.

As we can see from this scenario, our hypothetical lifetime logic for relaxed memory violates a key safety guarantee: that a lifetime κ's inheritance must happen-after all accesses to all borrows of κ (see §2). The root of the problem is that we are able to move view-agnostic lifetime tokens in and out of the fence modalities.

We now leave our hypothetical unsound lifetime logic and turn towards $\mathbf{RB_{rlx}}$'s lifetime logic which, being based on *view-dependent* lifetime tokens, is sound in the presence of relaxed memory.

4.2 Lifetime Tokens as View-Dependent Ghost State

If we were to port RustBelt's model of the lifetime token $[\kappa]_q$ directly to $\mathbf{RB_{rlx}}$ as a view-agnostic assertion, it would just assert ghost ownership of q with the ghost location κ:

$$[\![[\kappa]_q]\!](V) ::= \boxed{q}^{\kappa} \qquad \text{(RB-tok)}$$

Instead, in $\mathbf{RB_{rlx}}$, we enrich this model so that it depends on the view at which $[\kappa]_q$ is asserted:

$$[\![[\kappa]_q]\!](V) ::= \exists V_{\text{tok}}.\, \boxed{(q, V_{\text{tok}})}^{\kappa} * V_{\text{tok}} \sqsubseteq V \qquad \text{(Rlx-RB-tok)}$$

In this model, the ghost element is no longer just a fraction q, but a pair of the fraction and the *token view* V_{tok}. The token view V_{tok} represents what this particular fraction of the token has *observed*, *i.e.*, what borrow accesses the token has participated in. The model requires that V—the view at which the token is interpreted—has also at least observed what $[\kappa]_q$ has observed: $V_{\text{tok}} \sqsubseteq V$.

To understand the implications of this change, we need to understand what it means for a thread to own $[\kappa]_q$ in the model. As explained in §3.2, assertions in our logic are interpreted as view predicates in Iris. That a thread π owns $[\kappa]_q$ is interpreted in the model as the thread π owning $[\kappa]_q$ at its local view V_π, *i.e.*, $[\![[\kappa]_q]\!](V_\pi)$. In the model with Rlx-RB-tok, this gives us $V_{\text{tok}} \sqsubseteq V_\pi$. Therefore, owning the lifetime token $[\kappa]_q$ implies that the thread has observed all accesses that the token has been involved in (as encoded in V_{tok}).

[7] Indeed, we too make crucial use of the view-agnostic nature of ghost state in our own (more comprehensive) verification of `Arc` in $\mathbf{RB_{rlx}}$. See §7 for a more detailed comparison between our `Arc` proof and FSL++'s.

$X := 0; Y := 0;$
$X := 42; \;\|\; \texttt{if}\ {}^{*\texttt{rlx}}Y \mathrel{!}= 0$
$Y :=_{\texttt{rlx}} 1 \;\|\; \texttt{then}\ X := 57;$

(a) Buggy Message-Passing.

$\{[\kappa]_{1/2} * \&^{\kappa}_{\text{full}}(X \mapsto -) * \mathsf{Send}_Y([\kappa]_{1/2})\}$
$X := 42; \{[\kappa]_{1/2} * \mathsf{Send}_Y([\kappa]_{1/2})\}$
$\{\Delta[\kappa]_{1/2} * \mathsf{Send}_Y([\kappa]_{1/2})\}$ *Unsound!*
$Y :=_{\texttt{rlx}} 1; \{\text{True}\}$

(b) Buggy proof of thread 1.

$\{[\kappa]_{1/2} * \mathsf{Kill}(\kappa) * \mathsf{Inh}(\kappa, X \mapsto -) * \mathsf{Recv}_Y([\kappa]_{1/2})\}$
$\texttt{if}\,({}^{*\texttt{rlx}}Y \mathrel{!}= 0)$
$\{[\kappa]_{1/2} * \ldots * \nabla[\kappa]_{1/2}\}$
$\{[\kappa]_{1/2} * \mathsf{Kill}(\kappa) * \ldots * [\kappa]_{1/2}\}$ *Unsound!*
$\{[\dagger\kappa] * \mathsf{Inh}(\kappa, X \mapsto -)\}\ \{X \mapsto -\}\ X := 57; \{X \mapsto 57\}$

(c) Buggy proof of thread 2.

Figure 8. Buggy MP spuriously verified with view-agnostic lifetime tokens.

As a result of this change, FSL-Ghost-mod no longer applies to lifetime tokens, so transferring lifetime tokens in $\mathbf{RB}_{\texttt{rlx}}$ requires physical synchronization. This suffices to eliminate the hypothetical unsound verification in Fig. 8. The question remains, however: How do we know that we are back on solid ground? How can we prove that the proof rules for full borrows are sound under relaxed memory?

As we will see shortly, the key is to associate views not only with the lifetime token assertions, but with all the other assertions that play a role in the lifetime logic. In so doing, we can express invariants that govern the *view-dependent interactions* between those assertions. In the next subsection we will see how such invariants enable the porting of full borrows to $\mathbf{RB}_{\texttt{rlx}}$ without changing their interface.

4.3 Full Borrows

Recall that there are two ways of interacting with the content of a full borrow: (1) reclaiming the borrow using the inheritance (LftL-full-inh) and (2) accessing the borrow (LftL-full-acc). We now explain how to prove soundness of these two different forms of borrow interactions.

Proving inheritance. To prove a rule sound in the model of $\mathbf{RB}_{\texttt{rlx}}$, we first interpret the rule at the current local view of the thread that applies the rule and then prove it in Iris. In particular, to prove LftL-full-inh sound for $\mathbf{RB}_{\texttt{rlx}}$, we first interpret it at the local view V of the thread:

$$[\![[\dagger\kappa]]\!](V) * [\![\mathsf{Inh}(\kappa, P)]\!](V) \Rrightarrow\!\!\!\!\not\ast\ [\![P]\!](V)$$

The user of this rule provides us—the prover of the rule—with the dead token and the inheritance at V, and we need to give the user back P at the same V. Now, the lifetime logic is responsible for controlling ownership of the content of the borrow, P; so let us assume that, when no threads are accessing the borrow, P is maintained at a view V_C, which we call the *content view*. When we apply the inheritance, we will therefore be receiving $[\![P]\!](V_C)$. Since all assertions of GPFSL are view-monotone, if we can show that $V_C \sqsubseteq V$, we can *upgrade* P from V_C to V and obtain $[\![P]\!](V)$ to finish the proof.

The key to proving the goal $V_C \sqsubseteq V$ is to:

- associate with each lifetime logic assertion a view that represents what the assertion has observed, *i.e.*, what activities it has been involved in; and then
- establish and maintain invariants on those associated views that are sufficiently strong to prove our goal.

Below, we list a few associated views for assertions that interact with a full borrow $\&^{\kappa}_{\text{full}} P$:

- the *content view* V_C at which the borrow keeps P;
- the *token views* V_{tok}, one for every $[\kappa]_q$;
- the *full token view* V_κ of the full token $[\kappa]_1$, defined as the join of all token views V_{tok};
- the *dead token view* $V_\dagger \sqsupseteq V_\kappa$;
- the *borrow view* V_B of the borrow assertion $\&^{\kappa}_{\text{full}} P$;
- the *inheritance view* V_I of $\mathsf{Inh}(\kappa, P)$; and
- the *initial view* V_0, which is the content view of P when the borrow is first created.

In order to prove $V_C \sqsubseteq V$ for LftL-full-inh, we enforce the following invariant:

$$V_0 \sqsubseteq V_I \wedge \left(V_0 \neq V_C \Rightarrow V_C \sqsubseteq V_\kappa\right) \qquad \text{(LftL-full-bor-inv-1)}$$

The first part of the invariant, $V_0 \sqsubseteq V_I$, is needed for the corner case where the borrow is created but never used. In that case, P holds at $V_C = V_0$, the initial view. When inheritance happens, by virtue of owning $[\![\mathsf{Inh}(\kappa, P)]\!](V)$, we know from the definition of $\mathsf{Inh}(\kappa, P)$ that $V \sqsupseteq V_I \sqsupseteq V_0 = V_C$.

The second part of the invariant is for full borrows that have been accessed at least once. By owning $[\![[\dagger\kappa]]\!](V)$, we know that $V \sqsupseteq V_\dagger \sqsupseteq V_\kappa \sqsupseteq V_C$. □

This shows that LftL-full-bor-inv-1 allows us to prove LftL-full-inh. But what is the intuition for this invariant? As hinted at before, each piece of a lifetime token is intended to bear "witness" to any access to the borrow that the token is used for. Ultimately, the full token $[\kappa]_1$—which is the join of all tokens—must have witnessed *all* accesses to the borrow. Therefore, a thread owning the full token should have observed all modifications made to P by all of those accesses, so it can safely kill the lifetime and use the inheritance to reclaim P at its local view.

How do we maintain LftL-full-bor-inv-1? The first part, $V_0 \sqsubseteq V_I$, is easily maintained by the creation of the borrow with LftL-full-bor. The second part, $V_C \sqsubseteq V_\kappa$, is maintained by the rule LftL-full-ret. Note that the lifetime token $[\kappa]_q$ is withheld during the access. When the access finishes and P is returned to the borrow at an updated view V'_C, the rule uses V'_C to update the view of the withheld token $[\kappa]_q$ from V_{tok} to $V_{\text{tok}} \sqcup V'_C$ before returning it to the user. Since V_κ is

the join of all lifetime tokens, this effectively updates V_κ to $V_\kappa \sqcup V'_C \sqsupseteq V'_C$. With this, the invariant is re-established.

Proving accesses. To prove the rule for accessing full borrows, LftL-full-acc, we again interpret it in the model:

$$[\![\&^\kappa_{\text{full}} P]\!](V) * [\![[\kappa]_q]\!](V) \Rrightarrow [\![P]\!](V) * [\![\mathsf{Ret}(\kappa, P, q)]\!](V)$$

Given the lifetime token and the borrow assertion, we need to provide the user with synchronized access to P at the thread's local view V. Assuming that we can prove the return policy $\mathsf{Ret}(\kappa, P, q)$, we still need to ensure that P *holds* at V, *i.e.*, $V \sqsupseteq V_C$. For this, we rely on the following invariant:

$$V_C \sqsubseteq V_B \qquad \text{(LftL-full-bor-inv-2)}$$

That is, the content view V_C of P is always included in the borrow view V_B of $\&^\kappa_{\text{full}} P$. By owning $[\![\&^\kappa_{\text{full}} P]\!](V)$, we have $V \sqsupseteq V_B \sqsupseteq V_C$. Like LftL-full-bor-inv-1, LftL-full-bor-inv-2 is straightforward to maintain: When an access is returned, we update V_B with the new content view V'_C. □

In summary, associating lifetime logic assertions to views allows us to express and enforce strong invariants on interactions with lifetimes and borrows, so that we can port their original RustBelt rules (some of which are listed in Fig. 3) to $\mathbf{RB_{rlx}}$ without any changes to their interface.

5 Fractured and Atomic Borrows

Full borrows are perfect for modeling mutable references that can only have one user at a time. This is because accesses to full borrows are always sequential: at any moment in time, there can be only one ongoing access to a full borrow. For this reason, however, full borrows are not suitable for modeling types that are meant to be accessed concurrently by multiple threads, *e.g.*, shared references. In this section, we discuss $\mathbf{RB_{rlx}}$ ports of two alternatives to full borrows: *fractured* borrows and *atomic* borrows.

Table 1 compares several properties of different borrow types. These differences already exist in RustBelt except for the last column, which is unique to the relaxed-memory setting. (We will come back to that column in §5.2.) Both fractured and atomic borrows are created by conversion from full borrows (LftL-fract-bor and LftL-at-bor, Fig. 9). Once created, they both allow the borrowed contents to be shared concurrently by several parties. Their borrow assertions, $\&^\kappa_{\text{frac}} \Phi$ and $\&^\kappa_{\text{at}} P$, are freely duplicable so that the same borrow can be referred to and be accessed simultaneously by multiple threads. This is in contrast to the full borrow assertion $\&^\kappa_{\text{full}} P$, which is unique and is withheld during an access so as to ensure at most one access at a time.

But how is it possible, one may ask, that multiple threads can acquire the same resource at the same time? Fractured borrows and atomic borrows give different answers to this question. Fractured borrows only give out a *a fraction* of the resource—ensuring that enough fractions remain for all participants at all times—whereas atomic borrows enforce a strict turn-taking scheme, allowing access to the resource only for a single atomic step of execution.

LftL-fract-bor
$$\&^\kappa_{\text{full}} \Phi(1) \Rrightarrow \&^\kappa_{\text{frac}} \Phi$$

LftL-at-bor
$$\&^\kappa_{\text{full}} P \Rrightarrow \&^\kappa_{\text{at}} P$$

LftL-fract-acc
$$\&^\kappa_{\text{frac}} \Phi * [\kappa]_q \Rrightarrow \exists q'.\, \Phi(q') * \mathsf{Ret}(\kappa, \Phi, q, q')$$

LftL-fract-ret
$$\Phi(q') * \mathsf{Ret}(\kappa, \Phi, q, q') \Rrightarrow [\kappa]_q$$

LftL-at-acc
$$\frac{\{P * Q_1\}\, e\, \{v.\, P * Q_2\} \qquad \mathsf{atomic}(e)}{\&^\kappa_{\text{at}} P \vdash \{[\kappa]_q * Q_1\}\, e\, \{v.\, [\kappa]_q * Q_2\}}$$

Figure 9. Selected RustBelt rules for fractured and atomic borrows.

To meaningfully talk about fractions of resources, fractured borrows assume a *predicate* Φ over fractions that is compatible with fraction addition: $\Phi(q_1+q_2) \Leftrightarrow \Phi(q_1) * \Phi(q_2)$. The access rule LftL-fract-bor (Fig. 9) gives access to $\Phi(q')$ for some fraction q'. The corresponding return policy only returns the lifetime token once the exact q' of Φ is returned (LftL-fract-ret).

In contrast to fractured borrows, atomic borrows do provide full access to the resources contained within. This is possible by restricting the duration of the access: unlike fractured (and full) borrows, atomic borrows can only be accessed around a single, atomic instruction.[8] This restriction of atomic borrows is encoded in its access rule LftL-at-acc (Fig. 9), which allows accesses only around Hoare triples for atomic instructions.

Atomicity is crucial, for example, when several threads need to modify a shared variable, such as a reference counter for shared pointers. Therefore, while fractured borrows are designed to model *immutable* shared resources, atomic borrows are designed to model *mutable* shared resources in concurrent libraries. (Naturally, these libraries use atomic memory accesses compatible with LftL-at-acc.)

In the remainder of this section, we briefly discuss the porting of fractured and atomic borrows to $\mathbf{RB_{rlx}}$.

5.1 Porting Fractured Borrows

Fractured borrows, like all borrows, need to guarantee that all accesses happen-before the inheritance is applied. However, unlike full borrows where all accesses are ordered, accesses to a fractured borrow can happen independently from one another. Thus, for fractured borrows, we need to maintain that independent changes to fractions of Φ made by independent accesses are all observed by the thread performing the inheritance. The key to achieve this is to recognize that the

[8] In Iris parlance, atomic instructions are expressions that evaluate in just one step. This is not to be confused with the notion of atomic accesses in this paper, even though atomic accesses are indeed atomic instructions.

Borrow type	Access by multi-threads	Access duration	Access amount	Communication between accesses	Access at local view
Full borrows $\&^{\kappa}_{\mathbf{full}}\, P$	sequential	multiple steps	full	yes	yes
Fractured borrows $\&^{\kappa}_{\mathbf{frac}}\, \Phi$	concurrent	multiple steps	fractions	no	yes
Atomic borrows $\&^{\kappa}_{\mathbf{at}}\, P$	concurrent	atomic	full	yes	no

Table 1. Comparison of borrow types.

content view of Φ is no longer a single view, but consists of two views:

(1) the view V_{YTBA} of the *yet-to-be-accessed* portion of Φ,
(2) the view V_{AA} of the *already-accessed* portion of Φ.

V_{YTBA} is the view of the chunk of Φ that has not been given out to any access, as well as the view at which the fractured borrow was created. Meanwhile, V_{AA} tracks all the changes made by the accesses to all the chunks that have been given out to those accesses.

With that in mind, we repeat what we did for full borrows: defining the invariants for the inheritance and the accesses of fractured borrows.

Proving inheritance. We enforce an invariant on the two views of a fractured borrow $\&^{\kappa}_{\mathbf{frac}}\, \Phi$:

$$V_{\text{YTBA}} \sqcup V_{\text{AA}} \sqsubseteq V_{\kappa} \qquad \text{(LftL-fract-bor-inv-1)}$$

That is, instead of using a single content view V_C like in LftL-full-bor-inv-1, we use the view $V_{\text{YTBA}} \sqcup V_{\text{AA}}$ which is the view of the full resource $\Phi(1)$, and require that it is always included in the full token view κ. Thus, by similar reasoning to LftL-full-bor-inv-1, any changes to the fractured borrow's contents are guaranteed to happen-before the moment the inheritance is applied.

Proving accesses. We maintain a second invariant:

$$V_{\text{YTBA}} \sqsubseteq V_B \qquad \text{(LftL-fract-bor-inv-2)}$$

This invariant enables the synchronized access to a fraction of Φ in the accessing rule LftL-fract-acc: If the rule is applied at a thread's local view V and the thread owns a borrow assertion $\&^{\kappa}_{\mathbf{frac}}\, \Phi$ with some borrow view V_B, then $V \sqsupseteq V_B \sqsupseteq V_{\text{YTBA}}$. Thus the thread can obtain some portion $\Phi(q')$ from the the yet-to-be-accessed chunk at its local view V.

5.2 Porting Atomic Borrows

A challenge specific to porting atomic borrows arises from the fact that atomic borrows, like full borrows, allow communication between different accesses to the same borrow. This situation does not apply to fractured borrows because each access of a fractured borrow obtains an independent fraction of the underlying contents. In contrast, each access of an atomic borrow obtains the full contents and can modify them and thus can communicate with other accesses.

The trouble with communication between accesses is that atomic instructions—in particular atomic accesses around which atomic borrows are typically accessed—are not always synchronizing (*i.e.*, they do not imply happens-before). Therefore, LftL-at-acc (Fig. 9) is *not* sound in $\mathbf{RB_{rlx}}$. Imagine that if LftL-at-acc were sound, a thread π would access and modify P (*i.e.*, after the access P holds at π's local view), and immediately in the next step a different thread ρ would get synchronized access to P at its own local view, including the modifications made by π. In that case, the logic would allow synchronization from π to ρ without there being any physical synchronization to support that. In particular, we would be able to use atomic borrows to derive a version of Rlx-Send and Rlx-Recv (Fig. 7) *without* the Δ and ∇ modalities. With such rules allowing us to bypass the need for fences entirely, we could construct yet another spurious verification of the buggy MP example in Fig. 8.

Since the content view V_C of P can be constantly changed by different threads with atomic accesses to the borrow, there is in general no relationship between threads' local views and the content view V_C. Therefore, we need a new accessing rule for atomic borrows that does not equate the view at which the content is provided with the local view of the thread. In order to state the new rule, we add a new assertion to the logic, which we call the *view-join modality*: If P is a view-dependent assertion and V_b a view, then the view-join modality $\lfloor P \rfloor_{\sqcup V_b}$ is defined by $[\![\lfloor P \rfloor_{\sqcup V_b}]\!](V) ::= [\![P]\!](V \sqcup V_b)$, which means that P holds at the join of the current view and V_b. With that defined, here is the new atomic borrow accessing rule:

$$\frac{\text{Rlx-LftL-at-acc}\quad \forall V_b.\, \{\lfloor P \rfloor_{\sqcup V_b} * Q_1\}\, e\, \{v.\, \lfloor P \rfloor_{\sqcup V_b} * Q_2\} \qquad \text{atomic}(e)}{\&^{\kappa}_{\mathbf{at}}\, P \vdash \{[\kappa]_q * Q_1\}\, e\, \{v.\, [\kappa]_q * Q_2\}}$$

That is, after opening the atomic borrow at the current thread's view V, a thread obtains the content P at an arbitrarily larger view $V \sqcup V_b$. Here, V_b is a parameter that will be instantiated (by the prover of the rule) with the content view V_C. After obtaining access to P, the atomic instruction e can update the current thread's view to a larger view V'. After e has executed, the atomic borrow is closed by giving back P at the updated view $V' \sqcup V_b$.

Why must P be returned at $V' \sqcup V_b$? The answer lies in the inheritance for atomic borrows. To maintain the guarantee that the inheritance happens-after all accesses, when returning the borrow contents we upgrade and then fix the content view V_C to *equal* the lifetime view V_κ.[9] Equating these two views trivially maintains the inheritance invariant that $V_C \sqsubseteq V_\kappa$. To maintain the view equality, however, we now must mirror any change to V_κ in V_C. In particular, returning the borrow with RLX-LFTL-AT-ACC will update the lifetime token view V_{tok} (and therefore V_κ) by joining the thread's local view V' into it. This change thus needs to be mirrored in V_C, which explains why we demand P at $V' \sqcup V_b$ (thus upgrading V_C to $V' \sqcup V_C$).

Finally, note that for atomic borrows there is in general no relation between V_C and the borrow view V_B, so there is no second invariant (like LFTL-FULL-BOR-INV-2) to maintain.

6 Other Contributions of $\text{RB}_{\texttt{rlx}}$

ORC11 memory model. Rust inherits its memory model from C11 [2, 21]. The model suffers from a long-standing issue with *out-of-thin-air* reads, which pose a significant obstacle to formal verification [3]. Thus, following previous relaxed-memory separation logics, we work with a strengthened variant of C11 in which out-of-thin-air behaviors are eliminated by prohibiting load-store reordering on relaxed accesses. In particular, we target RC11 [16], which fixes this and other flaws of the model. Recent work by Ou and Demsky [18] suggests that the performance overhead of working with RC11 vs. C11 may not be so significant in practice.

However, as mentioned in the introduction, we require an operational account of relaxed memory—instead of an axiomatic one such as RC11—in order to instantiate the Iris framework. To this end, we develop ORC11, a semantics inspired by both the promising semantics [13] and iGPS [12]. The main novelty lies in ORC11's race detector, which extends iGPS's in order to account for the non-synchronizing nature of relaxed accesses. It does so by introducing explicit atomic/non-atomic read/write events into the operational semantics and tracking which events threads have seen.

To validate ORC11, we sketch (in the appendix) a proof of correspondence between ORC11 and the fragment of RC11 omitting SC accesses and SC fences (since those features are not used by the Rust libraries considered by RustBelt). The correspondence establishes that any program considered racy by RC11 will also be considered racy by ORC11.

GPFSL program logic. As part of developing $\text{RB}_{\texttt{rlx}}$, as mentioned in §3.2, we used Iris to derive a new relaxed-memory separation logic GPFSL, which combines the functionality of iGPS/GPS [12, 22] and FSL/FSL++ [5, 6]. In particular, it inherits fence modalities from FSL and improves on iGPS by supporting the reclamation of resources governed by iGPS's single-location protocols.

Proving soundness of λ_{Rust} type system and libraries. A major component of the original RustBelt verification was the proof of semantic soundness for the typing rules of the λ_{Rust} type system, which was performed in Iris-SC. Fortunately, we were able to reuse these proofs unchanged! One may wonder how this is possible given that $\text{RB}_{\texttt{rlx}}$ is built on top of GPFSL rather than Iris-SC. The answer is that both of them are implemented in Coq as instantiations of the MoSeL framework [15], which provides a uniform set of tactics applicable to different separation logics. Since the only rules on which Iris-SC and GPFSL differ are those pertaining to atomic accesses and atomic borrows, and since the safe fragment of λ_{Rust} does not concern itself with atomics, we could port the proof of semantic soundness of λ_{Rust} from Iris-SC to GPFSL without modification.

In contrast, since the concurrency libraries considered in RustBelt use relaxed-memory operations, we had to verify them afresh. They include: `thread::spawn`, `rayon::join`, `Mutex`, `RwLock`, and `Arc`. (The verifications of the sequential libraries `Rc`, `Cell`, and `RefCell` remain largely unchanged from RustBelt.) By far the most challenging of these to verify in $\text{RB}_{\texttt{rlx}}$ was `Arc`. To make the verification go through, we needed to strengthen two atomic reads from **`rlx`** to **`acq`** in the implementations of `Arc::get_mut` and `Arc::make_mut`. We conjecture that the relaxed access in `Arc::make_mut` is indeed sound but verifying this would have required a significantly more complex invariant. We refer the reader to the appendix for further details.

Data race in `Arc`. Our need to strengthen the atomic access in `Arc::get_mut` (for purposes of verification) led us to uncover it as a bug in the original implementation of the `get_mut` function. The original **`rlx`** read was insufficient to establish a happens-before relation between the dropping of the second-to-last clone of an `Arc`—a **`rel`** write—and subsequent exclusive access to the contents of the `Arc` through a call to `get_mut` on the sole remaining clone. This violation of happens-before can be used to construct a data race under the C11 memory model. The bug has been reported and is already fixed in Rust.

7 Related and Future Work

iGPS [12] has a mechanism to reclaim resources called *cancellable* single-location protocols. Cancellable protocols are, in fact, very similar to borrows in that they use fractional tokens to prove that the protocol is not cancelled, *i.e.*, still alive, and can still be accessed. Anyone owning the full token can cancel the protocol and directly trade the token for the protocol resources. Therefore, these protocols can be seen as a direct-style variant of borrows without the lifetime killing and inheritance business. iGPS cancellable protocols

[9] This only happens for borrows that are accessed at least once. Otherwise, V_C is included only in the inheritance view V_I as explained in §4.3, which still suffices to prove the inheritance sound.

are not as powerful as $\mathbf{RB_{rlx}}$ lifetime logic in that they cannot reclaim protocol resources at the thread's local view, simply because the fraction tokens used by them are also *view-agnostic*. Inspired by the lifetime logic, GPFSL generalizes cancellable protocols to also be *view-dependent* in the same style that lifetime tokens are made view-dependent, thus allowing protocol resource reclamation by either atomic borrows-based protocols or view-dependent cancellable protocols. We depend heavily on the latter in verifying `Arc`.

Doko and Vafeiadis [6] verify a subset of the `Arc` library. We improve on their results by (1) enlarging the scope of the verification to include important parts of the API such as the `make_mut` and `get_mut` functions (the latter of which we found to contain a data race) as well as the `Weak` reference type, (2) allowing full resource reclamation of both the contents and the reference-count fields of the `Arc` data structure, and (3) embedding our verification effort in the RustBelt framework, so that we can establish the soundness of `Arc` when linked with unknown well-typed λ_{Rust} code.

Kang et al.'s *promising semantics* [13] is a proposal for fixing C11's out-of-thin-air problem without prohibiting load-store reordering on relaxed accesses (as RC11 and ORC11 do). Svendsen et al. [20] introduce the first program logic for the promising semantics. Their logic is based on RSL [23] and supports relaxed accesses but not fences. Moreover, unlike FSL, it disallows the transfer of ownership through relaxed accesses, among other reasoning principles that have proven useful in $\mathbf{RB_{rlx}}$. Extending $\mathbf{RB_{rlx}}$ to account for promises is a very interesting avenue for future work.

Finally, it is worth re-iterating that ORC11 and GPFSL currently do not support SC accesses and SC fences. Adding support for SC would enable us to verify some interesting and challenging fine-grained concurrent algorithms, such as the work-stealing queue by Chase and Lev [4], as well as epoch-based resource reclamation schemes such as that implemented by Rust's `crossbeam` library [1].

Acknowledgments

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289).

References

[1] 2016. Crossbeam: Support for concurrent and parallel programming. Available at https://github.com/aturon/crossbeam.

[2] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. 55–66.

[3] Hans-J. Boehm and Brian Demsky. 2014. Outlawing ghosts: Avoiding out-of-thin-air results. In *MSPC*.

[4] David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*. 21–28. https://doi.org/10.1145/1073970.1073974

[5] Marko Doko and Viktor Vafeiadis. 2016. A program logic for C11 memory fences. In *VMCAI (LNCS)*. Springer, 413–430.

[6] Marko Doko and Viktor Vafeiadis. 2017. Tackling real-life relaxed concurrency with FSL++. In *ESOP*.

[7] Derek Dreyer. 2016. RustBelt project webpage. http://plv.mpi-sws.org/rustbelt/

[8] Derek Dreyer. 2018. Milner award lecture: The type soundness theorem that you really want to prove (and now you can). In *POPL*. https://www.youtube.com/watch?v=8Xyk_dGcAwk

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

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

[11] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. (2018). To appear in Journal of Functional Programming.

[12] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *ECOOP (LIPIcs)*. 17:1–17:29.

[13] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL*. ACM, 175–189.

[14] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. https://doc.rust-lang.org/stable/book/2018-edition/

[15] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP, Article 77 (2018).

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

[17] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.

[18] Peizhao Ou and Brian Demsky. 2018. Towards understanding the costs of avoiding out-of-thin-air results. *PACMPL* 2, OOPSLA, Article 136 (2018).

[19] John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *LICS*. https://doi.org/10.1109/LICS.2002.1029817

[20] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A separation logic for a promising semantics. In *ESOP 2018, Thessaloniki, Greece, April 14-20, 2018*. 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

[21] The Rust Team. 2018. *The Rustonomicon*. https://doc.rust-lang.org/stable/nomicon/atomics.html

[22] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*. ACM, 691–707.

[23] Viktor Vafeaidis and Chinmay Narayan. 2013. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*.

[24] Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation* 115, 1 (1994). https://doi.org/10.1006/inco.1994.1093