

RustBelt Meets Relaxed Memory

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, Derek Dreyer

▶ **To cite this version:**

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, Derek Dreyer. RustBelt Meets Relaxed Memory. POPL, Jan 2020, New Orleans, United States. 10.1145/3371101 . hal-02351793

HAL Id: hal-02351793
https://hal.science/hal-02351793

Submitted on 13 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

RustBelt Meets Relaxed Memory

HOANG-HAI DANG, MPI-SWS, Germany
JACQUES-HENRI JOURDAN, Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, France
JAN-OLIVER KAISER, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany

The Rust programming language supports 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 on the most interesting technical problem: how to reason about *resource reclamation under relaxed memory*, using a logical construction we call *synchronized ghost state*.

CCS Concepts: • **Theory of computation → Separation logic**; *Operational semantics*; Programming logic.

Additional Key Words and Phrases: Rust, semantic soundness, relaxed memory models, Iris

ACM Reference Format:
Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. . RustBelt Meets Relaxed Memory. 1, 1 (November), 29 pages.

1 INTRODUCTION

Rust [Klabnik and Nichols 2018] is a young and evolving 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 more insidious forms of anomalous behavior, such as data races and iterator invalidation, that safe languages typically fail to rule out. 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 reclaim 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 C/C++ and Java.

Only recently has Rust begun to receive attention from the programming languages research community. Notably, the RustBelt project [Dreyer 2016] has sought to set the safety claims of Rust on a more rigorous formal foundation. The initial work on RustBelt by Jung et al. [2018a] made two main contributions. First, Jung et al. 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

Authors' addresses: Hoang-Hai Dang, MPI-SWS, Saarland Informatics Campus, Germany, haidang@mpi-sws.org; Jacques-Henri Jourdan, Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, 91405, Orsay, France, jacques-henri.jourdan@lri.fr; Jan-Oliver Kaiser, MPI-SWS, Saarland Informatics Campus, Germany, janno@mpi-sws.org; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.

. XXXX-XXXX//11-ART
https://doi.org/

Coq proof assistant to verify formally that Rust's aforementioned safety guarantees do in fact hold, both for the core λ_{Rust} calculus and for a number of widely-used Rust libraries.

However, the initial work on RustBelt also made a significant simplifying assumption: it assumed a *sequentially consistent* (SC) model for concurrent memory accesses. On the one hand, sequential consistency [Lamport 1979]—*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 memory model* (hereafter, RMM), supporting a variety of different consistency levels for shared-memory accesses [Batty et al. 2011]. For programmers who demand strong synchronization, 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 *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 $\mathbf{RB_{rlx}}$, for short), the first formal validation of the soundness of Rust under relaxed memory. Although based closely on the original RustBelt, $\mathbf{RB_{rlx}}$ 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 [Jourdan 2018].

1.1 Relaxing RustBelt: Overview and Key Challenge

The overarching challenge in developing $\mathbf{RB_{rlx}}$ is that the logical foundation on which the original RustBelt is built is unsound for relaxed memory. To understand why, let us first review a bit about Rust and the structure of the RustBelt verification.

Background on Rust and RustBelt. 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.*, unchecked type casts, array accesses without bounds checks, or accesses of "raw" pointers whose aliasing is untracked by the type system). These libraries are *claimed* to be safe extensions to Rust because they encapsulate their uses of unsafe features 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 follows prior work on Foundational Proof-Carrying Code [Ahmed et al. 2010] by employing a *semantic soundness* proof. First, it defines a *semantic model* of Rust 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 features). Then, the RustBelt proof breaks into two main parts:

(1) *Safety of libraries that use unsafe features:* For any library that makes use of unsafe features, the implementation of the library is proven to satisfy the semantic model of its API, thus establishing that it is safe for clients to make use of the library. RustBelt proved safety for a number of widely-used Rust libraries, including `Arc`, `Rc`, `Cell`, `RefCell`, `Mutex`, and `RwLock`.
(2) *Safety of the λ_{Rust} type system:* The syntactic typing rules of λ_{Rust} are proven to respect the semantic model, thus establishing that code written in the "safe" fragment of Rust is in fact observably safe—*i.e.*, its behavior is well-defined.

Put together, these imply that if a program P is well-typed, and its only uses of unsafe features appear within the libraries that have been verified safe (in part 1), then P is observably safe.

In carrying out their semantic soundness proof for RustBelt, Jung et al. relied on *separation logic* [Reynolds 2002]. Separation logic is a good fit for modeling Rust because it is designed around the same notion of *ownership* as Rust's type system, and thus provides built-in support for ownership-based reasoning. More specifically, RustBelt was formalized in a higher-order concurrent separation logic framework called *Iris* [Jung et al. 2018b]. One benefit of using Iris is that it was designed to support the derivation of new separation logics with domain-specific reasoning principles. Jung et al. exploited this facility to derive a new logic called the *lifetime logic*, which they used extensively in their proofs in order to reason about Rust's "lifetimes" and "borrowing" mechanisms at a higher level of abstraction [Klabnik and Nichols 2018, §4.2, §10.3]. A second benefit of using Iris is that it comes with tactical support for developing *machine-checked* proofs interactively in Coq [Krebbers et al. 2017]; this support made it possible for RustBelt to be fully mechanized in Coq.

Separation logic for relaxed memory. So why do we say that the logical foundation of RustBelt is unsound for relaxed memory? The reason is as follows. The Iris framework (on which RustBelt is built) 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 proof rules of varying strength. In the case of RustBelt, Iris was instantiated with a sequentially consistent (SC) semantics for λ_{Rust}. This SC instantiation of Iris (call it "Iris-SC") provides a variety of proof rules that are valid only under SC semantics and not under relaxed-memory semantics. In particular, Iris-SC enables one to establish general *invariants* governing arbitrary regions of shared memory. Unfortunately, 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.

To adapt RustBelt to relaxed memory, we must therefore rebuild it using a logic that is suitably restricted so as to be sound under relaxed memory, yet still supports machine-checked proofs as Iris does. A promising path forward was charted by Kaiser et al. [2017], who showed how Iris could be used to derive a relaxed-memory separation logic called iGPS, targeting RA+NA (the fragment of C11 comprising release/acquire and non-atomic accesses). Following prior work on relaxed-memory separation logics [Vafeaidis and Narayan 2013; Turon et al. 2014], iGPS accounts for weak memory consistency by weakening the power of invariants: the user of iGPS may only establish *single-location invariants* (*i.e.*, invariants that govern a single shared memory location), the soundness of which is guaranteed by the *coherence* (or "SC per location") property of C11.

In this paper, drawing inspiration from FSL [Doko and Vafeiadis 2016, 2017], we will extend iGPS further to account for the additional features of the C11 memory model that Rust libraries make use of—specifically, *relaxed accesses* and *release/acquire fences*. We call this extended logic iRC11.

To a first approximation, we would therefore like to "port" RustBelt so that it is built on top of iRC11 rather than Iris-SC. Following the structure of RustBelt, this porting effort breaks down into two major tasks:

$$\text{SC-CInv-Acc}\ \frac{\{I * P\}\, e\, \{v.\, I * Q\} \qquad \text{atomic}(e)}{{}^{\tau}\boxed{I} \vdash \{[\tau]_q * P\}\, e\, \{v.\, [\tau]_q * Q\}}$$

$$\text{SC-CInv-Tok}\ \ [\tau]_{q+q'} \Leftrightarrow [\tau]_q * [\tau]_{q'}$$

$$\text{SC-CInv-Cancel}\ \ {}^{\tau}\boxed{I} \vdash [\tau]_1 \Rrightarrow\!\!\!\!\!\!\ast\ I$$

Fig. 1. Key rules for cancellable invariants in Iris-SC.

Task 1: Re-prove the safety of the Rust libraries considered by RustBelt, this time verifying their real, relaxed-memory implementations in iRC11.

Task 2: Re-prove the safety of the λ_{Rust} type system, this time relying only on proof rules that are sound in iRC11.

Key challenge. As it turns out, both of these tasks require us to overcome a technical challenge that is relevant not just to Rust but to relaxed-memory verification in general: namely, that **existing work on separation logic does not provide an adequate foundation for reasoning about resource reclamation under relaxed memory**. We will first explain this challenge in the context of Task 1, before briefly describing how it also informs Task 2.

Task 1: Re-prove the safety of Rust libraries under relaxed memory. One of the main motivations for using a "systems programming" language like Rust or C/C++ (as opposed to a garbage-collected language like Java) is to have more precise control over limited resources such as memory. In particular, the Rust programmer can be assured that when an object goes out of scope, the destructor (`drop` method) associated with its type will be invoked and any resources it owns will be reclaimed. Yet the safety of destructors is often quite subtle because objects can contain references to resources that are shared with other objects. For example, objects of type `Arc<T>` are simply aliases to a shared **`struct`** containing an object of type `T` along with a *reference counter*, which keeps track of the current number of active aliases to the object. Consequently, the destructor for `Arc<T>` cannot simply reclaim the shared **`struct`** that it points to: rather, it decrements the shared reference counter, and only if it observes that it was the last remaining alias can it safely reclaim the memory for the reference counter and invoke the destructor for the object of type `T`.

RustBelt showed how to put this subtle kind of resource reclamation on a sound formal footing using Iris-SC's mechanism of *cancellable invariants* (Fig. 1), a generalization of Gotsman et al. [2007] and Hobor et al. [2008]'s *storable* locks. A cancellable invariant ${}^{\tau}\boxed{I}$ is an invariant governing a shared resource (described by proposition I) which is only "active" for a certain period of time, after which point it is "cancelled". To access the shared resource during an atomic step of computation (SC-CInv-Acc), a thread must prove that the invariant is still active by exhibiting ownership of an *invariant token* $[\tau]_q$, where q is a fraction in (0,1]. This is an instance of the well-known concept of *fractional permissions* [Boyland 2003], and correspondingly, ownership of invariant tokens can be split or combined through fractional arithmetic (SC-CInv-Tok). If a thread π can assert ownership of $[\tau]_1$ (*i.e.*, the "full" τ token), it knows that no other thread can assert that the invariant is active; thus it is safe for π to cancel the invariant and reclaim full ownership of I (SC-CInv-Cancel), after which it can free the memory governed by I if it wants to. In RustBelt, cancellable invariants played a crucial role in verifying the safety of destructors such as `Arc`'s.

However, adapting cancellable invariants to the relaxed-memory setting turns out to be quite tricky—tricky enough that no existing relaxed-memory separation logic supports them.[1] Even if, following iGPS and its predecessors, we restrict invariants to govern a single location, a problem arises in how to model the cancellable invariant *tokens*. Under SC, one can simply model invariant

[1] iGPS supports a related notion of "fractional protocol", but as we explain in §6, it is not nearly as powerful as cancellable invariants and is thus not general enough to account for resource reclamation in Rust.

tokens as a form of *ghost state*, *i.e.*, purely logical state that is manipulated by the proof but does not appear in the physical program. But in existing relaxed-memory separation logics, ghost state is *unsynchronized*, meaning that ownership of it can be transferred between threads without the need for any physical synchronization. On the one hand (see §3.3), unsynchronized ghost state is indispensable for representing *globally consistent* state, such as (in the case of `Arc`) the number of `Arc` aliases currently in existence. On the other hand (see §4.1), if invariant tokens are modeled naively as unsynchronized ghost state, the logic of cancellable invariants becomes unsound!

Our solution is to instead model invariant tokens using a novel notion of *synchronized ghost state*: ghost state that implicitly tracks the subjective view of the thread that owns it, and that therefore can only be transferred between threads using physical synchronization. **Using synchronized ghost state, iRC11 offers the first general account of resource reclamation in relaxed-memory separation logic. We demonstrate its effectiveness on a number of real Rust libraries.**

Task 2: Re-prove the safety of the λ_{Rust} type system under relaxed memory. In contrast to RustBelt's proofs of safety for libraries, its proof of safety for the λ_{Rust} type system did not rely directly on cancellable invariants or any other SC-specific features of Iris-SC. Rather, as mentioned above, the safety proof for the type system made essential use of a Rust-oriented logic called the *lifetime logic*, which was a domain-specific logic derived within Iris-SC. Thus, if we are able to show that the lifetime logic remains sound under relaxed memory—by instead deriving its soundness in iRC11—then $\mathbf{RB_{rlx}}$ can inherit RustBelt's safety proof for the λ_{Rust} type system without modification!

Synchronized ghost state is the key to making this modular porting strategy possible. Specifically, the lifetime logic is centered around a mechanism called *borrow propositions*, describing resources that are borrowed for the duration of a Rust "lifetime" and that can be reclaimed once the lifetime is over. Borrow propositions are similar in many ways to cancellable invariants, but also more flexible and more complex in terms of the protocols they support for sharing and reclamation of resources. Just as synchronized ghost state enables us to adapt cancellable invariants to relaxed memory, it plays an analogously central role in adapting borrow propositions to relaxed memory as well.

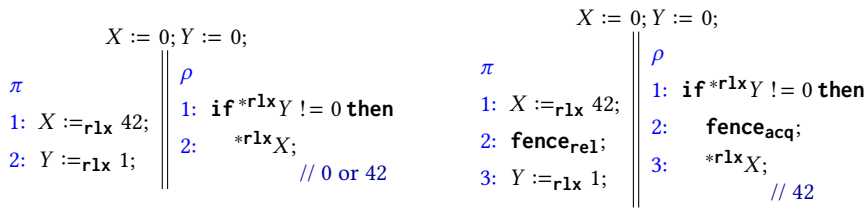
1.2 Contributions

In this paper, we present $\mathbf{RB_{rlx}}$, an adaptation of RustBelt to a relaxed memory model (or RMM), which (like its predecessor) is mechanized in Coq. The adaptation involves many components whose full technical explanation is beyond the scope of this paper. Instead, we focus on the key points of difference between RustBelt and $\mathbf{RB_{rlx}}$, and on the key technical innovations that make the adaptation feasible, most importantly our development of *synchronized ghost state*.

In summary, our contributions are as follows:

- We define ORC11, a new *operational-semantics-based* characterization of a large fragment of C11, including release/acquire/relaxed/non-atomic accesses and release/acquire fences.[2] Developing such an operational semantics for C11 is a necessary prerequisite for instantiating the Iris framework. Since the C11 model is known to be flawed [Boehm and Demsky 2014], we instead design ORC11 to match the semantics of RC11 (Repaired C11) [Lahav et al. 2017], and in the appendix [Dang et al. 2019] we sketch a proof of correspondence between them.
- We develop iRC11, a logic for ORC11 derived within Iris, which combines elements of iGPS and FSL, and moreover supports resource reclamation via cancellable invariants in a manner that is sound for relaxed memory. The soundness of iRC11 relies crucially on our novel construction of *synchronized ghost state*.

[2] Caveat: ORC11 omits SC accesses and fences because (1) they are not used by any of the libraries verified in RustBelt, and (2) it is still an open question how to develop a separation logic for reasoning about SC accesses in a relaxed-memory setting. See §6 for further details.



(a) MP with **rlx** accesses.

(b) MP with **rlx** accesses and fences.

Fig. 2. Message-Passing example in RMM.

- We use iRC11 to port RustBelt from SC to relaxed memory. In particular, the major components that required re-verification were the library proofs (since we are now verifying implementations with relaxed-memory operations in them) and the proof of soundness of RustBelt's lifetime logic. The proof of safety of λ_{Rust}'s type system, by virtue of being built atop the lifetime logic, did not need to be changed at all.

The rest of the paper is structured as follows. In §2, we briefly review the basics of relaxed-memory programming under the C11 memory model, as well as the basic idea (drawn from previous work) of how one can characterize such a memory model operationally. In §3, we present a representative fragment of iRC11, focusing on how it supports resource reclamation via cancellable invariants. In §4, we discuss how to prove soundness of iRC11: we motivate the need for synchronized ghost state in building a model for iRC11, and describe in some detail how the model works. In §5, we give further details about ORC11 (§5.1) and iRC11 (§5.2), about verifying and finding a bug in the `Arc` library (§5.3), and about porting the lifetime logic (§5.4). In §6, we conclude with a discussion of related work.

2 BACKGROUND: RELAXED-MEMORY OPERATIONAL SEMANTICS

We briefly review the C11 memory model in §2.1 and ORC11—the operational version of C11 that we use as the memory model for $\mathbf{RB}_{\mathbf{rlx}}$—in §2.2.

2.1 C11, Intuitively

The C11 memory model offers several different modes of memory accesses, including non-atomic (**na**), relaxed (**rlx**), release (**rel**), acquire (**acq**), and sequentially consistent (**sc**). Non-atomic accesses are "normal" data accesses, meaning that it is the programmer's responsibility to ensure that they are properly synchronized through other means. (If they are not properly synchronized—*i.e.*, there is a data race involving non-atomics—then C11 says the whole program has undefined behavior.) The remaining modes, collectively called *atomic* accesses, are allowed to be racy and are indeed used to establish synchronization among non-atomic accesses.

To explain what synchronization actually means, we explore the examples in Fig. 2. In Example 2a, we initialize two locations X and Y to 0, then spawn two threads π (on the left) and ρ (on the right). Thread π intends to pass a "message" to ρ. The message, 42, is stored in X. Thread π then sets the boolean flag Y to 1, to signal to ρ that the message is ready to be received. Once ρ sees the flag set, it attempts to read the message from X. However, both the intended value of 42 as well as the initial value of 0 could be read. That is, even though ρ has read 1 from Y, it is not guaranteed to read 42 from X. This is because the relaxed accesses of Y are not enough to establish synchronization.

In the C11 memory model, threads are not synchronized by default: they each have their own perspective on the values in shared memory, and thus may observe memory events in different

order. However, certain ways of performing accesses, as defined in C11, allow all threads to agree that one event *happens-before* another, and to "establish synchronization" is to somehow guarantee that two memory events of interest will be in the happens-before relation. Relaxed accesses are the weakest atomic accesses in C11 and do *not* guarantee happens-before. Thus, in Example 2a, the relaxed accesses on Y do not establish synchronization between the accesses on X.

In order to synchronize, we complement relaxed accesses with *fences*.[3] In Example 2b, thread π performs a release fence after the write to X, and then writes to Y. Meanwhile, ρ performs an acquire fence once it reads 1 from Y. Now, when ρ reads from X the result will be 42.

π successfully passes the message to ρ because C11 guarantees happens-before through chains of the form "release fence → relaxed write → relaxed read → acquire fence". That is, synchronization is guaranteed between the events *before* the release fence and the events *after* the acquire fence if the two fences are connected by the relaxed write and read. As a result, we know that π's write of 42 to X happens before ρ's read of X—or in other words, that ρ's read of X is *synchronized with* π's write to it. Since the write of 42 is the most recent write to X, we know that thread ρ must read 42.

Data races. Note that for Example 2a, where we do not have sufficient synchronization between the accesses to X, the worst thing can happen is that ρ would read unwanted values. However, if we were to replace the **rlx** accesses of X with non-atomic accesses (**na**), it would constitute a data race and the program would exhibit undefined behavior.

2.2 High-Level Overview of ORC11 (Operational RC11)

ORC11 is the operational version of C11 that we use for $\mathbf{RB}_{\mathtt{rlx}}$. Its expression language, closely following that of λ_{Rust} [Jung et al. 2018a], is a standard lambda calculus with recursive functions, fork-based concurrency, and references extended with relaxed consistency access modes:

$$\begin{aligned} &\mathit{AccessMode} \ni o ::= \mathtt{na} \mid \mathtt{rlx} \mid \mathtt{acq} \mid \mathtt{rel} \\ &\mathit{Expression} \ni e ::= \ldots \mid \mathtt{rec}\, f(\overline{x}) := e \mid \mathtt{fork}\{e\} \mid \mathtt{alloc}(e) \mid \mathtt{free}(e_1, e_2) \mid \\ &\qquad \ldots \mid {}^{*o}e \mid e_1 :=_o e_2 \mid \mathtt{CAS}_{o_f,o_r,o_w}(e_0, e_1, e_2) \mid \mathtt{FAA}_{o_r,o_w}(e_1, e_2) \mid \mathtt{fence}_o \mid \ldots \end{aligned}$$

The syntax includes standard types of memory accesses, including reads, writes, compare-and-set (**CAS**), and fetch-and-add (**FAA**), which atomically increments the contents of a location by a given integer. All memory accesses and fences are annotated with access modes o. As a shorthand notation, we use ${}^*\ell$ and $\ell := v$ for **na** reads and writes (omitting the o).

alloc(e) allocates a fresh block of memory with size determined by e. **free**(e_1, e_2) deallocates a block of memory starting at e_1 with size e_2 (we will often elide the size argument for simplicity). C11 only specifies that the lifetime of an object is from its allocation to deallocation, but does not specify a synchronization condition or possible races between allocation/deallocation and normal accesses. To fill this gap, we employ the following conditions that are widely thought to be reasonable.

The allocation of a block must happen-before all accesses to it. (Alloc-Safe)

The deallocation of a block must happen-after all accesses to it. (Free-Safe)

We consider violations of these conditions data races and, thus, undefined behavior. To implement them, we treat the semantics of allocations and deallocations as that of non-atomic writes with special values.

We follow the reduction semantics of λ_{Rust} except where interaction with memory is concerned, where we base the reduction rules on C11. As mentioned in the introduction, we cannot directly use

[3] We could instead use a pair of release write and acquire read accesses. We use fences for the sake of exposition.

C11 because (1) its semantics is flawed in several ways, and (2) it is formalized in an axiomatic (event-graph) style, whereas for the purpose of instantiating Iris, we require an operational semantics that relates the programs state before and after an execution step. To address (1), we instead target RC11 [Lahav et al. 2017], which fixes the major flaws in C11. To address (2), we take inspiration from the approaches of Kaiser et al. [2017] and Kang et al. [2017], developing ORC11, an operational version of RC11. We give some more details about ORC11 in §5.1. In this section, we focus on the high-level aspects of ORC11 that are relevant for understanding the main body of this paper.

The key concept of the ORC11 semantics is that of *views*. A thread's view records the memory events (reads and writes) that the thread has observed so far—*i.e.*, the events that happen-before the current step in the thread's execution. To account for C11's notion of synchronization, we define a partial *view inclusion* order on views: $V_1 \sqsubseteq V_2$ holds if and only if all events in V_1 are also present in V_2, in which case we sometimes say that V_2 has "seen" all the events in V_1. Correspondingly, views also come with a join operator $\sqcup$ that performs the role of synchronizing with another view: $V_1 \sqcup V_2$ corresponds to the union of events in V_1 and V_2. When a thread updates its view from V_1 to $V_1 \sqcup V_2$, we will say that the thread is synchronizing with V_2.

Every thread π actually maintains three views, which together form the *thread-local view* $\mathcal{V}$:

(1) the current view $\mathcal{V}.\mathsf{cur}$ contains all the events that π has observed,
(2) the release view $\mathcal{V}.\mathsf{rel}$ contains events that π had observed at its *last* release fence, and
(3) the acquire view $\mathcal{V}.\mathsf{acq}$ contains events that π will have observed by its *next* acquire fence.

Since a thread's views only grow over time, it is always the case that $\mathcal{V}.\mathsf{rel} \sqsubseteq \mathcal{V}.\mathsf{cur} \sqsubseteq \mathcal{V}.\mathsf{acq}$.

The *global* state consists of two parts $(\mathcal{M}, V_{\mathbf{Race}})$. First, $\mathcal{M}$ is the *message pool* that tracks all write events for all locations. It is modeled as a finite partial function from locations and timestamps to *messages*. Timestamps are used to encode a location's modification order ("*mo*")—a C11 relation that totally orders all writes to a single location. (The details of timestamps are largely inherited from Kaiser et al. [2017] and Kang et al. [2017]—for most of the paper, we will keep the discussion at the more abstract level of views and view inclusion.) Messages are records that contain (1) the value $m.(\mathsf{val})$ written, and (2) a view $m.(\mathsf{view})$ called the *message view*. The message view is used to communicate the observations from one thread's write to another thread's read, establishing synchronization (see below). The second component of the global state, $V_{\mathbf{Race}}$, is called the *race detector* view, because (unsurprisingly) it gets used by ORC11's race detector, as we describe in §5.1.

ORC11 is defined with a standard threadpool step relation, lifted from a per-thread step relation $(\mathcal{M}, V_{\mathbf{Race}}) \mid (e, \mathcal{V}) \rightarrow (\mathcal{M}', V'_{\mathbf{Race}}) \mid (e', \mathcal{V}')$, where the scheduled thread π is executing its expression e with its thread-local view $\mathcal{V}$ and the shared global state $(\mathcal{M}, V_{\mathbf{Race}})$.

We refer the reader to our appendix and Coq development [Dang et al. 2019] for the full semantics. Below, we give a high-level description for the semantics of the atomic operations used in Example 2b.

Step-Rlx-Write A relaxed write adds a message m, whose message view is the thread's *release* view $\mathcal{V}.\mathsf{rel}$ extended by the write itself, to the message pool $\mathcal{M}$. Only the current view $\mathcal{V}.\mathsf{cur}$ and acquire view $\mathcal{V}.\mathsf{acq}$, but *not* $\mathcal{V}.\mathsf{rel}$, are updated to include the relaxed write event itself.

Step-Rel-Fence A release fence joins the thread's current view $\mathcal{V}.\mathsf{cur}$ into its release view $\mathcal{V}.\mathsf{rel}$.

Step-Rlx-Read A relaxed read that reads the write encoded by m will join the message view $m.(\mathsf{view})$ into the thread's *acquire* view $\mathcal{V}.\mathsf{acq}$.

Step-Acq-Fence An acquire fence joins the acquire view $\mathcal{V}.\mathsf{acq}$ into the current view $\mathcal{V}.\mathsf{cur}$.

In Example 2b, in line $\pi 1$, the relaxed write of 42 to X is recorded in π's current view, but *not* its release view. In line $\pi 2$, the release fence synchronizes π's release view with its current view. In

line $\pi 3$, the relaxed write to Y creates a message m with π's release view—including the write of 42 to X. In line $\rho 1$, if reading 1, the relaxed read joins m.(view) into ρ's acquire view. In line $\rho 2$, the acquire fence joins ρ's acquire view into its current view, which now includes the write of 42 to X. In line $\rho 3$, the relaxed read selects a message to read from that is no older than ρ's most recent observed write to X. With ρ having observed π's write of 42 to X, it must then read 42. In this way, the ORC11 view semantics implements the release/acquire synchronization chain described above.

3 RESOURCE RECLAMATION UNDER RELAXED MEMORY WITH IRC11

iRC11 is a relaxed-memory separation logic for ORC11. Following iGPS [Kaiser et al. 2017], ORC11 is an instantiation of the Iris framework [Jung et al. 2018b]. Its soundness is derived within Iris, and it inherits its features both from the general Iris framework and from previous RMM logics. We begin in §3.1 by reviewing a small but representative set of features of iRC11 that come from previous RMM logics. In §3.2, we present the most notable novelty of iRC11: cancellable single-location invariants. In §3.3, we review the verification of the core of the `Arc` data type to demonstrate a crucially powerful property of the primitive ghost state assertions in RMM logics, namely that they are *unsynchronized*. (As explained in §1.1, we will need to develop a new form of *synchronized* ghost state when we model cancellation in §4.)

3.1 iRC11: The Old

Here, we review several core features of iRC11 that are inherited from prior RMM logics, including single-location invariants, fence modalities, and user-defined ghost state. A selection of rules concerning these features is displayed in Fig. 3. For now, please ignore all shaded parts, as these are new iRC11 additions pertaining to cancellable invariants, and we will discuss them in §3.2. More specifically, please ignore any parts concerning τ, as well as the two rules iRC11-CInv-Tok and iRC11-CInv-Cancel.[4]

Hoare triples and fancy updates. In these rules, we write $\{P\}\ e\ @\ \pi\ \{v.\ Q\}$ for Hoare triples with pre-condition P, expression e, thread id π (sometimes omitted), and post-condition Q where Q may mention v, *i.e.*, the result of evaluating e. The thread id π is needed for fence modality assertions, as we explain below. A number of proof rules also make use of Iris's *fancy updates*, written $P \Rrightarrow\!\!\!\!* Q$. Fancy updates are essentially a more flexible form of logical implication that also includes the ability to update ghost state—*e.g.*, see Count-Ghost-Update in §3.3—among other things.

Assertions as view predicates. First of all, assertions in RMM logics, iRC11 included, need to represent not just ownership of some resources, but ownership of some resources *with respect to* the local perspective of the thread owning those resources. 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 $\ell \mapsto v$, it should be guaranteed (among other things) that a read from ℓ will return v. In RMM, ownership of $\ell \mapsto v$ must therefore say something about the current local view V of the thread asserting it: V should contain the latest write to ℓ, and it should have value v. Otherwise, reading from ℓ could yield an older value and thus render at least one guarantee of the points-to assertion invalid.

The assertions in Fig. 3 are thus to be interpreted *at some view*. In the case of Hoare triples and assertions owned by a thread, the view used to interpret them is, by default, the thread's current

[4] **Note:** Throughout this paper, to simplify the presentation, we elide certain "administrative" details of proof rules concerning invariant namespaces and the "later" ($\triangleright$) modality. Invariant namespaces are present to ensure that we do not attempt to access invariants twice in a nested fashion. The $\triangleright$ modality is necessary to ensure soundness in the presence of impredicative invariants. We elide these details because they are completely standard in Iris developments, and do not interact interestingly with the RMM features. We refer the reader to Jung et al. [2018b] for details.

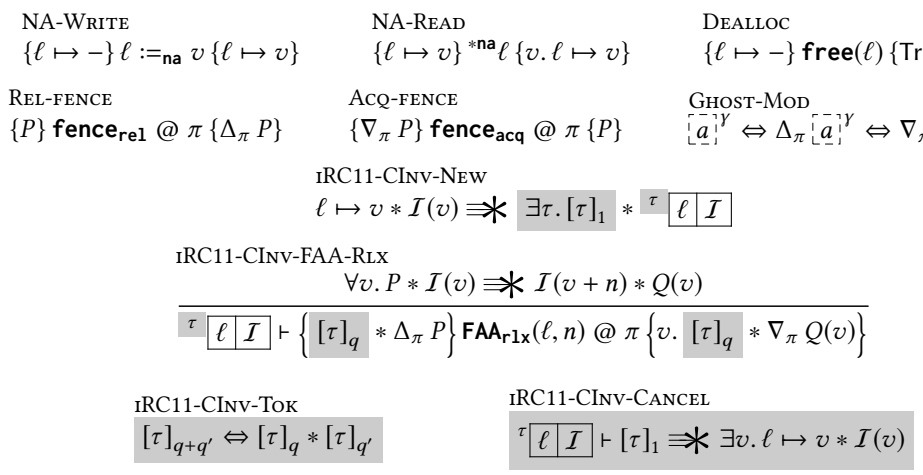


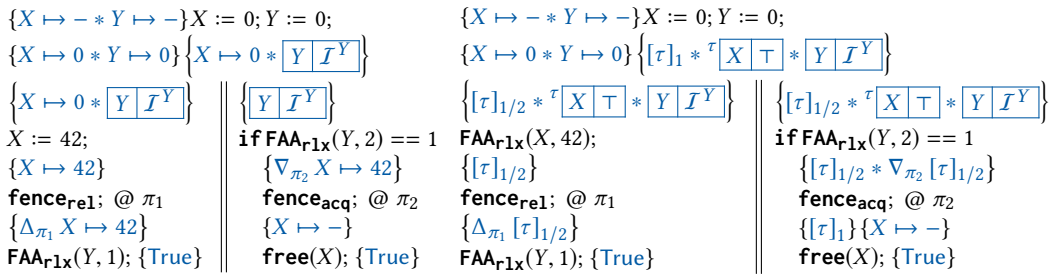
Fig. 3. Selected iRC11 rules (new additions for cancellable invariants are shaded). Note that fractions are always assumed to be well-formed, *i.e.*, $q \in (0, 1]$.

view. The view picked to interpret assertions, however, is not always necessarily some thread's current view. To allow transferring resources through writes, we also want to attach resources to write messages. In that case, the resources attached to a message m will be interpreted at the message view m.(view). Furthermore, as explained in §2.2, we have the fine-grained notion of a thread's local view $\mathcal{V}$ with its current view $\mathcal{V}$.cur, release view $\mathcal{V}$.rel, and acquire view $\mathcal{V}$.acq. This leads to more choices (and more obligations) in picking which view to use when interpreting an assertion. As we will see below, the release and acquire views come into play when interpreting assertions concerning relaxed accesses and fences.

Points-to assertions. A familiar feature in separation logics is the points-to assertion. In iRC11, the points-to assertion $\ell \mapsto v$ represents the full ownership of the location ℓ as well as the knowledge that ℓ has the current value v. The assertion $\ell \mapsto -$ simply ignores this knowledge about the value: $\ell \mapsto - ::= \exists v.\ \ell \mapsto v$. With full ownership, one can perform any operation on ℓ, including non-atomic operations (NA-Write, NA-Read), atomic operations, and deallocation (Dealloc). Following prior RMM logics, iRC11 also supports *fractional* points-to assertions (not shown here) so that the permission to non-atomically read a location can be split up between concurrent threads.

Consider a variant of the Message-Passing example in Fig. 4a. Here, after allocation, we can initialize two locations X and Y to 0 with NA-Write because we own their points-to assertions. After spawning two threads, since we give thread π_1 (on the left) the points-to $X \mapsto 0$, it can write non-atomically to X again with NA-Write. In thread π_2 (on the right), if somehow after the acquire fence we receive the "message" $X \mapsto -$ from thread π_1, then we can use Dealloc to free X. To complete the proof of this example, we need single-location invariants for Y.

Single-location invariants. All RMM logics have a notion of single-location invariants. This feature comes from the observation that, although general invariants are unsound in RMM because threads may have different views on a region of multiple locations, invariants that only govern a single location are sound because threads do agree on the order of writes to any single location. Prior RMM logics support a range of somewhat different single-location invariant mechanisms, including some that let the user establish "protocols" on how a location's value can evolve over time. iRC11 inherits such a protocol mechanism from iGPS, but for the purpose of this paper we focus on a



(a) With basic invariants. (b) With cancellable invariants.

Fig. 4. iRC11 demonstrated with Message-Passing example verifications.

simplified version that ignores the extra functionality of protocols, as that is orthogonal to the issue of resource reclamation.

The invariant assertion $\boxed{\ell \mid \mathcal{I}}$ asserts the knowledge that an invariant $\mathcal{I}$ governs ℓ. As an invariant represents *knowledge*, not exclusive ownership, it is a *duplicable* assertion that can be passed on to multiple concurrent threads. With iRC11-CInv-New we can initialize an invariant for ℓ if we have the full ownership $\ell \mapsto v$ and the resource $\mathcal{I}(v)$. The predicate $\mathcal{I}$, also called the *interpretation*, is a user-defined predicate on values: $\mathcal{I}(v)$ describes the invariant over shared resources that must hold in order for ℓ to have the value v. $\mathcal{I}(v)$ is a requirement that every write of value v to ℓ must provide. Intuitively, the resource $\mathcal{I}(v)$ is attached to the message m created by the write of v to ℓ. When a read of value v from ℓ finds the message m, it can use $\mathcal{I}(v)$ for its reasoning. As such, the interpretation encodes resources to be transferred from writes to reads.

This is demonstrated in the premise of the fetch-and-add (**FAA**) *access* rule iRC11-CInv-FAA-Rlx for an invariant $\boxed{\ell \mid \mathcal{I}}$. A **FAA** is a read-modify-write (RMW) operation that has the atomic effect of both a read and a write. (Note that we are focusing on **FAA** here, rather than plain relaxed writes and reads, merely because the proof rules for **FAA** are a bit simpler and they suffice for the purpose of our story in this paper. In iRC11, we of course also support rules for reads, writes, and **CAS**.) As the **FAA** is a write, we must establish the interpretation $\mathcal{I}(v + n)$ for the value $v + n$ that the **FAA** is going to write. But we can also use the interpretation $\mathcal{I}(v)$ for the value v that the **FAA** read and our local resource P to establish $\mathcal{I}(v + n)$. If there is any remaining resource $Q(v)$, we can take it out as our local resource afterwards. This is the standard way in RMM logics to model the *ownership transfer* that is achievable with RMW operations like **FAA**.

In Fig. 4a, we want to transfer $X \mapsto -$ through the communication from thread 1's **FAA** to thread π_2 s **FAA** on Y. To do so, we set up an invariant $\mathcal{I}^Y$ for Y:

$$\mathcal{I}^Y(v) ::= v \geq 0 \land \mathbf{if}\ v = 1\ \mathbf{then}\ X \mapsto -\ \mathbf{else}\ \mathrm{True}$$

That is, the invariant owns $X \mapsto -$ if the value of Y is currently 1. With $Y \mapsto 0$, we can initialize the invariant for Y with iRC11-CInv-New where $\mathcal{I}^Y(0)$ is trivial. We then pass knowledge of the invariant $\boxed{Y \mid \mathcal{I}^Y}$ to both threads. When thread π_1 writes 1 (**FAA** from 0 to 1) to Y, it uses iRC11-CInv-FAA-Rlx with $P ::= X \mapsto -$ to transfer the ownership of X into $\mathcal{I}(1)$ (and with $Q ::= \mathrm{True}$ to take nothing out). When thread π_2 reads 1 (**FAA** from 1 to 3) from thread π_1's write, it also uses iRC11-CInv-FAA-Rlx with $Q ::= X \mapsto -$ to take the ownership out of $\mathcal{I}^Y(1)$ (and with $P ::= \mathrm{True}$ because it does not need anything to establish the trivial $\mathcal{I}^Y(3)$). The transfer, however, is not completed yet, because there are caveats in using the **rlx** access mode, which we explain next.

Fence modalities. To model the effects of relaxed accesses and fences, iRC11 inherits two modalities from FSL [Doko and Vafeiadis 2016, 2017]—the *release* modality Δ and the *acquire* modality ∇—which allow us to talk about ownership of resources at a thread's release or acquire views. The assertion $\Delta_\pi\ P$ represents ownership of P at thread π's release view, while the assertion $\nabla_\pi\ P$ represents ownership of P at thread π's acquire view.

The motivation for these modalities as follows. We have some resource described by the proposition P that we want to transfer from one thread to another through a pair of a relaxed write and a relaxed read (or in our example, a pair of relaxed **FAA**'s), communicating through a single-location invariant (in our example, $\mathcal{I}^Y$). However, when the "producer" thread π_1 performs its relaxed write, the message view of that write is drawn from π_1's release view, not its current view. Hence, we need a way of insisting (in the precondition of the relaxed write) that the P that π_1 is sending holds under its release view—that is what is denoted by $\Delta_{\pi_1}\ P$. Dually, when the "consumer" thread π_2 performs its relaxed read, the message view it reads will only be joined into its acquire view, not its current view. Hence, we need a way of insisting (in the postcondition of the relaxed read) that π_2 only receives ownership of P under its acquire view—that is what is denoted by $\nabla_{\pi_2}\ P$. Since **FAA** combines a read and a write, the **FAA** rule (iRC11-CInv-FAA-Rlx) combines both of these reasoning principles into one.

Of course, we now need a way of actually *introducing* $\Delta_{\pi_1}\ P$ and *eliminating* $\nabla_{\pi_2}\ P$. These steps are achieved by rules Rel-fence and Acq-fence, which allow one to transfer any proposition *into the release modality* at the point of a **rel** fence, or *out of the acquire modality* at the point of an **acq** fence, because those are the points where the current and release/acquire views get synchronized.

In Fig. 4a, because we are using relaxed **FAA**'s, we have to provide $\Delta_{\pi_1}\ X \mapsto -$ in thread π_1, and we receive $\nabla_{\pi_2}\ X \mapsto -$ in thread π_2. In thread π_1, with a release fence, we use Rel-fence to put $X \mapsto -$ under the release modality and then perform the **FAA**. In thread π_2, with an acquire fence, we use Acq-fence to regain $X \mapsto -$. Then the transfer is completed and thread π_2 can safely deallocate X.

3.2 iRC11: The New – Cancellable Invariants

The problem with standard single-location invariants from previous logics is that they cannot be reclaimed. Consider the example in Fig. 4b, which is identical to Fig. 4a except that X is used atomically instead of non-atomically. To use X atomically, we need to govern it with invariants. But in prior logics, any invariant placed on X would govern it *forever*, making it non-reclaimable.

One key novelty of iRC11 is the ability to *cancel* its single-location invariants, so that atomic locations can be reclaimed fully. To make single-location invariants cancellable, iRC11 takes a page from Iris-SC cancellable invariants (Fig. 1) where the invariant is guarded by a token τ (see the shaded part in Fig. 3). As seen in iRC11-CInv-FAA-Rlx, only those who own a fraction $[\tau]_q$ of τ can access the invariant. Similar to SC cancellable invariants, the ownership $[\tau]_q$ guarantees that the invariant ${}^{\tau}\boxed{\ell \mid \mathcal{I}}$ has not been cancelled and is still accessible.

After initializing the invariant with iRC11-CInv-New, we obtain the full token $[\tau]_1$ which can be split and rejoined with iRC11-CInv-Tok so that multiple threads can access the invariant concurrently. Once all threads are done using the location through the invariant, the thread π owning the full token $[\tau]_1$ can cancel the invariant with the rule iRC11-CInv-Cancel. The thread π then reclaims the points-to $\ell \mapsto v$ for some value v, which guarantees that π has synchronized with all other accesses to ℓ, and thus π can safely deallocate ℓ or use ℓ non-atomically. The thread also reclaims $\mathcal{I}(v)$, the interpretation at the *last* write before the cancellation. $\mathcal{I}(v)$ can be useful, for example, when ℓ is a lock and v is the "unlocked" state. In that case, the lock-protected content is still in $\mathcal{I}(v)$ and we can reclaim it. This feature is thus important in verifying locks, of which Rust's `Mutex<T>` library is an example.

To verify Fig. 4b, we create a cancellable invariant for X, guarded by τ, with a trivial interpretation $\mathcal{I}^X ::= \text{True}$ (written $\top$ in the proof outline) because we only want to access X concurrently, not to transfer any resource through it. We still use a non-cancellable invariant for Y since we do not care about reclaiming Y here. But we do use a slightly different interpretation for Y: $\mathcal{I}^Y(v) ::= v \geq 0 \land \textbf{if}\, v = 1\, \textbf{then}\, [\tau]_{1/2}\, \textbf{else}\, \text{True}$. Here, instead of transferring $X \mapsto -$ (which allows non-atomic accesses to X) through the invariant of Y, we transfer $[\tau]_{1/2}$ (which allows atomic accesses to X). We use the invariant $\mathcal{I}^Y$ only to transfer half of the token ($[\tau]_{1/2}$) from thread π_1 to thread π_2, because we also want to give the other half to thread π_2 (when spawning π_2) so that it can access X concurrently with thread π_1 (not shown in the example). After the acquire fence, thread π_2 will then regain the full token $[\tau]_1$ (instead of $X \mapsto -$); then it can use iRC11-CInv-Cancel to cancel the invariant and reclaim $X \mapsto -$, after which it can safely free X.

3.3 User-Defined Ghost State in Relaxed Memory

We turn our attention now to an essential feature of the more recent RMM logics—*user-defined ghost state*—which iRC11 inherits directly from Iris. Ghost state is logical state that is used to track additional information in the verification, but is not part of physical state. In Iris, the user of the framework is free to define the particular structure of ghost state—in the form of a *partial commutative monoid* (PCM)—that is appropriate for their proof. More specifically, the user can define their own PCM M and Iris will give them a new class of *ghost assertions* of the form $\boxed{a}^\gamma$, which asserts the ownership of some *ghost element* a (a member of M) stored at a *ghost location* γ. Iris also provides a few basic proof rules for reasoning about these ghost assertions, with which the user can derive a "ghost theory" (a logical API) applicable to their PCM [Jung et al. 2018b].

User-defined ghost state is useful in deriving domain-specific logics. For example, the points-to assertions, single-location invariants, and fence modalities are all derived by iRC11 in Iris, each with the help of a specialized PCM. Furthermore, iRC11 also exposes this facility to its users, so that they can employ it for their own verifications of RMM algorithms. And, indeed, ghost state has proven indispensable for verifying intricate concurrent data structures in both traditional SC and RMM separation logics. In the setting of RMM, however, ghost state enjoys an additional property that is encoded in the Ghost-Mod rule (Fig. 3).

Ghost-Mod states that the ghost assertion $\boxed{a}^\gamma$ can move freely in and out of the fence modalities. This is because ghost state belongs to the class of *unsynchronized* assertions, in the sense that their ownership is *not tied to the physical, subjective view of the thread asserting them*. Recall that $\Delta_\pi\, P$ and $\nabla_\pi\, P$ assert that P holds at π's release view and acquire view, respectively. Since $\boxed{a}^\gamma$ is unsynchronized and thus does not care at which view it is interpreted, it is equivalent to $\Delta_\pi \boxed{a}^\gamma$ or $\nabla_\pi \boxed{a}^\gamma$. As a result, $\boxed{a}^\gamma$ can be transferred from one thread to another *without the need for physical synchronization*—in particular, without the need for release/acquire fences. (For more details, please see the model of fence modalities and unsynchronized ghost state in §5.2.)

To help the reader appreciate the importance of Ghost-Mod, we now quickly review the verification of Core `Arc`, a simplified version of Rust's `Arc` library. (See §5.3 for more details about our verification of the full `Arc`.) Core `Arc` has been verified previously in FSL++ [Doko and Vafeiadis 2017], but their proof did not account for the reclamation of `Arc`'s reference count field. In $\mathbf{RB_{rlx}}$, we improve on their proof by fully verifying `Arc`'s destructor using iRC11 cancellable invariants.

3.3.1 Core `Arc` Library. `Arc<T>`, short for *Atomically Reference Counted*, is used to share atomically an object of type `T`, whose mutation is disabled by default. To mutate `T`, one needs `T` to support thread-safe mutability, for example with `T` being an atomic type, or with `T` wrapped inside a lock (*e.g.*, `Mutex<T>`). The following Rust example instantiates `Arc` with an atomic integer `AtomicUsize` and demonstrates how `Arc` is typically used:

```
new(v) := let a = alloc(2) in
          a.counter := 1;
          a.data := v;
          a
deref(a) := *na a.data

drop(a) := if FAArel(a.counter, −1) == 1
             fenceacq;
             free(a, 2)
clone(a) := FAArlx(a.counter, 1);
            a
```

Fig. 5. Implementation of Core `Arc`.

```
1  let arc1 = Arc::new(AtomicUsize::new(5)); // create the first Arc pointer
2  let arc2 = Arc::clone(&arc1);             // clone for the second pointer
3  thread::spawn(move || {                   // give arc2 to child thread
4    println!("child: {:?}", arc2.fetch_add(1, Ordering::Relaxed)); // drop(arc2);
5  });
6  println!("main: {:?}", arc1.fetch_add(2, Ordering::Relaxed));    // drop(arc1);
```

In line 1 in the main thread, a new `Arc` pointer `arc1` is created to govern an atomic integer allocated in shared memory, and `arc1`'s internal `counter` field for the number of references to the content is set to 1. An `Arc` pointer acts almost like its underlying content, so in line 6 we can call `fetch_add` on `arc1` as if on the atomic integer itself. To share the content with the child thread, we create another `arc2` by `clone`-ing `arc1` (line 2), which has the effect of incrementing the `counter` field to 2: there are now 2 `Arc` objects sharing the atomic integer. Unsurprisingly, to allow concurrent `clone`-ing, the `counter` field is itself implemented with an atomic integer, too.

When the `Arc` pointers go out of scope (after lines 4 and 6), their destructor—the `drop` method—is called, and the `counter` field is decremented accordingly. The last call of `drop` will deallocate both the `data` *and* `counter` fields.

The implementation of Core `Arc` is given in Fig. 5. The `new` method allocates a region of two locations for the `counter` and `data` fields, then initializes them. The `deref` method provides access to the `data` field, effectively allowing an `Arc<T>` to behave like its content `T`. The `clone` method does a **`rlx FAA`** by 1 to increment `counter` and then returns a copy of `a`.

Finally, the `drop` method does a **`rel FAA`** by −1 to decrement `counter`. If the value of `counter` was 1 before the decrement—*i.e.*, this is the last `drop`—`drop` additionally does an acquire (**`acq`**) fence before deallocating both the `counter` and `data` fields. With that acquire fence, the deallocation of the region `a` in the last `drop` is guaranteed to be synchronized with all previous calls to `drop`. This, in turn, guarantees that the deallocation is synchronized with all concurrent accesses by other pointers to the region `a`—*i.e.*, there is no data race between the deallocation and the accesses.

3.3.2 Core `Arc`'s Invariant. Core `Arc` has the following simple specification:

$$\{\mathsf{True}\}\ \mathtt{new}(v)\ \{\mathsf{a}.\ \exists\ \tau, \gamma.\ \mathsf{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})\} \qquad \text{(ARC-New)}$$

$$\{\mathsf{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})\}\ \mathtt{clone}(\mathsf{a})\ \{\mathsf{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I}) * \mathsf{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})\} \qquad \text{(ARC-Clone)}$$

$$\{\mathsf{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})\}\ \mathtt{deref}(\mathsf{a})\ \{x.\ x = v * \mathsf{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})\} \qquad \text{(ARC-Deref)}$$

$$\{\mathsf{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})\}\ \mathtt{drop}(\mathsf{a})\ \{\mathsf{True}\} \qquad \text{(ARC-Drop)}$$

Here, ARC is an abstract predicate that represents the logical ownership of an `Arc` object and gives one permission to invoke methods on the `Arc` object `a`, and $\mathcal{I}$ is a single-location invariant governing the shared location `a.counter`, which enables threads to access `a.counter` atomically. The two main challenges of the verification are (1) cloning the ARC permission (in ARC-Clone) even though `clone` only uses a relaxed **`FAA`** and (2) safely deallocating the region at the last invocation of `drop` (ARC-Drop).

The definitions of ARC and $\mathcal{I}$ are given as follows. We explain them piecemeal below, beginning with the unshaded parts, which are essentially the same as in the proof of Doko and Vafeiadis [2017].

$$\mathsf{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I}) ::= \exists q.\, \mathsf{a.data} \overset{q}{\longmapsto} v * [\tau]_q * \overset{\tau}{\boxed{\mathsf{a.counter} \mid \mathcal{I}^{\gamma,v}}} * \boxed{\mathsf{Count}(q)}^{\gamma} \quad \text{(ARC-Own)}$$

$$\mathcal{I}^{\gamma,v}(n) ::= \begin{cases} \mathsf{False} & n < 0 \\ \boxed{\mathsf{TotalCount}(0,0)}^{\gamma} & n = 0 \\ \exists q_{\mathrm{in}}, q_{\mathrm{out}} \in (0,1).\, \mathsf{a.data} \overset{q_{\mathrm{in}}}{\longmapsto} v * [\tau]_{q_{\mathrm{in}}} & \\ * \, q_{\mathrm{in}} + q_{\mathrm{out}} = 1 * \boxed{\mathsf{TotalCount}(n, q_{\mathrm{out}})}^{\gamma} & n > 0 \end{cases} \quad \text{(ARC-Inv)}$$

Each ARC permission has several components. First, it has a *fractional* points-to assertion for a.data. This fractional assertion is essential for verifying ARC-Deref, since the deref method relies on the ARC permission to justify a non-atomic read of a.data. In the verification of ARC-Clone, this fractional permission is split into two half permissions ($\mathsf{a.data} \overset{q/2}{\longmapsto} v$), which are then used to satisfy the two ARC predicates in the postcondition. Second, ARC stipulates the knowledge that a.counter is governed by the single-location invariant $\mathcal{I}$. This invariant, in turn, owns the *remainder* of a.data: the fraction of a.data—namely, q_{in}—that is not owned by any ARC. Third, ARC owns a *ghost* element $\boxed{\mathsf{Count}(q)}^{\gamma}$. We now explain in detail what purpose this ghost element serves, and how it interacts with the invariant $\mathcal{I}$.

3.3.3 Unsynchronized Ghost State for Core Arc. The invariant $\mathcal{I}$ needs to maintain that the total number of ARC permissions currently in existence is equal to the current physical value n of a.counter. It does so using a "ghost theory" of counting permissions [Bornat et al. 2005]. This ghost theory involves two ghost assertions:

- the *single count* assertion, $\boxed{\mathsf{Count}(q)}^{\gamma}$, which represents the contribution of a single ARC predicate to the total number of ARC permissions currently in existence, as well as the fact that that ARC predicate owns a q fraction of a.data, and
- the *total count* assertion, $\boxed{\mathsf{TotalCount}(n, q_{\mathrm{out}})}^{\gamma}$, which stipulates that currently there are n ARC permissions in existence, and that the sum of all their fractions of a.data equals q_{out}.

The invariant $\mathcal{I}$ then enforces that the n in $\boxed{\mathsf{TotalCount}(n, q_{\mathrm{out}})}^{\gamma}$ matches the current physical value of a.counter, and that the remainder fraction (q_{in}) of a.data owned by $\mathcal{I}$ is precisely $1 - q_{\mathrm{out}}$.

Formally, these ghost assertions are built from Iris's Auth PCM [Jung et al. 2018b]: $\boxed{\mathsf{Count}(q)}^{\gamma} ::= \boxed{\circ\,(1, q)}^{\gamma}$ and $\boxed{\mathsf{TotalCount}(n, q)}^{\gamma} ::= \boxed{\bullet\,(n, q)}^{\gamma}$. The details of this construction are not important for the present discussion; the important point is that it establishes the soundness of various rules for manipulating these assertions, such as the following *ghost update* rule:

$$\boxed{\mathsf{TotalCount}(n, q_{\mathrm{out}})}^{\gamma} * \boxed{\mathsf{Count}(q)}^{\gamma} \Rrightarrow \boxed{\mathsf{TotalCount}(n+1, q_{\mathrm{out}})}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma} \quad \text{(Count-Ghost-Update)}$$

This rule allows us to split a single count into two by increasing the total count by 1, without changing the total fraction q_{out}. It is needed in the proof of clone: there, we need to produce two single counts $\boxed{\mathsf{Count}(q/2)}^{\gamma}$, for the two new ARC permissions, from the one single count $\boxed{\mathsf{Count}(q)}^{\gamma}$ of the original ARC permission, and this requires accessing the TotalCount predicate in $\mathcal{I}$ and applying Count-Ghost-Update.

However, there is a problem. Since clone uses a relaxed **FAA**, we will be using rule iRC11-CInv-FAA-Rlx to access the invariant $\mathcal{I}$, but if we instantiate that rule with $P ::= \boxed{\mathsf{Count}(q)}^{\gamma}$ and

$Q ::= \boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}$, what we obtain is:

$$\left\{\Delta_\pi \left(\boxed{\mathsf{Count}(q)}^{\gamma}\right)\right\}\ \mathbf{FAA_{rlx}}(\texttt{a.counter}, 1)\ @\ \pi\ \left\{\nabla_\pi \left(\boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}\right)\right\}$$

But what we really need is:

$$\left\{\boxed{\mathsf{Count}(q)}^{\gamma}\right\}\ \mathbf{FAA_{rlx}}(\texttt{a.counter}, 1)\ @\ \pi\ \left\{\boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}\right\}$$

This is where Ghost-Mod saves the day! We can immediately derive the latter Hoare triple from the former from the fact that both the pre and post are unsynchronized ghost state assertions.

In general, unsynchronized ghost state is useful for encoding objective, *view-agnostic* information about globally consistent properties of a data structure, such as (in the case of `Arc`) the total number of `Arc` permissions currently in existence. Fully relaxed (**`rlx`**) accesses suffice to update such view-agnostic state consistently, and the Ghost-Mod rule lets us exploit that. We hasten to add that this is not a novel observation—as Doko and Vafeiadis [2017] have already noted: "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."

3.3.4 Core `Arc` Fully Reclaimed. The FSL++ verification of Core `Arc` can only reclaim the `data` field, but not the `counter` field. In the proof of the last `drop`, Doko and Vafeiadis [2017] can only reclaim $\texttt{a.data} \mapsto -$, because `a.counter` is governed by a permanent invariant. Using iRC11 cancellable invariants, however, we can verify what Rust's `Arc` really does, which is to reclaim both fields.

Specifically, we create a cancellable invariant for `a.counter`, guarded by τ, and then treat $[\tau]_q$ analogously to $\texttt{a.data} \overset{q}{\mapsto} v$: wherever we use $\texttt{a.data} \overset{q}{\mapsto} v$ to represent shared ownership of `a.data`, we mirror that with the use of $[\tau]_q$ for the exact same q to represent shared ownership of `a.counter`. This proof extension is shaded in the definition of ARC (ARC-Own) and $\mathcal{I}$ (ARC-Inv). Therefore, when the last `drop` regains the full ownership of $\texttt{a.data} \mapsto -$, it also regains the full token $[\tau]_1$, enough to invoke iRC11-CInv-Cancel and reclaim ownership of $\texttt{a.counter} \mapsto -$. The whole region can then be safely deallocated. This example demonstrates that the changes needed for adapting uses of ordinary invariants to uses of cancellable invariants are modular and manageable.

4 MODELING CANCELLABLE INVARIANTS WITH SYNCHRONIZED GHOST STATE

In the previous section, we saw how iRC11's support for both cancellable invariants and user-defined ghost state played a crucial role in verifying data structures like Rust's `Arc`. As it turns out, ghost state also plays an important role in building a *model* for cancellable invariants—in particular, in defining the semantics of *invariant tokens*. However, in the relaxed-memory setting, it is *unsound* to model invariant tokens using the *unsynchronized* ghost state we have seen so far. In this section, we show how to soundly model iRC11 cancellable invariants instead using a novel construction of *synchronized* ghost state.

4.1 Deriving Unsoundness in a Naive Model with Unsynchronized Ghost State

Let us begin by showing why a naive model of iRC11's cancellable invariant tokens using unsynchronized ghost state cannot possibly work.

Suppose that it did—that is, suppose that invariant tokens were modeled purely as some form of unsynchronized ghost state assertion. If that were possible, it would mean that, according to Ghost-Mod, token assertions would be agnostic to the fence modalities: $[\tau]_q \Leftrightarrow \Delta_\pi\, [\tau]_q \Leftrightarrow \nabla_\pi\, [\tau]_q$. We will show that this in turn would enable us to spuriously verify the example given in Fig. 6. This example is exactly the same as Fig. 4b, except that there are no fences, and thus there is no

$$\{X \mapsto - * Y \mapsto -\} X := 0; Y := 0; \{X \mapsto 0 * Y \mapsto 0\} \left\{ [\tau]_1 * {}^{\tau}\boxed{X \mid \mathsf{True}} * \boxed{Y \mid \mathcal{I}^Y} \right\}$$

$\left\{ [\tau]_{1/2} * {}^{\tau}\boxed{X \mid \mathsf{True}} * \boxed{Y \mid \mathcal{I}^Y} \right\}$	$\left\{ [\tau]_{1/2} * {}^{\tau}\boxed{X \mid \mathsf{True}} * \boxed{Y \mid \mathcal{I}^Y} \right\}$
$\mathtt{FAA}_{\mathtt{rlx}}(X, 42);$	$\mathtt{if}\ \mathtt{FAA}_{\mathtt{rlx}}(Y, 2) == 1$
$\{[\tau]_{1/2}\}\ \{\Delta_{\pi_1}\ [\tau]_{1/2}\}$ *Unsound!*	$\{[\tau]_{1/2} * \nabla_{\pi_2}\ [\tau]_{1/2}\}\ \{[\tau]_1\}$ *Unsound!*
$\mathtt{FAA}_{\mathtt{rlx}}(Y, 1);\ \{\mathsf{True}\}$	$\{X \mapsto -\}\ \mathtt{free}(X);\ \{\mathsf{True}\}$

Fig. 6. Buggy example verified where $[\tau]_q$ is modeled with unsynchronized ghost state.

Raw-CInv-New
$$I \Rrightarrow \exists \tau.\, [\tau]_1 * {}^{\tau}\boxed{I}$$

Raw-CInv-Tok
$$[\tau]_{q+q'} \Leftrightarrow [\tau]_q * [\tau]_{q'}$$

Raw-CInv-Acc
$$\frac{\forall V_i.\, \{\lfloor I \rfloor_{\sqcup V_i} * P\}\ e\ @\ \pi\ \{v.\, \lfloor I \rfloor_{\sqcup V_i} * Q\} \qquad \text{atomic}(e)}{{}^{\tau}\boxed{I} \vdash \{[\tau]_q * P\}\ e\ @\ \pi\ \{v.\, [\tau]_q * Q\}}$$

Raw-CInv-Cancel
$${}^{\tau}\boxed{I} \vdash [\tau]_1 \Rrightarrow I$$

Fig. 7. Selected rules for RMM-sound raw cancellable invariants.

synchronization between thread 2's deallocation of X and the **FAA** on X performed by thread 1. As a violation of Free-Safe (see §2.2), this constitutes a data race and thus undefined behavior.

To spuriously verify this example, we use the exact same proof as Fig. 4b, except that instead of using the fence rules, we simply use Ghost-Mod to move the token $[\tau]_q$ across the fence modalities. *Note that this proof step is only possible because we assume the tokens are modeled with unsynchronized ghost state.* Then, when thread 2 regains the full token $[\tau]_1$, it uses iRC11-CInv-Cancel to obtain $X \mapsto -$. This, in turn, licenses the subsequent racy deallocation of X.

As we can see from this scenario, any model for cancellable invariant tokens based purely on unsynchronized ghost state violates a key safety guarantee:

An invariant's cancellation must happen-after all accesses to it. (Cancel-Safe)

4.2 Raw Cancellable Invariants

It is obvious that the invariant tokens need to be made *view-dependent*, so that Ghost-Mod does not apply to them and the example in Fig. 6 is not verifiable. It is less obvious, though, what a sound model of invariant tokens (guaranteeing Cancel-Safe) should look like.

Before we can answer that question, it is helpful to first simplify the problem. In this subsection, we present an important intermediate mechanism we call *raw cancellable invariants*. Raw cancellable invariants are sound for RMM but support a much simpler set of proof rules than iRC11 cancellable invariants do. In §4.3, we will then show how to construct a model of raw invariant tokens using synchronized ghost state. Finally, in §4.4, we will use raw cancellable invariants as the key stepping stone for building a model for the single-location cancellable invariants of iRC11.

Selected rules of raw cancellable invariants are given in Fig. 7. Like the previous forms of cancellable invariants we have seen, a raw cancellable invariant ${}^{\tau}\boxed{I}$ describes an invariant protected by an invariant token τ. However, the rules for raw cancellable invariants are much closer to those for Iris-SC's cancellable invariants than iRC11's. In particular, raw cancellable invariants are general invariants that are *not* restricted to a single location: they can contain arbitrary resources, and yet they are still sound in RMM. The invariant can be created with Raw-CInv-New if we can provide the initial invariant content I, and can be cancelled with Raw-CInv-Cancel if we have the full token $[\tau]_1$, after which we reclaim ownership of the content I. These rules are exactly the same as those of SC-sound cancellable invariants (Fig. 1).

The only rule that differs from Iris-SC's cancellable invariants is the access rule Raw-CInv-Acc. Recall that the access rule allows any thread to access and update the *invariant content* I during a single, atomic physical step (thus the side-condition atomic(e)).[5] The rule for accessing raw invariants is, however, significantly weakened from its SC counterpart (SC-CInv-Acc)—this is in order to keep raw invariants sound in RMM. Recall that in iRC11, assertions are to be thought of as *view predicates*. If the SC access rule SC-CInv-Acc were sound for raw cancellable invariants, it would mean that at any instant a thread π could assume that the shared resource backing up the invariant satisfied I at π's *local* view V_π, and it could also update the resource to one satisfying I at V_π. But clearly, since in RMM different threads have different views of memory, such reasoning is unsound. (It is only sound under SC because under SC all threads share the same view of memory.)

Since the *content view* V_i—*i.e.*, the view at which I is justified—can be constantly changed by different threads accessing the invariant, there is in general no relationship between threads' current local views and the content view V_i. Therefore, the access rule Raw-CInv-Acc cannot equate the view at which the content is provided with the current local view of the thread. The rule can only provide I protected by a new modality we call the *view-join modality*. To explain this modality, we must first review some basics of how assertions are modeled in iRC11.

Basics of modeling iRC11 assertions. To understand the model $[\![\cdot]\!]$ of iRC11 assertions in Iris, recall that $[\![\cdot]\!]$ is of type $\mathit{vProp} \to \mathit{View} \to \mathit{iProp}$, where vProp is the type of iRC11 assertions and iProp is the type of Iris assertions. The model $[\![\cdot]\!]$ interprets iRC11 assertions as view predicates in Iris, *i.e.*, for $P \in \mathit{vProp}$, $[\![P]\!] \in \mathit{View} \to \mathit{iProp}$.

Furthermore, it is crucial that these predicates be *view-monotone*, *i.e.*, that iRC11 assertions remain valid when the thread witnesses additional memory events. Formally, monotonicity means that if $V_1 \sqsubseteq V_2$, 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 local view. For this, the validity of P must be monotone in the thread's local view and, in general, be monotone with respect to any view.

In the following, we use blue font-face to distinguish Iris assertions against iRC11 assertions, which are in black font-face.

View-join modality. If $P \in \mathit{vProp}$ and $V_b \in \mathit{View}$, then the semantics of 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 thread's current view V and V_b.

The rule Raw-CInv-Acc uses this view-join modality to restore soundness of the access rule under RMM. Notice that the appearances of the invariant content I in the premise occur under the view-join modality $\lfloor I \rfloor_{\sqcup V_i}$. This means that, when we use Raw-CInv-Acc to open a raw cancellable invariant ${}^{\tau}\boxed{I}$, we only gain access to I at an *arbitrarily larger* view $V \sqcup V_i$, where V_i represents the view at which the invariant content is currently justified. During the access to I, the instruction e can update π's current view to a larger view V', so long as it returns the invariant content at the view $V' \sqcup V_i$. To explain why this weakened access rule is sound, we now delve into the details of synchronized ghost state and the model of raw invariants.

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

4.3 The Model of Raw Cancellable Invariants

The model of invariant tokens and raw cancellable invariants is as follows.

$$\llbracket [\tau]_q \rrbracket(V) ::= \exists V_{\mathrm{tok}}.\, \boxed{\mathsf{PartialV}(q, V_{\mathrm{tok}})}^{\tau} * V_{\mathrm{tok}} \sqsubseteq V \qquad \text{(Raw-CInv-Model-Tok)}$$

$$\llbracket {}^{\tau}\boxed{I} \rrbracket(V) ::= \boxed{\exists V_i.\, \boxed{\mathsf{PartialV}(1, V_i)}^{\tau} \vee (\llbracket I \rrbracket(V_i) * \boxed{\mathsf{FullV}(V_i)}^{\tau})} \qquad \text{(Raw-CInv-Model)}$$

Invariant tokens. First of all, invariant tokens $[\tau]_q$ are *view-dependent* assertions: even though owning a token $[\tau]_q$ means owning only the ghost element $\boxed{\mathsf{PartialV}(q, V_{\mathrm{tok}})}^{\tau}$, this ghost ownership is tied to the view V at which the assertion is interpreted through the *token view* V_{tok}. In particular, the ghost element $\boxed{\mathsf{PartialV}(q, V_{\mathrm{tok}})}^{\tau}$ records both the fraction q, which represents how much of the invariant this token owns, and the token view V_{tok}, which represents what this particular fractional token has *observed*, *i.e.*, what invariant accesses this fractional token has participated in. The model requires that V—the view at which the token is interpreted—has also at least observed what $[\tau]_q$ has observed: $V_{\mathrm{tok}} \sqsubseteq V$. Being view-dependent, $[\tau]_q$ therefore no longer enjoys Ghost-Mod, so the spurious verification in §4.1 is excluded.

Invariant assertions. The model of invariant assertions ${}^{\tau}\boxed{I}$ simply encodes the two possible states of the invariant: "active" or "cancelled". Thus it is an *Iris invariant* of a disjunction (see Raw-CInv-Model). The right-hand side of the disjunction encodes the active state, where the content I is still available in the invariant at some content view V_i. In the active state the Iris invariant also owns a ghost element $\boxed{\mathsf{FullV}(V_i)}^{\tau}$ that records the view V_i in the ghost state. The left-hand side of the disjunction encodes the cancelled state, which asserts ownership of the full *fractional* element $\boxed{\mathsf{PartialV}(1, V_i)}^{\tau}$. Note that the invariant assertion ${}^{\tau}\boxed{I}$ itself is view-agnostic: it ignores the view V at which it is asserted. The relation between the content view V_i and the token views V_{tok}'s is managed entirely by the ghost elements $\boxed{\mathsf{FullV}(V_i)}^{\tau}$ and $\boxed{\mathsf{PartialV}(q, V_{\mathrm{tok}})}^{\tau}$.

Synchronized ghost state. These view-dependent ghost elements are members of a "synchronized ghost state" instance that we build to model raw invariants. This ghost construction has two kinds of elements: (1) a unique element $\boxed{\mathsf{FullV}(V_f)}^{\tau}$ that is used to record the *full view* V_f, and (2) fractional elements $\boxed{\mathsf{PartialV}(q, V_p)}^{\tau}$ that are used to associate some *partial view* V_p with some fraction q. The ghost construction is built to maintain the following property:

> *The join of all partial views (the V_p's from all $\boxed{\mathsf{PartialV}(q, V_p)}^{\tau}$'s) is always equal to the full view V_f in $\boxed{\mathsf{FullV}(V_f)}^{\tau}$.* (Sync-Ghost)

This property guarantees that the partial view V_p of the full fractional element $\boxed{\mathsf{PartialV}(1, V_p)}^{\tau}$ is actually equal to the full view V_f of $\boxed{\mathsf{FullV}(V_f)}^{\tau}$: $V_p = V_f$. *The Sync-Ghost property is what we require for view-dependent ghost state to be synchronized ghost state.* By synchronized ghost state we mean any ghost construction that is built on the notion of *fractional observations*. That is, the ghost state has fractional elements that track the subjective observations of the threads the elements are tied to, and, most importantly, the full fractional element is guaranteed to have tracked all observations.

In the case of raw cancellable invariants, the observations are the views around which threads access and update the invariant content I. Intuitively, we record the view V_i of the invariant content I as the full view in $\boxed{\mathsf{FullV}(V_i)}^{\tau}$. The token view V_{tok} in the ghost ownership $\boxed{\mathsf{PartialV}(q, V_{\mathrm{tok}})}^{\tau}$ of some token $[\tau]_q$ tracks the changes to I made by each access that $[\tau]_q$ participated in. By Sync-Ghost, the full token view $V_{\mathrm{tok_full}}$ of the full token $[\tau]_1$ will thus be equal to the content view V_i. Consequently a thread owning $[\tau]_1$ must have observed all changes to the invariant content I.

Raw-CInv-Model-Sync
$$\boxed{\mathsf{FullV}(V_i)}^{\tau} * \boxed{\mathsf{PartialV}(q, V_{\text{tok}})}^{\tau} \Rightarrow V_{\text{tok}} \sqsubseteq V_i \land (q = 1 \Rightarrow V_{\text{tok}} = V_i)$$

Raw-CInv-Model-Update
$$\boxed{\mathsf{FullV}(V_i)}^{\tau} * \boxed{\mathsf{PartialV}(q, V_{\text{tok}})}^{\tau} \Rrightarrow \boxed{\mathsf{FullV}(V' \sqcup V_i)}^{\tau} * \boxed{\mathsf{PartialV}(q, V' \sqcup V_{\text{tok}})}^{\tau}$$

Raw-CInv-Model-Join
$$\boxed{\mathsf{PartialV}(q, V)}^{\tau} * \boxed{\mathsf{PartialV}(q', V')}^{\tau} \Rightarrow \boxed{\mathsf{PartialV}(q + q', V \sqcup V')}^{\tau} * q + q' \leq 1$$

Fig. 8. Selected properties (at Iris level) of the ghost construction for raw cancellable invariants.

To maintain Sync-Ghost, the ghost construction for $\boxed{\mathsf{PartialV}(q, V_{\text{tok}})}^{\tau}$ and $\boxed{\mathsf{FullV}(V_i)}^{\tau}$ admits the rules in Fig. 8. (We omit the details of how they are encoded, but for Iris enthusiasts the resource algebra from which they are drawn is Auth (Option (Frac × View$_{\sqcup}$)).) Raw-CInv-Model-Sync says that any token view V_{tok} is included in the content view V_i, and the full token view $V_{\text{tok_full}}$ of $[\tau]_1$ is exactly V_i. Raw-CInv-Model-Join requires that the fractions consistently cannot sum up to more than 1, and also allows us to join together partial token views of the fractions when we are recollecting them. Raw-CInv-Model-Update formalizes a restriction on how the ghost state can grow: We can update a token view V_{tok} by extending it with some V' *only if* we simultaneously update the content view V_i in the same way. This makes sure that every change in the full view V_i is accounted for by some token view V_{tok}, and thus Sync-Ghost is maintained.

Proving cancellation. To understand how the model works, we briefly present the proof of the cancellation rule Raw-CInv-Cancel. To prove a rule sound in the model, we first interpret the rule at the current view V of the thread π that applies the rule and then prove it in Iris. For Raw-CInv-Cancel, we need to prove the following in Iris:

$$[\![^{\tau}\boxed{I}]\!](V) * [\![[\tau]_1]\!](V) \Rrightarrow [\![I]\!](V)$$

The user of this rule provides us—the prover of the rule—with the full token $[\tau]_1$ at V, and we need to give the user back I at the same V. Looking at the interpretation of tokens, we effectively have the full fractional $\boxed{\mathsf{PartialV}(1, V_{\text{tok_full}})}^{\tau}$ with a token view $V_{\text{tok_full}} \sqsubseteq V$. We then open the Iris invariant and find a content view V_i and the two possibilities for the invariant state (see Raw-CInv-Model). If the invariant were in the cancelled state, we would have *two* full fractional $\boxed{\mathsf{PartialV}(1, -)}^{\tau}$ and Raw-CInv-Model-Join would give us contradiction from $1 + 1 \leq 1$. Thus the Iris invariant must be in the active state. By owning the full fraction, with Raw-CInv-Model-Sync we know that the thread's current view V must have observed all changes to the invariant content: $V \sqsupseteq V_{\text{tok_full}} = V_i$. With that, we now can take the content $[\![I]\!](V_i)$ out of the invariant and upgrade it to $[\![I]\!](V)$ for the user, because I is view-monotone. To finish the proof, we use $\boxed{\mathsf{PartialV}(1, V_{\text{tok_full}})}^{\tau}$ to switch the Iris invariant to the cancelled state and re-establish it. □

Proving access. The proof outline of the access rule Raw-CInv-Acc is shown in Figure 9. The proof is an application of standard Iris proof rules. As in cancellation, we first open the invariant and deduce that it must be in the active state. Then, we apply the rule of consequence which leaves us with two side goals that we discuss below. Finally, we use framing to arrive at our premise.

As indicated in the outline, we have two goals to prove for the application of the rule of consequence: (1) prove that from the pre-condition of the conclusion we can obtain the pre-condition of the premise and (2) from the post-condition of the premise, we can obtain the post-condition of the conclusion. In task (1), our goal is $[\![I]\!](V) \Rrightarrow [\![\lfloor I \rfloor_{\sqcup V_i}]\!](V)$, *i.e.*, $[\![I]\!](V) \Rrightarrow [\![I]\!](V \sqcup V_i)$, which

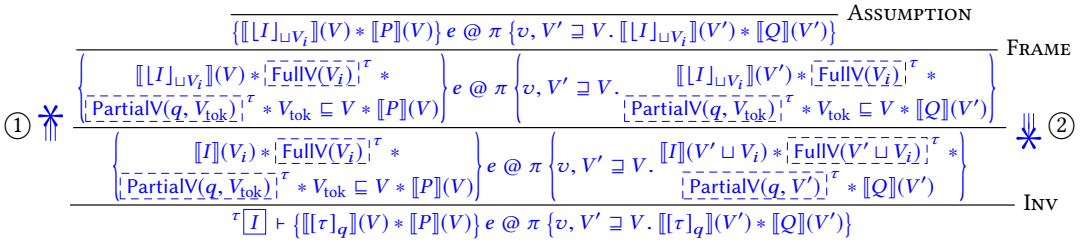


Fig. 9. Proof outline of Raw-CInv-Acc (at Iris level).

follows from view monotonicity. In task (2), our goal is

$$\boxed{\mathsf{FullV}(V_i)}^{\tau} * \boxed{\mathsf{PartialV}(q, V_{\mathsf{tok}})}^{\tau} * V_{\mathsf{tok}} \sqsubseteq V \Rrightarrow \boxed{\mathsf{FullV}(V' \sqcup V_i)}^{\tau} * \boxed{\mathsf{PartialV}(q, V')}^{\tau}$$

This proof step amounts to an application of Raw-CInv-Model-Update where we make use of the fact that $V' \sqcup V_{\mathsf{tok}} = V'$. This follows from $V_{\mathsf{tok}} \sqsubseteq V \sqsubseteq V'$. □

4.4 The Model of iRC11 Single-Location Cancellable Invariants

The access rule for raw cancellable invariants, Raw-CInv-Acc, may seem overly restrictive: due to the use of the view-join modality in the premise, the invariant content is only made accessible at an *arbitrarily large* view $V \sqcup V_i$, where V is the thread's local view and V_i is *unknown*. How, the reader may wonder, can that be useful? To illustrate how this apparent limitation is not really a problem at all, we now give a high-level explanation of how raw cancellable invariants can be used to model iRC11 single-location cancellable invariants. (For purposes of presentation, we abstract away here from certain gory details of the construction—largely inherited from Kaiser et al. [2017]—that are orthogonal to the goal of supporting safe cancellation.)

$$[\![\mathsf{ATOM}(\ell, \mathcal{I})]\!](V_i) ::= \exists h.\, \mathsf{History}(\ell, h) * (\forall m \in h.\, m.\mathsf{view} \sqsubseteq V_i) * \mathop{✳}_{m \in \mathsf{Frontier}(h)} \left([\![\mathcal{I}(m.\mathsf{val})]\!](m.\mathsf{view})\right) * \ldots$$

$$^{\tau}\boxed{\ell \mid \mathcal{I}} ::= {}^{\tau}\boxed{\mathsf{ATOM}(\ell, \mathcal{I})}$$

(iRC11-CInv-Model)

The iRC11 cancellable invariant assertion $^{\tau}\boxed{\ell \mid \mathcal{I}}$ is nothing more than a raw cancellable invariant (guarded by τ) whose invariant content is $\mathsf{ATOM}(\ell, \mathcal{I})$, an Iris view-predicate with three parts:

(1) The first component of ATOM—$\mathsf{History}(\ell, h)$—asserts ownership of the entire history h of writes to ℓ. This makes it the owner of location ℓ and enforces that all accesses to ℓ will need to go through the raw cancellable invariant.
(2) The second component of ATOM is the knowledge that the view V_i at which the invariant is interpreted contains *all* message views in ℓ's history. (This means in particular that V_i is synchronized with all writes to ℓ.)
(3) The third component of ATOM is the invariant content $[\![\mathcal{I}(m.\mathsf{val})]\!](m.\mathsf{view})$ for every message m in $\mathsf{Frontier}(h)$, *i.e.*, every message corresponding to a write to ℓ that has not been read from by an RMW operation (like **FAA**). (For messages not in the frontier, ownership of the invariant content may have been transferred away from ATOM by an RMW operation.)

Crucially, the ATOM predicate uses the view V_i (at which it is interpreted) exclusively as an *upper bound* on the message views for the writes to ℓ, a property that is crucial for safe *cancellation*. The remaining parts of ATOM are either view-agnostic (in the case of History) or they hold at views that end up synchronizing with some component of the thread's local view when a physical

memory operation is performed. This allows us to all but ignore V_i when *accessing* the invariant, thus avoiding the apparent problem of how to reason about the view-join modality in Raw-CInv-Acc.

Proving cancellation. Let us now see how to prove iRC11-CInv-Cancel. Suppose we invoke the rule for thread π whose *current* view is V_π. Since we own $[\tau]_1$, we can immediately use Raw-CInv-Cancel to cancel the underlying raw invariant, thus reclaiming ownership of $[\![\mathsf{ATOM}(\ell, \mathcal{I})]\!](V_\pi)$. By the definition of ATOM, we now own the history of writes to ℓ, and we know that V_π has seen all writes to ℓ—including the most recent write of value v with message m—so we can assert $\ell \mapsto v$ at thread π's current view V_π. Furthermore, we now also own the invariant interpretation $[\![\mathcal{I}(v)]\!](m.\mathsf{view})$, because certainly m is in the frontier of ℓ's history. Since we know that $m.\mathsf{view}$ is contained in the thread view V_π, we can assert $\mathcal{I}(v)$ holds at V_π as well. □

Proving access. We now turn to the proof of iRC11-CInv-FAA-Rlx, the iRC11 rule for relaxed **FAA** operations. Consider an execution of $\mathsf{FAA}(\ell, n)$ in thread π with local views $\mathcal{V}$ before and $\mathcal{V}'$ after the instruction, reading message m_r (with value v) and writing message m_w (with value $v + n$). In the proof of iRC11-CInv-FAA-Rlx, we access the raw invariant ${}^{\tau}\boxed{\mathsf{ATOM}(\ell, \mathcal{I})}$ using Raw-CInv-Acc. This gives us $[\![\lfloor \mathsf{ATOM}(\ell, \mathcal{I})\rfloor_{\sqcup V_i}]\!](\mathcal{V}.\mathsf{cur})$—*i.e.*, $[\![\mathsf{ATOM}(\ell, \mathcal{I})]\!](\mathcal{V}.\mathsf{cur} \sqcup V_i)$, for some unknown V_i.

As suggested above, the fact that V_i is unknown is not actually a problem: as we now explain, the proof of the ownership transfer at the heart of iRC11-CInv-FAA-Rlx does not care about V_i at all—it only cares about specific messages and their interpretations.

We start out with $[\![P]\!](\mathcal{V}.\mathsf{rel})$ from our precondition and $[\![\mathcal{I}(v)]\!](m_r.\mathsf{view})$ from ATOM (since m_r is in the frontier—it must be so because *we* are reading from it, which means it could not have been read from by an RMW before). By view monotonicity, we therefore have $[\![P * \mathcal{I}(v)]\!](\mathcal{V}.\mathsf{rel} \sqcup m_r.\mathsf{view})$. Applying the premise of iRC11-CInv-FAA-Rlx to this, we obtain $[\![\mathcal{I}(v+n) * Q(v)]\!](\mathcal{V}.\mathsf{rel} \sqcup m_r.\mathsf{view})$. From that we need to establish (1) $[\![\mathcal{I}(v+n)]\!](m_w.\mathsf{view})$ to close the invariant and (2) $[\![Q(v)]\!](\mathcal{V}'.\mathsf{acq})$ for the postcondition. To justify these steps, it suffices to observe that (*i*) m_w's message view is precisely $\mathcal{V}.\mathsf{rel} \sqcup m_r.\mathsf{view}$ extended to include the new write to ℓ, and (*ii*) after executing the **FAA** operation, π's updated acquire view ($\mathcal{V}'.\mathsf{acq}$) includes both its original release view ($\mathcal{V}.\mathsf{rel}$) and the view of the read message ($m_r.\mathsf{view}$). We thus obtain (1) and (2) immediately by view monotonicity.

Notice how the unknown V_i did not come up even once in the last paragraph! Indeed, V_i is only relevant to the final step where we have to re-establish ATOM at $\mathcal{V}'.\mathsf{cur} \sqcup V_i$ which we call V_i'. The crucial part here lies in showing $V_i' \sqsupseteq m_w.\mathsf{view}$, *i.e.*, making sure that V_i' is synchronized with the view of the new write message. It is easy to check that V_i' contains $m_r.\mathsf{view}$ (through $m_r.\mathsf{view} \sqsubseteq V_i$) and $\mathcal{V}.\mathsf{rel}$ (through $\mathcal{V}.\mathsf{rel} \sqsubseteq \mathcal{V}.\mathsf{cur} \sqsubseteq \mathcal{V}'.\mathsf{cur}$). This leaves the write itself. As an RMW operation, **FAA** records the new write in the updated view, $\mathcal{V}'.\mathsf{cur}$. Hence, $V_i' \sqsupseteq m_w.\mathsf{view}$ and we can re-establish ATOM at V_i'. □

5 OTHER CONTRIBUTIONS

In this paper, we have focused on our central technical contribution of synchronized ghost state. However, $\mathbf{RB_{rlx}}$ encompasses a number of other contributions, a few of which we highlight in this section: ORC11's race detector (§5.1), a model of fence modalities (§5.2), the verification of `Arc` and the bug we found in its implementation (§5.3), and, finally, the adaptation of RustBelt's lifetime logic to RMM (§5.4).

5.1 Data Race Detection in ORC11

ORC11 is the first operational semantics that incorporates a race detector for non-atomic accesses into a language with release-acquire accesses, relaxed accesses, and fences. ORC11's race detector extends the race detector Kaiser et al. [2017] developed for iGPS, in order to address the extra effects

of relaxed accesses. To explain the necessity of this extension, we first discuss why the approach of Kaiser et al. [2017] does not scale to relaxed accesses.

The iGPS race detector, introduced by Kaiser et al. [2017] for the release-acquire/non-atomic fragment of C11, is somewhat unusual in that it does not in fact detect all races in *every* execution. (Recall that, by C11, two accesses to the same location are racy if one of them is non-atomic, one is a write, and there is no happens-before relation between them.) Instead, although iGPS forbids write-before-read races—that is, races where a write is interleaved before a racing read—it *allows* read-before-write races—where a read is interleaved before a racing write. To illustrate this asymmetry, consider the following example code:

$$X := 0;$$
$$^*X \;||\; X := 37$$

In this program there are two possible interleavings, both of which are considered racy by C11. The iGPS race detector detects a race in the interleaving where the read from X is executed after the write to X, but it does not detect a race in the interleaving where X is read first.

The upside of iGPS's approach is that reads do not need to be tracked by the race detector, which reduces the amount of state in the operational semantics. The downside, of course, is that some races are not detected—a seemingly rather severe problem for a race detector! The reason this is not a problem in iGPS is that Hoare triples imply absence of races for *all* executions of a program. In order to be able to claim that the iGPS *logic* ensures absence of data races according to C11, it thus suffices for the race detector in the operational semantics to detect a race on *some* execution of every program that is racy according to C11. And indeed it does: for programs with only release-acquire and non-atomic accesses (the domain of iGPS), for any execution with a read-before-write race, there is always a differently interleaved execution with a write-before-read race, which iGPS's race detector will detect.

In the presence of relaxed accesses, however, the iGPS race detector is no longer sufficient, because the property mentioned above is no longer true. That is, it is possible to construct programs that have executions in which the read-before-write races happen, but there is no interleaving where the write will be executed before the read. For example, consider the following program:

$$X := 0; Y := 0;$$
$$^*X; Y :=_{\mathtt{rlx}} 1 \;||\; \mathbf{while}\,(^{*\mathtt{rlx}}Y == 0); X := 37$$

Here, the non-atomic read in the left thread is guaranteed to be executed before the non-atomic write to X in the right thread, and there is no interleaving where the reverse can happen. The iGPS race detector would *not* declare this program racy, but the two accesses to X are not related by happens-before and are thus considered a race by C11.

To account for such programs, we extend iGPS's race detector, which already tracked non-atomic writes, to track *all* access events, including atomic writes, and atomic and non-atomic reads in threads' local views. These events will then be sent across threads when they perform synchronization. The race detector view $V_{\mathbf{Race}}$ of the global state (§2.2) then records, for every location, the latest non-atomic write, a global set of atomic writes, a global set of atomic reads, and a global set of non-atomic reads. When a thread π is performing an access a, depending on the kind of access a, the ORC11 race detector will then require that the thread π must have observed certain events in the global race detector view $V_{\mathbf{Race}}$.

In the example above, the non-atomic read of X by the left thread, when executed, will add a fresh read event $\varepsilon_{\mathbf{na}}$ into $V_{\mathbf{Race}}$'s set of non-atomic reads for X. The non-atomic write of X by the right thread is guaranteed to be executed after the read by the left thread. However, when the write is executed, the race detector requires that the right thread must have observed in its local view *all* read events, including $\varepsilon_{\mathbf{na}}$, in order to be deemed non-racy. Since the right thread did not

synchronize with the left thread to obtain $\varepsilon_{\mathsf{na}}$ in its local view, its write to X will be declared racy by the ORC11 race detector.

To show a formal correspondence between ORC11 and RC11 [Lahav et al. 2017], we exploit the fact that ORC11 is very close to the "promise-free" fragment of Kang et al. [2017] extended with non-atomics and a race detector. Kang et al. [2017] already proved a formal correspondence between their promise-free fragment and C11. Building on their result, we show (on paper) that any racy execution in RC11 can be replayed as a racy execution in ORC11 [Dang et al. 2019]. The proof is relatively straightforward since ORC11 explicitly tracks read and write events.

5.2 The Model of Fence Modalities

In §3.3, we argue that the Ghost-Mod rule is sound without actually giving the model of unsynchronized ghost state nor the fence modalities. While the model of unsynchronized ghost state can be easily given as

$$[\![\boxed{a}^{\gamma}]\!](V) ::= \lfloor a \rfloor^{\gamma} \qquad \text{(USGS-Model)}$$

it is rather technically tricky to get a simple and clean model of the fence modalities. In this section we briefly discuss how the model of fence modalities works.

In iGPS, a thread's subjective view is a simple view of type *View*, and consequently an iGPS assertion is interpreted as a function of type $\mathit{View} \to \mathit{iProp}$. In iRC11, even though a thread's subjective view is a triple of three views, we still managed keep the interpretation $[\![\cdot]\!]$ of an iRC11 assertion as simply a function of type $\mathit{View} \to \mathit{iProp}$. That may seem rather suspicious, for as we mentioned in §3.1, the release and acquire views of a thread π are needed to model the release Δ_π and acquire ∇_π modalities, respectively. If $[\![\cdot]\!]$ is only a function of type $\mathit{View} \to \mathit{iProp}$, then, assuming that by default the view V supplied to $[\![\cdot]\!](V)$ is the thread's current view, how can $[\![\cdot]\!]$ get access to the release or acquire view that it needs in order to interpret the fence modalities?

To model the fence modalities, we exploit extra ghost state that enables the interpretation $[\![\cdot]\!]$ to gain indirect access to the thread π's physical release view and acquire view:

$$[\![\Delta_\pi\, P]\!](V) ::= \exists V_{\mathtt{rel}}.\, \boxed{\mathrm{RelV}(V_{\mathtt{rel}})}^{\pi} * [\![P]\!](V_{\mathtt{rel}}) \qquad \text{(RelMod-Model)}$$

$$[\![\nabla_\pi\, P]\!](V) ::= \exists V_{\mathtt{acq}}.\, \boxed{\mathrm{AcqV}(V_{\mathtt{acq}})}^{\pi} * [\![P]\!](V_{\mathtt{acq}}) \qquad \text{(AcqMod-Model)}$$

Here, $\boxed{\mathrm{RelV}(V_{\mathtt{rel}})}^{\pi}$ and $\boxed{\mathrm{AcqV}(V_{\mathtt{acq}})}^{\pi}$ are elements of a view-dependent ghost state instance for the thread π. These are governed by a global invariant, which enforces that they always record *snapshots* of π's release and acquire views, *i.e.*, $V_{\mathtt{rel}} \sqsubseteq \mathcal{V}.\mathrm{rel}$ and $V_{\mathtt{acq}} \sqsubseteq \mathcal{V}.\mathrm{acq}$, where $\mathcal{V}$ is π's local view. (For Iris enthusiasts, these are the snapshot elements in the Master-Snapshot monoid.) Thus, $[\![\Delta_\pi\, P]\!](V)$ (respectively, $[\![\nabla_\pi\, P]\!](V)$), ignoring the view V at which it is being interpreted, states that P holds at some $V_{\mathtt{rel}} \sqsubseteq \mathcal{V}.\mathrm{rel}$ ($V_{\mathtt{acq}} \sqsubseteq \mathcal{V}.\mathrm{acq}$). By view-monotonicity this means also that P holds at $\mathcal{V}.\mathrm{rel}$—the release view of π (respectively, $\mathcal{V}.\mathrm{acq}$—the acquire view of π).

Soundness of Ghost-Mod. With these definitions, it is now clear how Ghost-Mod is sound. For example, the direction "$\Delta_\pi \lfloor a \rfloor^{\gamma} \Rrightarrow \lfloor a \rfloor^{\gamma}$" is trivial after unfolding the model of the release modality. The direction "$\boxed{a}^{\gamma} \Rrightarrow \Delta_\pi \lfloor a \rfloor^{\gamma}$" is also essentially trivial because it is always possible to create a snapshot $\boxed{\mathrm{RelV}(\mathcal{V}.\mathrm{rel})}^{\pi}$ of the thread's release view $\mathcal{V}.\mathrm{rel}$. □

5.3 Verification of Relaxed-Memory Libraries

Adapting RustBelt to ORC11 required us to re-verify all the internally unsafe libraries considered by RustBelt, *i.e.*, to show that these libraries still properly encapsulate their unsafe behaviors within *safe* interfaces when we consider their real relaxed-memory implementations. **RB$_{\mathtt{rlx}}$** has ported all verifications done in RustBelt, including the following concurrency libraries: `thread::spawn`,

```
is_unique(a) := if CAS_acq(a.weak, 1, −1) then
                  let uniq = (*^acq a.strong == 1) in
                  a.weak :=_rel 1;
                  uniq
                else false

get_mut(a) := if is_unique(a) then
                Some (a.data)
              else None
```

Fig. 10. The implementation of `Arc::get_mut`.

`rayon::join`, `Mutex`, `RwLock`, and `Arc`. (The sequential libraries—`Rc`, `Cell`, `RefCell`—remain largely unchanged from RustBelt.) We briefly discuss the most challenging verification effort in $\mathbf{RB_{rlx}}$: the full `Arc` library (as opposed to the Core `Arc` discussed in §3.3).

`Arc<T>` and `Weak<T>`. `Arc<T>` has a sibling library `Weak<T>` which is also an atomic reference counter. The `Weak` type, being very similar to `Arc`, counts how many `Weak` pointers are in existence, and also comes with its own `clone` and `drop` functions. However, while owning an `Arc` guarantees access to the underlying object of type `T`, owning a `Weak` does not prevent the underlying object from being reclaimed. In order to access the object with a `Weak` pointer, one first calls `Weak::upgrade` to upgrade the `Weak` pointer to an `Arc` pointer. `Weak::upgrade` can fail when the object has already been reclaimed, that is, when there is no `Arc` pointer left. A `Weak` pointer is typically created by calling `Arc::downgrade` on a shared reference of `Arc`.

The challenge in verifying `Arc` and `Weak` in RMM is that they are implemented together by two tightly coupled atomic locations—one for each counter, and reasoning about the relation between multiple locations in RMM is known to be complicated [Turon et al. 2014]. The interplay of the two counters is further complicated by the library's support for temporarily reclaiming full ownership of the underlying content when the thread knows it owns the last unique `Arc` and `Weak` pointers. The functions `Arc::get_mut` (see Figure 10) and `Arc::make_mut` provide these capabilities: they return a mutable reference `&mut T` to the underlying content. The reclamation is temporary because when the reference goes out of scope (when the lifetime of the mutable reference ends), the content is returned and the original `Arc` pointer can be used again.

In $\mathbf{RB_{rlx}}$, we need to use two separate iRC11 cancellable invariants for each counter. At the same time, we employ both unsynchronized and synchronized ghost state to maintain the intricate relation between the two counters (see the appendix [Dang et al. 2019] for more details).

Insufficient Synchronization in `Arc<T>::get_mut`. Unfortunately, our proof setup was not strong enough to verify `Arc` and `Weak` without change (although at least partly for good reason!). The read of the `Arc` counter `a.strong` in `is_unique` (used by `get_mut`, Fig. 10), as well as another read implemented in `make_mut` for the similar role, were **`rlx`** in the original code and we had to strengthen them both to **`acq`** in order to make the verification go through. The reason is that, while we managed to temporarily get the full resources out by a read, the **`rlx`** reads do not give us those resources at the thread's current view (they are under a ∇ modality). We conjecture that the **`rlx`** read in `make_mut` is in fact sufficient, but the **`rlx`** read in `get_mut` turned out to be *insufficient*. The bug has been reported and fixed in the Rust codebase [Jourdan 2018].

Fig. 11 shows an example where a data race (according to C11) can arise when using `get_mut` in otherwise safe code. Here, there are two **`na`** operations: the read of the underlying integer in line 3 (child thread) and the write to the same integer in line 6 (parent thread). The read should be safe because the child thread owns `arc2` and the underlying integer is shared and *immutable*. The write should be safe because a successful `get_mut` gives the parent (who owns `arc1`) temporary full non-atomic access to the integer. This can only happen after the child thread finishes and `arc2`

```
1  let mut arc1 = Arc::new(0);
2  let     arc2 = Arc::clone(&arc1);
3  thread::spawn( move || { let u : u32 = *arc2; ... /* drop(arc2); */ } );
4  loop { match Arc::get_mut(&mut arc1) {
5           None => {}
6           Some(m) => { *m = 1u32; return; }}}
```

Fig. 11. Example demonstrating the bug we found in `Arc`.

has been `drop`-ed. However, the two **na** operations constitute a data race according to C11 because neither one happens-before the other. More specifically, in the child thread, when `arc2` goes out of scope it will be destructed by `Arc::drop` (line 3), which uses a release (**rel**) **FAA** (see the code in Fig. 5). This release **FAA** will be read by `get_mut` (in the second check of `is_unique`, Fig. 10) in the parent thread (line 4). If this read had been **acq**, then there would have been a release-acquire synchronization between `drop` and `get_mut`, and the **na** read of the child thread would have been guaranteed to happen-before the **na** write of the parent thread. However, the read was **rlx**, and thus no synchronization can be established between the two **na** operations.

5.4 Adapting the Lifetime Logic to Relaxed Memory

RustBelt's semantic soundness proof for the Rust type system depends on the *lifetime logic*. The lifetime logic provides so-called *borrow propositions*, a family of mechanisms similar to *raw* cancellable invariants (see §4.2) that are instrumental in handling various features required by Rust's type system and by the verification of Rust libraries.

Borrow propositions improve on raw cancellable invariants in multiple ways, the most important being that they decouple the assertions used to access the underlying resource from the assertions used to reclaim it. This is achieved by offering three kinds of assertions: *borrow propositions* (of which there are several different kinds), *lifetime tokens* (similar to invariant tokens), and an *inheritance assertion*. Borrow propositions and (fractions of) lifetime tokens are used together to gain temporary access to the underlying resource. Separately, the inheritance assertion can be used together with the full lifetime token to reclaim the contents of the borrow.

These assertions can be owned by different threads and if we were to model them using unsynchronized ghost state we would run into the same problems as in §4.1. (A full counterexample is presented in Section 4 of Dang et al. [2019].) Thus, we make essential use of synchronized ghost state in the model of lifetime tokens, just as we did for cancellable invariant tokens.

Fortunately, despite this change to the model of the logic, the lifetime logic's proof rules are almost entirely sound in ORC11. The only feature that required changes is atomic borrows (see below). However, atomic borrows are only used in the verification of concurrent libraries and *not* in the soundness proof of the type system because Rust's type system does not know anything about concurrency. The proof of safety of the λ_{Rust} type system thus did not need to be changed.

Atomic borrows. The lifetime logic offers several kinds of borrow propositions, which represent different ways of managing ownership of resources. Only one of them, *atomic borrows*, allows shared atomic access to the underlying resource.

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

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

Under SC, accesses to atomic borrows were automatically synchronized. This is no longer true in RMM. Consequently, we apply the same technique used for raw cancellable invariants to weaken the access rule, providing access to the underlying resource P only under the view-join modality.

Other borrows. The proof rules for other kinds of borrows did not need changing because these borrows always ensure proper synchronization between accesses. As an example, consider *full borrows*. Full borrows can only ever be owned by one thread and whichever thread owns the full borrow assertion is the only one that can access the underlying resource.

Owing to this exclusiveness, full borrows enjoy a particularly strong access rule shown below. Concretely, LftL-full-acc states that the full borrow assertion $\&^{\kappa}_{\mathrm{full}} P$ can be exchanged for the underlying resource P at the thread's view together with a return predicate $\mathsf{Ret}(\kappa, P, q)$, which can then be used to return P in exchange for $\&^{\kappa}_{\mathrm{full}} P$. Note that access to P is not restricted to an atomic step of execution. Instead, P can be returned at any later point by applying LftL-full-ret.

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

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

To establish the soundness of these proof rules, we require an additional instance of synchronized ghost state on top of the synchronized ghost state used for lifetime tokens. The purpose of this extra ghost state is to maintain that that all accesses to the borrowed content P are fully synchronized. In particular, under the hood, P is held in an invariant at a content view V_i together with a piece of ghost state that records V_i. The borrow assertion $\&^{\kappa}_{\mathrm{full}} P$ carries its own synchronized ghost state as well, satisfying the invariant that the view V of the thread that owns $\&^{\kappa}_{\mathrm{full}} P$ must *include* the view V_i of the borrowed content. This allows us to deduce that it is safe for the thread that owns $\&^{\kappa}_{\mathrm{full}} P$ to gain access to P at its own local view, as required to prove soundness of LftL-full-acc.

6 RELATED AND FUTURE WORK

Program logics and verifications. He et al. [2018] present GPS+, a extension to GPS that supports relaxed accesses and release/acquire fences. However, their logic lacks support for relaxed **CAS**/**FAA** operations which `Arc` uses. Additionally, their logic is based on the axiomatic semantics of C11 which cannot be used together with Iris and thus would not allow us to extend the existing RustBelt development. GPS+ also does not support mechanized verification of programs.

iGPS [Kaiser et al. 2017] supports a mechanism called *fractional protocols*, which is closely related to cancellable invariants. However, iGPS's fractional protocols are not as powerful as iRC11's cancellable invariants in that they cannot reclaim the resources governed by the protocol at the thread's local view. This is because the protocol tokens they use are modeled with unsynchronized ghost state. By using synchronized ghost state, we can support full reclamation of all resources governed by either iRC11 cancellable invariants or borrow propositions.

[Tassarotti et al. 2015] use GPS to verify an implementation of the Read-Copy-Update (RCU) technique. With GPS, they are able verify the reclamation of clients' non-atomic locations, but not RCU's internal atomic locations because they are governed by GPS's *non*-cancellable protocols. [Kaiser et al. 2017] fixed this problem by re-verifying RCU in iGPS with their fractional protocols. However, as mentioned above, fractional procotols are not a general solution.

Gotsman et al. [2007] provide an SC-based logic where ownership of a location can be turned into a lock with fractional permissions for shared accesses, and later when the full permission of the lock is collected, the ownership of the location can be reclaimed. Hobor et al. [2008] provide a similar mechanism that additionally allows attaching "invariant resources" to locks. Our cancellable invariants are more general than these mechanisms in that our cancellable invariants are not specifically tied to locks and are proven sound with respect to a much weaker memory model.

Doko and Vafeiadis [2017] verify a subset of the `Arc` library with FSL++. 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.

Operational semantics for relaxed memory. Podkopaev et al. [2016] develop an operational account of a subset of C11 that includes relaxed accesses and non-atomics. However, it lacks support for fences and thus could not be used as is (to build a logic) to verify `Arc`. Their semantics also does not forbid the data race in Section 5.1.

Doherty et al. [2019] develop an operational semantics based on event graphs for the release/acquire/relaxed fragment of RC11. They also develop an invariant-based logic geared towards automated verification for programs in that fragment. As their operational semantics supports neither non-atomics nor fences, it is not expressive enough to handle the Rust libraries targeted by $\mathbf{RB_{rlx}}$.

Kang et al.'s *promising semantics* [Kang et al. 2017] 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. [2018] introduce the first program logic for the promising semantics. Their logic is based on RSL [Vafeaidis and Narayan 2013] 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 iRC11 currently do not support SC accesses and SC fences. In fact, we are not aware of any existing RMM separation logic that does.[6] 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 [2005], as well as epoch-based resource reclamation schemes such as that implemented by Rust's `crossbeam` library [Turon 2016].

ACKNOWLEDGMENTS

We would like to thank Jeehoon Kang, Ori Lahav, and Viktor Vafeiadis for their suggestions on building the race detector of ORC11. We would also like to thank Ralf Jung and Robbert Krebbers for various discussions on the original RustBelt development, as well as for their maintanance effort on both Iris and $\mathbf{RB_{rlx}}$. Finally, we would like to thank the anonymous reviewers from both PLDI 2019 and POPL 2020 for their constructive suggestions concerning presentation.

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).

[6] The abstract of the RSL paper [Vafeaidis and Narayan 2013] claims that it supports reasoning about SC accesses, but according to Vafeiadis [personal communication], this is a mistake, and indeed the body of the paper does not.

REFERENCES

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *TOPLAS* 32, 3 (2010), 1–67.

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

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

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. (2005), 259–270. https://doi.org/10.1145/1040305.1040327

John Boyland. 2003. Checking interference with fractional permissions. In *SAS (LNCS)*. https://doi.org/10.1007/3-540-44898-5_4

David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *SPAA*. 21–28. https://doi.org/10.1145/1073970.1073974

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. Appendix and Coq development accompanying this paper. http://plv.mpi-sws.org/rustbelt/rbrlx/

Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2019. Verifying C11 Programs Operationally. In *PPoPP*. 355–365. https://doi.org/10.1145/3293883.3295702

Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *VMCAI (LNCS)*. Springer, 413–430.

Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *ESOP*.

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

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *APLAS*. 19–37.

Mengda He, Viktor Vafeiadis, Shengchao Qin, and João F. Ferreira. 2018. GPS+: Reasoning About Fences and Relaxed Atomics. *International Journal of Parallel Programming* 46, 6 (2018), 1157–1183. https://doi.org/10.1007/s10766-017-0518-x

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP*. 353–367.

Jacques-Henri Jourdan. 2018. Insufficient synchronization in Arc::get_mut. Rust issue #51780, https://github.com/rust-lang/rust/issues/51780.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *PACMPL* 2, POPL, Article 66 (2018).

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. (2018). To appear in Journal of Functional Programming.

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.

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.

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

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. https://doi.org/10.1145/3009837.3009855

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

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.

Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. 2016. Operational Aspects of C/C++ Concurrency. *CoRR* abs/1606.01400 (2016). arXiv:1606.01400 http://arxiv.org/abs/1606.01400

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

Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A Separation Logic for a Promising Semantics. In *ESOP*. 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying read-copy-update in a logic for weak memory. In *PLDI*. 110–120. https://doi.org/10.1145/2737924.2737992

Aaron Turon. 2016. Crossbeam: Support for concurrent and parallel programming. Available at https://github.com/aturon/crossbeam.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *OOPSLA*. ACM, 691–707.

Viktor Vafeaidis and Chinmay Narayan. 2013. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *OOPSLA*.