

An approach for modularly verifying the core of Rust's atomic reference counting algorithm against the (Y)C20 memory consistency model

Bart Jacobs and Justus Fasse

KU Leuven, Department of Computer Science, DistriNet Research Group, Leuven, Belgium

ABSTRACT

We propose an approach for modular verification of programs that use relaxed-consistency atomic memory access primitives and fences. The approach is sufficient for verifying the core of Rust's Atomic Reference Counting (ARC) algorithm. We first argue its soundness, when combined with a simple static analysis and admitting an open sub-problem, with respect to the C20 memory consistency model. We then argue its soundness, even in the absence of any static analysis and without any assumptions, with respect to YC20, a minor strengthening of XC20, itself a recently proposed minor strengthening of C20 that rules out out-of-thin-air behaviors but allows load buffering. In contrast to existing work on verifying ARC, we do not assume acyclicity of the union of the program-order and reads-from relations. We define an interleaving operational semantics, prove its soundness with respect to (Y)C20's axiomatic semantics, and then apply any existing program logic for fine-grained interleaving concurrency, such as Iris.

KEYWORDS Relaxed Memory Consistency, Separation Logic, Modular Verification, Rust, Atomic Reference Counting.

1. Introduction

Most work on modular verification of shared-memory multi-threaded programs so far (e.g. (Jung et al. 2015, 2018)) has assumed *sequential consistency*, i.e. that in each execution of the program, there exists some total order on the memory accesses such that each read access yields the value written by the most recent preceding write access in this total order. However, for performance reasons, many real-world concurrent algorithms, such as Rust's Atomic Reference Counting (ARC) algorithm¹, use *relaxed-consistency* memory accesses that do not respect such a total order.

Consider the three *litmus tests* LBD, LB, and LBF in Fig. 1a, small concurrent programs accompanied by a precondition

JOT reference format:

Bart Jacobs and Justus Fasse. *An approach for modularly verifying the core of Rust's atomic reference counting algorithm against the (Y)C20 memory consistency model*. Journal of Object Technology. Vol. 25, No. 1, 2026.

Licensed under Attribution 4.0 International (CC BY 4.0)

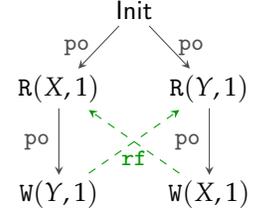
<http://dx.doi.org/10.5381/jot.2026.25.1.a3>

¹ <https://doc.rust-lang.org/std/sync/struct.Arc.html>

$X = Y = 0$ and a postcondition $a = b = 1$. We say the behavior specified by a litmus test is *observable* if the program has an execution that satisfies the pre- and postcondition. When implemented using relaxed accesses, the LB (*load buffering*) behavior is observable, because of instruction reorderings performed by the compiler and/or the processor. Since acquire fences have no effect in the absence of release operations, so is the LBF behavior. The LBD behavior, however, known as an *out-of-thin-air (OOTA)* behavior, is not observable on any real execution platform. Recent editions of the C standard attempt to precisely describe the concurrency behaviors that a C implementation is allowed to exhibit, by listing a number of *axioms* that each execution must satisfy. We refer to the formalization thereof used in (Moiseenko et al. 2025) as C20. These axioms allow the LB and LBF behaviors; unfortunately, since the C20 concept of an execution does not consider dependencies, it cannot distinguish LBD from LB and allows it too. Strengthening C20 to rule out OOTA behaviors like LBD while still allowing all desirable optimizations, such as the reorderings exhibited by LB, has been a long-standing open problem. In the mean-

$$\begin{array}{l}
X = Y = 0 \\
a = X; \quad \parallel \quad b = Y; \\
\mathbf{if}(a == 1) \quad \parallel \quad \mathbf{if}(b == 1) \\
Y = 1; \quad \parallel \quad X = 1; \\
a = b = 1 \\
\text{(LBD)}
\end{array}$$

$$\begin{array}{l}
X = Y = 0 \\
a = X; \quad \parallel \quad b = Y; \\
Y = 1; \quad \parallel \quad X = 1; \\
a = b = 1 \\
\text{(LB)}
\end{array}$$

$$\begin{array}{l}
X = Y = 0 \\
a = X; \quad \parallel \quad b = Y; \\
\mathbf{fence}_{\mathbf{acq}}; \quad \parallel \quad \mathbf{fence}_{\mathbf{acq}}; \\
Y = 1; \quad \parallel \quad X = 1; \\
a = b = 1 \\
\text{(LBf)}
\end{array}$$


(b) A C20 execution graph

(a) Three memory consistency litmus tests

Figure 1 Three memory consistency litmus tests and a C20 execution graph explaining the LB/LBD behavior

time, most work on modular verification of relaxed-consistency programs (Vafeiadis & Narayan 2013; Doko & Vafeiadis 2016, 2017; Dang et al. 2020), has assumed the absence of cycles in the union of program order (po, the total order on a thread’s memory accesses induced by the program’s control flow) and the reads-from (rf) relation relating each write event to the read events that read from it. Without such cycles, it is possible to prove the soundness of program logics by induction on the size of $(\text{po} \cup \text{rf})^+$ prefixes of the execution. In addition to ruling out LBD, however, this acyclicity assumption also rules out LB and LBf.

A breakthrough was achieved, however, with the very recent proposal of XMM (Moiseenko et al. 2025), a framework for concurrency semantics based on *re-execution*. Given an underlying memory model, XMM slightly restricts it to rule out OOTA behaviors, without ruling out porf cycles altogether. Specifically, XMM incrementally builds executions through Execute steps and Re-Execute steps. An Execute step adds an event that reads from an existing event. Thus, it never introduces a porf cycle. A Re-Execute step fixes an rf^{-1} -closed subset of the events of the original execution, called the *committed set*, and then re-builds an execution from scratch, where read events may temporarily “read from nowhere” *but only if they are in the committed set*. Re-Execute steps enable the construction of porf cycles, but the values read are *grounded* by an earlier execution. Applying XMM to C20 yields XC20, which allows LB (and LBf) but rules out LBD.

In this paper, we propose an approach for modularly verifying relaxed-consistency programs that is sound in the presence of porf cycles. We use the core of ARC as a motivating example; its code and intended separation logic specification are shown in Fig. 2. Function $\text{alloc}(v)$ allocates an ARC instance whose payload is v . Every owner of a permission to access the ARC instance at address a holding payload v (denoted $\text{arc}(a, v)$) can read the payload using function $\text{get}(a)$, implemented using a simple nonatomic read. Function $\text{clone}(a)$ duplicates the permission to access a , and $\text{drop}(a)$ destroys it. When the last permission is destroyed, the instance is deallocated. The first field of the ARC data structure is a counter, initialized to 1, incremented at each clone using a relaxed-consistency (rlx) fetch-and-add (FAA) instruction, and decremented at each drop. Since the decrements have release (rel) consistency, they all synchronize with the acquire (acq) fence in the thread that reads

$$\begin{array}{l}
\text{alloc}(v) = \\
\quad \{\mathbf{True}\} \\
\quad \mathbf{cons}(1, v) \\
\quad \{r. \text{arc}(r, v)\} \\
\text{get}(a) = \\
\quad \{\text{arc}(a, v)\} \\
\quad [a + 1]_{\mathbf{na}} \\
\quad \{r. \text{arc}(a, v) \wedge r = v\} \\
\text{clone}(a) = \\
\quad \{\text{arc}(a, v)\} \\
\quad \mathbf{FAA}_{\mathbf{rlx}}(a, 1) \\
\quad \{\text{arc}(a, v) * \text{arc}(a, v)\} \\
\text{drop}(a) = \\
\quad \{\text{arc}(a, v)\} \\
\quad \mathbf{let } n = \mathbf{FAA}_{\mathbf{rel}}(a, -1) \mathbf{ in} \\
\quad \mathbf{if } n = 1 \mathbf{ then } (\\
\quad \quad \mathbf{fence}_{\mathbf{acq}}; \\
\quad \quad \mathbf{free}(a); \mathbf{free}(a + 1) \\
\quad \quad) \\
\quad \{\mathbf{True}\}
\end{array}$$

Figure 2 Core ARC, with desired specs

counter value 1, ensuring that the deallocation does not race with any of the accesses.

As we will show, this algorithm is correct under C20 *provided that the following two properties hold*:

All accesses of the counter
are either increments or release decrements. (1)

No access of the counter reads a value $v \leq 0$. (2)

In particular, clearly, the presence of relaxed decrements would break the algorithm. Verifying (1) or (2) using separation logic is tricky, and we don’t know how to do it. Indeed, consider LBf above. Suppose that we want to verify that all modifications of X and Y are release writes, so that they synchronize with the acquire fences. The problem here is that both writes are *behind* fences. To reason about the fences, we would have to assume the property before verifying it, which is unsound. Instead, we simply assume these properties. Actually, (1) can be verified using a *syntactic* approach. By using a Java-like static type system to protect the `Arc.counter` field it is then possible to syntactically check all accesses to it. For (2), in contrast, we have no suggestions; we leave it as an open problem.

In §2, we briefly recall the C20 memory consistency model. In §3, we propose an approach for verifying programs under C20, under certain assumptions to be checked by other means,

and we use it to verify Core ARC. Specifically, we propose an operational semantics (opsem) for C20 programs instrumented with an *atomic specification* for each location accessed atomically, that specifies a set of *enabled operations* for the location, as well as a *tied precondition* for each enabled operation and a *tied postcondition* for each enabled operation-result pair, both elements of an algebra of *tied resources*. We assume all access events that occur are enabled. In the opsem, an atomic access nondeterministically yields any result enabled under its atomic specification. We argue the opsem’s soundness with respect to C20’s axiomatic semantics, and we apply the Iris (Jung et al. 2015, 2018) logic to the opsem to verify Core ARC.

In §5, then, we adapt this approach to obtain an approach for verifying programs under a minor strengthening of XC20, which we call YC20. Without any assumptions, we verify Core ARC against YC20. We use the grounding guarantees offered by YC20 to prove that all atomic accesses of a location are enabled under its atomic specification. But first we recall the XC20 memory consistency model in §4 and define YC20. We finish by discussing related work (§6) and offering a conclusion (§7).

2. The C20 memory consistency model

We here briefly recall the C20 memory consistency model. For a gentler presentation, see Batty (2015).

In C20, the memory operations $o \in \mathcal{O} = \{R_{na}, R_{rlx}, R_{acq}, fence_{acq}, fence_{rel}\} \cup \{W_{na}(v), W_{rlx}(v), W_{rel}(v) \mid v \in \mathbb{Z}\} \cup \{RMW_{rlx}(f), RMW_{rel}(f), RMW_{acq}(f), RMW_{acqrel}(f) \mid f \in \mathbb{Z} \rightarrow \mathbb{Z}\}$ are the nonatomic, relaxed, and acquire reads, acquire and release fences, nonatomic, relaxed and acquire writes of a value v , and relaxed, release, acquire, and acquire-release read-modify-write (RMW) operations that atomically read a value v from a location and write value $f(v)$, for some function $f \in \mathbb{Z} \rightarrow \mathbb{Z}$. Each memory event is labeled by a tuple (t, ℓ, v, o) specifying the event’s thread $t \in ThreadIds$, location $\ell \in \mathbb{Z}$, result value $v \in \mathbb{Z}$, and operation $o \in \mathcal{O}$.

The result value of a write is always 0. The result value of an RMW is the value that is read, before the modification is performed. It follows that an event labelled by operation $RMW(f)$ and result value v writes value $f(v)$.

There is a special *initialization thread* $t_{init} \in ThreadIds$ that performs a nonatomic write of value 0 to each memory location used by the program.

In C20, the set of behaviors of a program is given by its set of consistent C20 execution graphs. A C20 execution graph is a tuple $(E, lab, po, rf, mo, rmw)$, where E is a set of events, lab is a function that maps each event $e \in E$ to its label, and the *program order* po , *reads-from* relation rf , *modification order* mo , and *read-modify-write* relation rmw are subsets of $E \times E$. A C20 execution graph is *well-formed* if all of the following conditions hold:

- Program order relates the initialization event for each location ℓ to each other access of that location, and otherwise relates two events only if they belong to the same thread. For each non-initialization thread t , program order totally orders the events of t .

- The reads-from relation only relates write or RMW events to read or RMW events. It only relates events writing a value v to location ℓ to events reading value v from location ℓ . It relates at most one write or RMW event to any given read or RMW event. If it relates events e_1 and e_2 by the same thread, e_1 precedes e_2 in program order.
- For each location ℓ , modification order totally orders the write and RMW events on ℓ , and only relates writes or RMW events, and only relates events on the same location.
- rmw only relates read events to write events. It only relates events to their immediate program order successor.

Unless otherwise noted, we only consider well-formed execution graphs.

We say a read or RMW r reads from a write or RMW w if $(w, r) \in rf$. We say an execution graph is *rf-complete* if each read and each RMW reads from some write or RMW.

This definition allows for two ways to represent RMWs: as singular events labelled by an RMW operation, and as pairs of a read and write event related by rmw . We say an execution is *high-level* if its rmw relation is empty, and *low-level* if it has no events labelled by RMW operations. For each high-level execution, there is exactly one corresponding low-level execution (up to graph isomorphism), obtained by replacing each RMW event by the corresponding pair of read and write events and rmw edge. In the other direction, the correspondence is not unique since a pair of read and write events does not uniquely determine an RMW operation’s update function f .

To define consistency of a C20 execution graph, we first need to define the derived relations sw (*synchronizes-with*), hb (*happens-before*), fr (*from-reads*), and eco (*extended coherence order*):

- sw relates a release write or RMW w or a release fence succeeded in program order by a relaxed write or RMW w to an acquire read or RMW r or an acquire fence preceded in program order by a relaxed read or RMW r if r reads from w or there exists a *release sequence* from some RMW that reads from w to some RMW that r reads from. A release sequence is a sequence of RMWs where each next one reads from the preceding one.
- hb is the transitive closure of the union of po and sw .
- fr relates each read or RMW event r to each modification order successor of the write or RMW that r reads from (if any).
- eco is the transitive closure of the union of rf , mo , and fr .

We say a C20 execution graph is *consistent* if it is rf -complete and both of the following properties hold:

- It respects *coherence*, meaning that hb is irreflexive and, furthermore, if an event e happens-before an event e' , e' is not related to e by eco (i.e. eco is consistent with hb).
- It respects *atomicity of RMWs*, meaning that there is no event e such that fr relates the read event of an RMW to e and mo relates e to the write event of the same RMW.

In C20, the set of executions of a program is given by the set of consistent C20 execution graphs generated by the program.

The execution graph in Fig. 1b is an execution of the LB and LBD programs. It has a porf cycle.

We say a C20 execution graph has a data race if it contains two events on the same location, at least one of which is a write or RMW, and at least one of which is nonatomic. In C, if any of a program's executions has a data race, the program is considered to have undefined behavior. The goal of our verification approach is to verify absence of data races.

3. A verification approach for C20

In this section, we propose an operational semantics for C20 (§3.1) and we apply it to verify Core ARC (§3.2).

3.1. An operational semantics for C20

We define an interleaving operational semantics for an instrumented version of our programming language.

3.1.1. Instrumented programs We add two auxiliary commands to the syntax of the programming language: **begin_atomic**(ℓ, Σ) and **end_atomic**(ℓ). When turning a nonatomic memory location ℓ into an atomic location using the **begin_atomic** command, one has to specify an *atomic specification* $\Sigma = (v_0, \mathcal{R}_G, \mathcal{R}_L, \rho_0, \text{pre}, \text{post})$ consisting of an initial value v_0 , cancellative commutative monoids² of global tied resources \mathcal{R}_G (ranged over by ρ) and local tied resources \mathcal{R}_L (ranged over by θ), an initial global tied resource $\rho_0 \in \mathcal{R}_G$, a partial function $\text{pre} : \mathcal{O} \rightarrow \mathcal{R}_G \times \mathcal{R}_L$, mapping each *enabled operation* o to its *tied precondition* (ρ, θ) consisting of a *global tied precondition* ρ and a *local tied precondition* θ , and a partial function $\text{post} : \mathcal{O} \times \mathbb{Z} \rightarrow \mathcal{R}_G \times \mathcal{R}_L$ mapping a pair of an enabled operation o and a result value v at which o is *enabled* to a *tied postcondition* (ρ', θ') consisting of a *global tied postcondition* ρ' and a *local tied postcondition* θ' .³

Example 1. For the Core ARC proof, we will use the following atomic specification: $v_0 = 1$, $\mathcal{R}_G = \mathcal{R}_L = (\mathbb{N}, +, 0)$, $\rho_0 = 1$, $\text{pre} = \{(FAA_{\text{rlx}}(1), (1, 0)), (FAA_{\text{rel}}(-1), (1, 0)), (\text{fence}_{\text{acq}}, (0, 1))\}$, and

$$\begin{aligned} \text{post} = & \{((FAA_{\text{rlx}}(1), z), (2, 0)) \mid z. 1 \leq z\} \\ & \cup \{((FAA_{\text{rel}}(-1), z), (0, 0)) \mid z. 2 \leq z\} \\ & \cup \{((FAA_{\text{rel}}(-1), 1), (0, 1))\} \\ & \cup \{((\text{fence}_{\text{acq}}, 0), (0, 1))\} \end{aligned}$$

In words: we start out with one unit of global tied resource. The enabled operations are the relaxed increments, the release decrements, and the acquire fences. The enabled operation-result pairs are the relaxed increments that read a positive value, the release decrements that read a positive value, and the acquire fences (which always have result value 0). Incrementing and decrementing both consume one unit of global tied resource. Incrementing produces two units, and decrementing produces nothing, except if the result value (i.e. the value that was read,

² A monoid is an algebra (R, \cdot, ε) given by a set R with an associative binary composition operator \cdot and a unit element ε . It is *cancellative* if $v \cdot v_0 = v' \cdot v_0 \Rightarrow v = v'$.

³ We consider only *valid* atomic specifications, where no nonatomic operations or accesses are enabled.

before the decrement) is 1, in which case it produces one unit of local tied resource. Accesses are enabled only at result values (i.e. values read, before the modification) $v \geq 1$. A fence consumes one unit of local tied resource and produces it again.

Here is a more readable Hoare triple-style representation:

$$\begin{aligned} & \frac{1 \leq v}{\{(1, 0)\} FAA_{\text{rlx}}(1) \rightsquigarrow v \{(2, 0)\}} \\ & \frac{2 \leq v}{\{(1, 0)\} FAA_{\text{rel}}(-1) \rightsquigarrow v \{(0, 0)\}} \\ & \{(1, 0)\} FAA_{\text{rel}}(-1) \rightsquigarrow 1 \{(0, 1)\} \\ & \{(0, 1)\} \text{fence}_{\text{acq}} \rightsquigarrow 0 \{(0, 1)\} \end{aligned}$$

For now, we assume that for any **begin_atomic**(ℓ, Σ) event, all accesses a of ℓ that do not happen before it happen after it and enabled under Σ , i.e. $(\text{lab}(a).o, \text{lab}(a).v) \in \text{dom post}$.⁴ In §5, we eliminate all of these assumptions.

The syntax of instrumented programs is as follows:

$$\begin{aligned} e ::= & v \mid x \mid e + e \mid e = e \\ c ::= & \mathbf{cons}(\bar{e}) \mid [e]_{\text{na}} \mid [e] :=_{\text{na}} e \mid \mathbf{free}(e) \\ & \mid \mathbf{begin_atomic}(e, \Sigma) \mid \mathbf{end_atomic}(e) \mid o(e) \\ & \mid \mathbf{if } e \mathbf{ then } c \mid \mathbf{let } x = c \mathbf{ in } c \mid \mathbf{fork}(c) \end{aligned}$$

Command $o(e)$ applies memory access operation o to the location yielded by expression e . Notice that even fence operations are qualified by a location this way. Indeed, our operational semantics takes into account only the synchronization induced by the interplay between the fence and the accesses of this particular location. It is future work to investigate the severity of this incompleteness and to lift it, if necessary.

3.1.2. Consistency of a tied resource with an event We derive a cancellative commutative monoid of *thread-bound tied resources* $\mathcal{R}_B = \text{ThreadIds} \rightarrow \mathcal{R}_L$, ranged over by Θ . Each element of \mathcal{R}_B associates a local tied resource with each thread. Its composition is the pointwise composition and its unit element is the function that maps all thread ids to ε . We further derive a cancellative commutative monoid of *total tied resources* $\mathcal{R}_T = \mathcal{R}_G \times \mathcal{R}_B$ (ranged over by ω), whose composition is the componentwise composition and whose unit element is $(\varepsilon, \varepsilon)$. We will abuse \cdot and ε to denote the composition and unit element of any of these monoids. Initially, there are no local tied resources, so the initial total tied resource is (ρ_0, ε) .

An *atomic location trace* $(G, E_{\text{at}}, \text{init})$ for a location ℓ under an atomic specification Σ consists of a consistent C20 execution graph G , a set $E_{\text{at}} \subseteq G.E$ of events on ℓ , all of whose operations are enabled under Σ (i.e. $\text{lab}(E_{\text{at}}).o \subseteq \text{dom } \Sigma.\text{pre}$), and a nonatomic write $\text{init} \in G.E$ of $\Sigma.v_0$ to ℓ that happens-before all events in E_{at} , such that each read or RMW event in E_{at} reads from some event in $E_{\text{at}} \cup \{\text{init}\}$.

⁴ This implies the program has no **end_atomic**(ℓ) operations, and also does not deallocate the location, suggesting a garbage-collected setting. Note: other resources, like the ARC's payload, might still be non-garbage-collected native resources.

We say an event is *enabled* under Σ if the event is labeled by (t, ℓ, v, o) such that o is enabled at result v : $(o, v) \in \text{dom } \Sigma.\text{post}$.

We say an atomic location trace is *fully enabled* if all events in E_{at} are enabled.

We say a total tied resource ω is *consistent* with an operation o by a thread t resulting in a value v under an atomic specification Σ , denoted $\Sigma, t, v, o \models \omega$, if there exists a fully enabled atomic location trace under Σ that includes an event $e \in E_{\text{at}}$ labeled by an operation o by thread t resulting in v and there exists a subset $E_{\text{ex}} \subseteq E_{\text{at}}$ that includes all events in E_{at} that happen before e but does not include e itself or any events that happen after e , such that for every total ordering of E_{ex} consistent with happens before, starting from $(\Sigma, \rho_0, \varepsilon)$ it is possible to, in this order, consume the tied precondition and produce the tied postcondition of each event $e' \in E_{\text{ex}}$ and arrive at ω . We say that the consistency is *witnessed* by the atomic location trace, e , and E_{ex} .

3.1.3. Operational semantics We instrument the state space of the programming language as follows. The regular heap h stores the values of nonatomic cells only; for atomic cells the semantics does not track the value; it only tracks, in the *atomic heap* A , each atomic location's atomic specification and total tied resource.

A *configuration* $\gamma = (h, A, T)$ consists of a regular heap h , an atomic heap A , and a *thread pool* T which is a partial function mapping the thread id of each started thread to the command it is currently executing. A configuration γ can *step* to another configuration γ' , denoted $\gamma \rightarrow \gamma'$, if there is a started thread t whose command is of the form $K[c]$, where K is a reduction context and c is a command that can step in the current state according to the head step relation \xrightarrow{t}_h , defined by a number of step rules, shown in Fig. 3 and in the Appendix.

Notice that, per rules ATOMICOP and ATOMICOP-STUTTER, if the tied precondition for an operation is not available, the command gets stuck.⁵ Otherwise, the operation consumes the tied precondition and nondeterministically produces a result value v at which the operation is enabled, and the corresponding global tied postcondition ρ'' and local tied postcondition θ'' .

An operation stutters while the total tied resources are not consistent with the operation. This means the thread might be blocked until all events that happen-before it in the C20 execution have happened in the opsem execution. Indeed, the set E_{ex} in the definition of consistency reflects the set of events that have happened in the opsem execution, and the opsem nondeterministically executes the events in any order consistent with hb. Importantly, however, this does *not* imply that all (or any) of the operation's rf^+ -predecessors, which explain the operation's result, have necessarily happened in the opsem execution yet.

We also allow operations to stutter spuriously; this does not matter since our approach is for safety only, not termination or other liveness properties.

⁵ If a configuration where some thread is stuck is reachable, the program is considered unsafe. We prove below that if a program is safe, it has no undefined behavior per C20.

Theorem 1. *If a program is safe under the operational semantics, then it has no undefined behavior under C20 semantics.*

Proof. Fix a C20 execution. It is sufficient to prove, for each hb-prefix of the execution, that the prefix is data-race-free and that the opsem configuration corresponding to the prefix is reachable by taking the opsem steps corresponding to the events of the prefix in any order consistent with hb. By induction on the size of the prefix. *For more details, see Appendix A.2.* \square

3.2. Proof of Core ARC

To satisfy the assumptions stated in §3.1, except for the assumption that no access of the counter reads a value $v \leq 0$, we verify a version of the code in Fig. 2 written in a hypothetical Java-like language with C20 memory semantics, shown in Fig. 4. We assume the hypothetical AtomicLong class guarantees its initialization happens-before all accesses. The goal is to prove that, if accessing the payload requires an arc permission, then all accesses happen-before the closing of the payload.

For the atomic location, we use the atomic specification of Example 1.

Notice that the global tied resource seems to track the value of the location exactly. Since a location's tied resource is tracked in the opsem's atomic heap, does this not imply sequentially consistent semantics? No, because the tied resource reflects only the events that have happened so far in the opsem execution; it does not take into account the "future" events, even though the former may read from the latter. It follows that in general, the global tied resource at the time of an access does *not* match the value read by that access.

As we will see below, each arc permission will include ownership of one unit of global tied resource.

This choice of atomic specification is motivated by the following lemmas, which guarantee that only one decrement reads 1, and that after the acquire fence, there are no outstanding arc permissions:

Lemma 1. *If a total tied resource (ρ, Θ) is consistent with a decrement that reads 1, then $\Theta = 0$ ⁶:*

$$\Sigma, t, 1, \text{FAA}_{\text{rel}}(-1) \models (1, 0) \cdot (\rho, \Theta) \Rightarrow \Theta = 0$$

Proof. Fix an atomic location trace, an event e , and a set E_{ex} that witnesses the consistency. By contradiction: assume some event e' in E_{ex} has a nonzero local tied postcondition. It follows that e' is a decrement that reads 1. However, since both e and e' are decrements that read 1, some increment that reads 0 must intervene between e' and e in modification order. This contradicts the fact that all events in the atomic location trace are enabled. \square

Lemma 2. *If a total tied resource (ρ, Θ) is consistent with an acquire fence, then $\rho = 0$:*

$$\Sigma, t, 0, \text{fence}_{\text{acq}} \models (0, 0[t := 1]) \cdot (\rho, \Theta) \Rightarrow \rho = 0$$

⁶ We abuse 0 to denote the thread-bound resource that maps each thread to 0.

BEGINATOMIC

$$\frac{(\ell, \Sigma.v_0) \in h}{(h, A, \mathbf{begin_atomic}(\ell, \Sigma)) \xrightarrow{t}_h (h[\ell := \emptyset], A[\ell := (\Sigma, (\Sigma.\rho_0, \varepsilon))], 0)}$$

ENDATOMIC

$$\frac{(\ell, (\Sigma, \omega)) \in A}{(h, A, \mathbf{end_atomic}(\ell)) \xrightarrow{t}_h (h[\ell := v], A[\ell := \perp], 0)}$$

ATOMICOP

$$\frac{\begin{array}{l} (\ell, (\Sigma, (\rho \cdot \rho', \Theta[t := \theta \cdot \theta']))) \in A \\ (o, (\rho, \theta)) \in \Sigma.\text{pre} \quad ((o, v), (\rho'', \theta'')) \in \Sigma.\text{post} \quad \Sigma, t, v, o \models (\rho \cdot \rho', \Theta[t := \theta \cdot \theta']) \end{array}}{(h, A, o(\ell)) \xrightarrow{t}_h (h, A[\ell := (\Sigma, (\rho'' \cdot \rho', \Theta[t := \theta'' \cdot \theta'])], v)}$$

ATOMICOP-STUTTER

$$\frac{(\ell, (\Sigma, (\rho \cdot \rho', \Theta[t := \theta \cdot \theta']))) \in A \quad (o, (\rho, \theta)) \in \Sigma.\text{pre}}{(h, A, o(\ell)) \xrightarrow{t}_h (h, A, o(\ell))}$$

Figure 3 Selected step rules of the operational semantics

```

public class Arc<T extends Closeable> {
  private final AtomicLong counter = new AtomicLong(1);
  public final T payload;

  public Arc(T payload)
  { this.payload = payload; }

  public void clone() { counter.fetch_and_add_relaxed(1); }

  public void drop() {
    long v = counter.fetch_and_add_release(-1);
    if (v == 1) {
      AtomicLong.fence_acquire();
      payload.close();
    }
  }
}

```

Figure 4 Core ARC in a Java-like language with C20 memory semantics

Proof. Fix an atomic location trace, an event e , and a set E_{ex} that witnesses the consistency. By the existence of the local tied resource in thread t , some event e' must precede e in thread t that produces it. e' must be a decrement that reads value 1. It follows that the events that mo-precede e' , as ordered by mo, must consist of the *init* event, followed by n increments and n decrements, in some order. Since the decrements (including e') are release events, they happen-before e and are therefore in E_{ex} . Since their tied preconditions can be consumed, there must be at least n increments in E_{ex} , each of which happens-before one of these decrements and therefore mo-precedes e' . It suffices to prove that the total number of increments in E_{ex} is n . For this, in turn, it suffices to prove that all increments in E_{ex} mo-precede e' . Suppose an increment $e'' \in E_{\text{ex}}$ mo-succeeds e' . It therefore cannot happen-before e' or any of the decrements mo-preceding e' . But then there exists an order of the events in E_{ex} consistent with hb where e'' occurs at a point where the global tied resource has already reached 0 so the tied precondition for e'' cannot be consumed, which is a contradiction. \square

We apply stock Iris (Jung et al. 2015, 2018) to our operational semantics. For an atomic location ℓ we use an Iris cancellable invariant with tag τ containing $\text{Inv}_{\ell, \tau, \gamma, \gamma'}$ defined as follows, where F ranges over bags of positive real numbers, $\sum F$ means the sum of the elements of F , $\#F$ means the number of elements in F , and $\ell \xrightarrow{q} _$ denotes a fractional permission (Boyland 2003; Bornat et al. 2005) with fraction $q \in [0, 1] \subseteq \mathbb{R}$ for location ℓ :

$$\begin{aligned} \text{Inv}_{\ell, \tau, \gamma, \gamma'} &= \exists \rho, \Theta, F. \\ \ell \xrightarrow{\text{at}} (\Sigma, (\rho, \Theta)) &* (\Theta \neq 0 \vee [\tau]_{1/2}) * \ell + 1 \xrightarrow{1-\sum F} _ \\ &* [\tau]_{(1-\sum F)/2} * \bullet(\rho, \Theta) \uparrow * \bullet F \uparrow_{\gamma'} \wedge \rho = \#F \end{aligned}$$

The invariant holds full permission for the atomic location. Furthermore, one half of the cancellation token $[\tau]$ is initially inside the invariant and removed by the thread that decreases the count

to zero, exploiting Lemma 1 (at which point a local tied resource is produced, which is never destroyed). There is an element $q \in F$ for each outstanding arc permission, denoting the fraction of the ownership of $\ell + 1$ owned by that arc permission; the remainder is inside the invariant. That arc permission also owns a fraction $q/2$ of the cancellation token. “Fictional ownership” by arc permissions of tied resources and elements of F is realized using Iris *ghost cells* γ and γ' using the AUTH resource algebra, whose *authoritative parts*, denoted by \bullet , are held in the invariant, and for which a *fragment*, denoted by \circ , is held in each arc permission. A ghost cell’s fragments always add up exactly to the authoritative part.

We define predicate `arc` as follows:

$$\text{arc}(\ell, v) = \exists q, \tau, \gamma, \gamma'. [\tau]_{q/2} * \tau \boxed{\text{Inv}_{\ell, \tau, \gamma, \gamma'}} * \ell + 1 \xrightarrow{q} v$$

$$* \circ \{q\}_{\gamma'} * \circ (1, 0)_{\gamma}$$

Immediately before the `fenceacq` command, thread t owns $[\tau]_{1/2} * \circ (0, 0 [t := 1])_{\gamma}$. When the fence succeeds, thanks to Lemma 2, the thread can cancel the invariant, end the atomic location, and free the memory cells.

4. The XC20 and YC20 memory consistency models

We brief recall the XC20 memory consistency model (Moiseenko et al. 2025), and we define our minor strengthening of it, which we call YC20.

Given a C20 execution graph G , we define the *relaxed program order* `rpo` as the transitive closure of the subset of `po` that only relates events e and e' if:

- e is a relaxed (or stronger⁷) read or RMW and e' is an acquire fence, or
- e is an acquire (or stronger) event, or
- e' is a release (or stronger) event, or
- e is a release fence and e' is a relaxed (or stronger) write.

A program behavior is allowed by the XC20 memory consistency model if it matches a consistent C20 execution graph that can be *constructed* from the empty graph through a number of *XMM construction steps*. There are two kinds of such steps: Execute steps and Re-Execute steps.

An Execute step simply adds a `porf`-maximal event: it relates graphs G and G' if both are consistent C20 execution graphs and $G'.E = G.E \cup \{e\}$ and $G'|_{G.E} = G$ and e is `porf`-maximal in G' .

A Re-Execute step selects an `rf`-complete subset of the events of the original execution, called the *committed set*, and a `po`-maximal `po`-prefix of the committed set called the *determined set*, and then constructs a new execution, starting from the determined set, where, at every step, a `po`-maximal event e is added such that either e is not a read or RMW event, or e reads from an event that was added earlier, or e is in the committed set. The newly constructed execution must in the end be a consistent

C20 execution graph and must include the entire committed set, which implies that all events that were added from the committed set again read from the same event that they read from in the original execution.

More precisely, a Re-Execute step relates graphs G and G' if all of the following hold:

- G and G' are consistent C20 execution graphs,
- there exists a set of events C , called the *committed set*, that is a subset of $G.E$ and $G'.E$ such that $G|_C = G'|_C$ and every read or RMW event in C reads from an event in C ,
- there exists a subset D of C , called the *determined set*, that is `G.po`-prefix-closed, i.e. any event that `G.po`-precedes an event in D is also in D ,
- D is `G.po`-maximal in C , i.e. if an immediate `G.po`-successor of an event in D is in C then it is in D ,
- if $G'.rpo$ relates an event e to a non-determined event e' , then e is a determined event,
- G' can be constructed from $G|_D$ through a sequence of Guided Steps under C .

A Guided Step under C relates graphs G and G' if G' is obtained by adding a `po`-maximal event to G that either is not a read or RMW event, or reads from an event in G , or is in C .

The definitions above define XMM and XC20 if *low-level* executions are used, where each RMW is represented as a pair of a read event and a write event, related by the `rmw` relation, and they define YMM and YC20 if *high-level* executions are used, where each RMW is represented as a single event labelled by an RMW operation. The only difference is that in XMM, it is possible for only one of the two events constituting an RMW to be in the committed set of a Re-Execute step, whereas in YMM, an RMW is entirely in the committed set or not at all. In XMM, if a Re-Execute step relates graphs G and G' , as far as we can tell, it is possible for the write of a relaxed increment in G to be used to “justify” a relaxed decrement in G' , which would seem to break our approach for verifying Core ARC.

Moiseenko et al. (2025) proved a number of important results about XC20, including optimal compilation schemes to the main instruction set architectures and the soundness of a number of important program transformations performed by optimizing compilers. We would expect these to still hold for YC20, but verifying this is future work.

5. A verification approach for YC20

In this section, we propose an operational semantics for YC20 (§5.1) and we apply it to verify Core ARC (§5.2).

5.1. An operational semantics for YC20

We say a total tied resource ω is *grounding-consistent* with an operation o by a thread t resulting in a value v under an atomic specification Σ , denoted $\Sigma, t, v, o \models_{\mathbf{g}} \omega$, if there exists an atomic location trace under Σ where E_{at} includes an event e labeled by an operation o by thread t resulting in v and all events in $E_{\text{at}} \setminus \{e\}$ are enabled under Σ and there exists a subset $E_{\text{ex}} \subseteq E_{\text{at}}$ that includes all events that happen before e as well as all release events in $E_{\text{at}} \setminus \{e\}$ but does not include e itself or

⁷ with `na < rlx < acq < acqrel` and `rlx < rel < acqrel`

any events that happen after e , such that for every total ordering of E_{ex} consistent with happens before, starting from $(\Sigma, \rho_0, \varepsilon)$ it is possible to, in this order, consume the tied precondition and produce the tied postcondition of each event $e' \in E_{\text{ex}}$ and arrive at ω . We say the grounding-consistency is *witnessed* by the atomic location trace, e , and E_{ex} .

Our operational semantics for YC20 is exactly the same as the one for C20 presented in §3.1, except that we do not make the assumptions made there, and that we add one constraint on atomic specifications: that each operation’s tied precondition be *sufficient*. Informally, this means that during the grounding process the tied precondition ensures the operation is performed only at a value at which it is enabled under the atomic specification. Formally, if $(o, (\rho, \theta)) \in \text{pre}$ and $\Sigma, t, v, o \models_{\text{g}} (\rho, \varepsilon[t := \theta]) \cdot \omega$, then $(o, v) \in \text{dom post}$.

Theorem 2. *If a program is safe under the operational semantics, then it has no undefined behavior under YC20 semantics.*

Proof. It is sufficient to prove that each C20 execution reachable through YMM’s Execute and Re-Execute steps is *grounded* and has no undefined behavior. An execution is grounded if, essentially, each atomic event is enabled under its atomic specification. By induction on the number of YMM steps. First, we prove based on the definition of YMM that for any Execute or Re-Execute step, if the original execution is grounded, the new execution is *weakly grounded*, meaning essentially that there exists some *grounding order*, a total order on the events of the execution, such that each event is either grounded or reads from an event that precedes it in the grounding order. We then prove, for any weakly grounded C20 execution, that each hb-prefix of each grounding-order-prefix-closed subset of the execution is data-race-free and grounded and corresponds to an opsem configuration that is reachable by taking the opsem steps corresponding to the events of the prefix in any order consistent with hb. By induction on the size of the subset and nested induction on the size of the prefix. *For more details, see Appendix A.3.* \square

5.2. Proof of Core ARC

We can now verify the unmodified ARC code from Fig. 2, without any assumptions. The atomic specification used, the Lemmas 1 and 2 and the Iris proof are adopted unchanged from §3.2. The only difference is that we now need to prove the following lemma:

Lemma 3. *The tied preconditions are sufficient.*

Proof. Fix an operation o such that $(o, (\rho, \theta)) \in \text{pre}$ and a value v such that $\Sigma, t, v, o \models_{\text{g}} (\rho, 0[t := \theta]) \cdot \omega$. It suffices to prove that $(o, v) \in \text{dom post}$. Let $(G, E_{\text{at}}, \text{init})$ be an atomic location trace and $e \in E_{\text{at}}$ and $E_{\text{ex}} \subseteq E_{\text{at}}$ an event and set of events that witness the grounding-consistency. The case $o = \text{fence}_{\text{acq}}$ is trivial because its only possible result value is 0; assume o is an FAA operation. By rf-completeness of the atomic location trace and the fact that all events other than e are enabled under Σ , e must read a nonnegative value v . It suffices to prove that $v \neq 0$. By contradiction; assume $v = 0$. Then there exists a sequence of FAA events in E_{at} such that the first one reads from

init, each next one reads from the previous one, and e reads from the last one. It follows that this sequence contains n increment operations and $n + 1$ decrement operations. Since E_{ex} contains all release events in $E_{\text{at}} \setminus \{e\}$, it contains these $n + 1$ decrement operations. Since it is possible to consume all of their tied preconditions, E_{ex} must also contain n increment operations that each happen-before one of these decrement operations. It follows that E_{ex} must contain the n increment operations that mo-precede e . It now suffices to prove that E_{ex} contains only these n increment operations. Suppose there was some increment event $e' \in E_{\text{ex}}$ that mo-succeeds e . It follows that e' cannot happen-before any of the decrements that mo-precede e . It follows that there is an order on the events of E_{ex} consistent with hb where e' occurs after exactly n increments and $n + 1$ decrements so the global tied resource has already reached 0 at the point where the tied precondition for e' is consumed, which is a contradiction. \square

6. Related work

Core ARC was verified before in FSL++ (Doko & Vafeiadis 2017), an extension of Fenced Separation Logic (Doko & Vafeiadis 2016) with support for ghost state, as well as in the Iris-based approach of Relaxed RustBelt (Dang et al. 2020). The latter even verified full ARC, except that they had to strengthen two relaxed accesses to acquire accesses. One of these was in fact a bug in ARC; the status of the other remains open. Both of these earlier proofs, however, assume the absence of perf cycles.

The concept of tied resources was inspired by AxSL (Hammond et al. 2024), an Iris-based separation logic for the relaxed memory model of the Arm-A processor architecture, which allows load buffering. In AxSL, however, tied resources are tied to *events*, not locations. While an in-depth comparison is future work, their logic and their soundness proof appear to be quite different from ours.

7. Conclusion

We presented a preliminary result on the modular verification of relaxed-consistency programs under YC20 semantics, a minor strengthening of XC20, a new memory consistency model that rules out out-of-thin-air behaviors while allowing load buffering, and we used it to verify Core ARC. The most urgent next step is to further elaborate the soundness proof, and ideally mechanize it, to mechanise the Core ARC proof, and to verify that the results proved for XC20 by Moiseenko et al. (2025) also hold for YC20. Others are to investigate whether the approach supports other important relaxed algorithms besides Core ARC (such as full ARC), and how to apply the approach in a semi-automated verification tool such as VeriFast (Vogels et al. 2015). It would also be interesting to try and eliminate the open assumption we used to prove Core ARC under C20 semantics, or else to obtain some kind of an impossibility result.

Our operational semantics is a non-intrusive extension of classical semantics for sequentially-consistent (SC) languages, such as the HeapLang language targeted by Iris by default. This suggests its compatibility with existing applications, extensions,

and tools for logics for SC languages such as Iris and VeriFast. The opsem used by Relaxed RustBelt (Dang et al. 2020), in contrast, replaces the regular heap by a *message pool* combined with three *views* per thread (the acquire view, the current view, and the release view). Comparing the pros and cons of these approaches is important future work.

Acknowledgments

We thank Evgenii Moiseenko for helpful interactions. This research is partially funded by the Research Fund KU Leuven, and by the Cybersecurity Research Program Flanders.

References

- Batty, M. J. (2015). *The C11 and C++11 concurrency model* (Doctoral dissertation, University of Cambridge, UK). Retrieved from <https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.708458>
- Bornat, R., Calcagno, C., O’Hearn, P. W., & Parkinson, M. J. (2005). Permission accounting in separation logic. In J. Palsberg & M. Abadi (Eds.), *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005* (pp. 259–270). ACM. Retrieved from <https://doi.org/10.1145/1040305.1040327> doi: 10.1145/1040305.1040327
- Boyland, J. (2003). Checking interference with fractional permissions. In R. Cousot (Ed.), *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings* (Vol. 2694, pp. 55–72). Springer. Retrieved from https://doi.org/10.1007/3-540-44898-5_4 doi: 10.1007/3-540-44898-5_4
- Dang, H., Jourdan, J., Kaiser, J., & Dreyer, D. (2020). RustBelt meets relaxed memory. In *Proceedings of the 47th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2020* (Vol. 4, pp. 34:1–34:29). ACM. Retrieved from <https://doi.org/10.1145/3371102> doi: 10.1145/3371102
- Doko, M., & Vafeiadis, V. (2016). A program logic for C11 memory fences. In B. Jobstmann & K. R. M. Leino (Eds.), *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016, Proceedings* (Vol. 9583, pp. 413–430). Springer. Retrieved from https://doi.org/10.1007/978-3-662-49122-5_20 doi: 10.1007/978-3-662-49122-5_20
- Doko, M., & Vafeiadis, V. (2017). Tackling real-life relaxed concurrency with FSL++. In H. Yang (Ed.), *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings* (Vol. 10201, pp. 448–475). Springer. Retrieved from https://doi.org/10.1007/978-3-662-54434-1_17 doi: 10.1007/978-3-662-54434-1_17
- Hammond, A., Liu, Z., Pérami, T., Sewell, P., Birkedal, L., & Pichon-Pharabod, J. (2024, January). An axiomatic basis for computer programming on the relaxed Arm-A architecture: The AxSL logic. In (Vol. 8). New York, NY, USA: ACM. Retrieved from <https://doi.org/10.1145/3632863> doi: 10.1145/3632863
- Jung, R. (2020). *Understanding and evolving the Rust programming language* (Doctoral dissertation, Saarland University, Saarbrücken, Germany). Retrieved from <https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647>
- Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., & Dreyer, D. (2018). Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28, e20. Retrieved from <https://doi.org/10.1017/S0956796818000151> doi: 10.1017/S0956796818000151
- Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., & Dreyer, D. (2015). Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In S. K. Rajamani & D. Walker (Eds.), *Proceedings of the 42nd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015* (pp. 637–650). ACM. Retrieved from <https://doi.org/10.1145/2676726.2676980> doi: 10.1145/2676726.2676980
- Moiseenko, E., Meluzzi, M., Meleshchenko, I., Kabashnyi, I., Podkopaev, A., & Chakraborty, S. (2025, January). Relaxed memory concurrency re-executed. In *Proceedings of the 52nd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2025* (Vol. 9). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3704908> doi: 10.1145/3704908
- Vafeiadis, V., & Narayan, C. (2013). Relaxed separation logic: a program logic for C11 concurrency. In A. L. Hosking, P. T. Eugster, & C. V. Lopes (Eds.), *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013* (pp. 867–884). ACM. Retrieved from <https://doi.org/10.1145/2509136.2509532> doi: 10.1145/2509136.2509532
- Vogels, F., Jacobs, B., & Piessens, F. (2015). Featherweight VeriFast. *Log. Methods Comput. Sci.*, 11(3). Retrieved from [https://doi.org/10.2168/LMCS-11\(3:19\)2015](https://doi.org/10.2168/LMCS-11(3:19)2015) doi: 10.2168/LMCS-11(3:19)2015

About the authors

Bart Jacobs is an associate professor at the DistriNet research group at the department of Computer Science at KU Leuven (Belgium). His main research interest is in modular formal verification of concurrent programs. You can contact the author at bart.jacobs@kuleuven.be or visit <https://distrinet.cs.kuleuven.be/people/BartJacobs/>.

Justus Fasse is a PhD student at the DistriNet research group at the department of Computer Science at KU Leuven (Belgium). His research is on modular formal verification of concurrent programs, with a focus on total correctness properties. You can

contact the author at justus.fasse@kuleuven.be or visit <https://distrinet.cs.kuleuven.be/people/JustusFasse/>.

A. Appendix

A.1. Operational semantics

We define the *reduction contexts* K as follows:

$$K ::= \square \mid \mathbf{let } x = K \mathbf{ in } c$$

We write $K[c]$ to denote the command obtained by replacing the hole (\square) in K by c .

We use $c[v/x]$ to denote substitution of a value v for a variable x in command c . Notice that our expressions e never get stuck and have no side-effects. We treat closed expressions that evaluate to the same value as equal.

The step relation of our operational semantics is defined in Figs. 3 and 5. The primed nonatomic write command $[\ell] :='_{\text{na}} v$ is a syntactic construct that is not allowed to appear in source programs and occurs only during execution. It denotes a nonatomic write in progress. This way of modeling nonatomic writes, borrowed from RustBelt (Jung 2020), ensures that a program with a data race has a configuration reachable in the opsem where one of the racing threads is stuck.

A.2. A Verification Approach for C20: Soundness

In this section, we prove that if a program is safe under the operational semantics, it has no undefined behavior under C20 semantics. We here concentrate on proving data-race-freedom; other types of undefined behavior can be handled similarly.

For the remainder of this section, we fix an instrumented program and we assume that it is safe under the operational semantics.

We fix a C20 execution graph G .

In this section we assume that for any location ℓ that is accessed atomically, a `begin_atomic`(ℓ, Σ) event coincides with the initializing write to ℓ , which happens-before all other accesses of ℓ , and we assume all accesses of ℓ are enabled under Σ . (We lift these assumptions in §5.) We say ℓ has atomic specification Σ .

Definition 1 (Execution Prefix). *We say a subgraph of the execution is an execution prefix if it is prefix-closed with respect to happens-before, i.e. if an event is in the prefix, then all events that happen before it are also in the prefix.*

Definition 2 (Data race). *A data race is a pair of accesses of the same location, not ordered by happens-before, at least one of which is a write and at least one of which is nonatomic. For the purposes of this definition, we treat allocations, deallocations, and conversion to or from atomic mode as nonatomic writes.*

Notice that under the assumptions of this section, all data races are among nonatomic accesses and atomic accesses are never involved in a data race.

We say a configuration γ of the operational semantics corresponds to an execution prefix P , denoted $P \sim \gamma$, if all of the following hold:

- the thread pool matches the final configurations of the threads of the prefix
- the prefix has no data races.
- the nonatomic heap maps each allocated cell that is only accessed nonatomically to the value written by its final write.
- the atomic heap maps each allocated cell ℓ that is accessed atomically to its atomic specification and to the tied resource reached by, starting from $(\Sigma, \rho_0, \epsilon)$, first producing the tied postconditions of the atomic access events on ℓ in the prefix, and then consuming their tied preconditions.

Importantly, we have $P \sim \gamma \wedge P \sim \gamma' \Rightarrow \gamma = \gamma'$.

Lemma 4. *In any reachable configuration of the operational semantics, for each location ℓ , one of the following holds:*

- *Not yet allocated:* $h(\ell) = \perp$ and $A(\ell) = \perp$
- *In nonatomic mode:* $\exists v. h(\ell) = v \vee h(\ell) = \emptyset$ and $A(\ell) = \perp$
- *In atomic mode:* $h(\ell) = \emptyset$ and $\exists \Sigma, \omega. A(\ell) = (\Sigma, \omega)$
- *Deallocated:* $h(\ell) = \emptyset$ and $A(\ell) = \perp$

Proof. By induction on the number of steps. \square

We say an execution prefix P is *valid* if it corresponds to a configuration γ and γ is reached by executing the events of P in any order consistent with happens-before.

A.2.1. Proof of main lemma In this sub-subsection, we prove the main lemma, which says that every execution prefix is valid. By induction on the size of the prefix. We fix an execution prefix P , and we assume all smaller prefixes are valid. The case where P is empty is trivial; assume P is nonempty.

Lemma 5. *P is data-race-free.*

Proof. By contradiction. Assume there are two events $e, e' \in P$, in threads t and t' not ordered by hb. Obtain P' by removing from P e as well as the events that happen-after e or e' . By the induction hypothesis, P' corresponds to some configuration, and this configuration is reachable by executing e' last. Therefore, in this configuration, t is about to execute e and t' has just executed e' . By case analysis on e and e' , we obtain a contradiction. \square

Lemma 6. *For every order on the events of P consistent with hb, a configuration γ such that $P \sim \gamma$ is reachable by executing the events in this order.*

Proof. Fix such an order. Let e , in thread t , be the final event in this order. Apply the induction hypothesis to $P' = P \setminus \{e\}$ to obtain some γ' such that $P' \sim \gamma'$. It suffices to prove that executing e in γ' reaches a configuration γ such that $P \sim \gamma$. By case analysis on e . We elaborate one case.

- **Case** e is an atomic operation. By the fact that γ' is reachable and, by the safety of the program, therefore not stuck, we can, starting from γ' , consume e 's tied precondition and produce its tied postcondition to obtain γ . It remains to prove that $\Sigma, t, v, o \models \omega$, where $\text{lab}(e) = (t, \ell, v, o)$ and $\gamma'.A(\ell) = (\Sigma, \omega)$. Take as the atomic location trace the

$$\begin{array}{c}
\text{CONS} \\
\hline
\frac{\{\ell, \dots, \ell + n - 1\} \cap \text{dom } h = \emptyset}{(h, A, \mathbf{cons}(v_1, \dots, v_n)) \xrightarrow{t}_h (h[\ell := v_1, \dots, \ell + n - 1 := v_n], A, \ell)} \\
\text{NA-WRITE-START} \\
\hline
(h[\ell := v_0], A, [\ell] :=_{\text{na}} v) \xrightarrow{t}_h (h[\ell := \emptyset], A, [\ell] :='_{\text{na}} v) \\
\text{NA-READ} \\
\hline
(h[\ell := v], A, [\ell]_{\text{na}}) \xrightarrow{t}_h (h[\ell := v], A, v) \\
\text{NA-WRITE-END} \\
\hline
(h[\ell := \emptyset], A, [\ell] :='_{\text{na}} v) \xrightarrow{t}_h (h[\ell := v], A, 0) \\
\text{FREE} \\
\hline
(h[\ell := v], A, \mathbf{free}(\ell)) \xrightarrow{t}_h (h[\ell := \perp], A, 0) \\
\text{IF-TRUE} \\
\hline
\frac{v \neq 0}{(h, A, \mathbf{if } v \mathbf{ then } c) \xrightarrow{t}_h (h, A, c)} \\
\text{IF-FALSE} \\
\hline
(h, A, \mathbf{if } 0 \mathbf{ then } c) \xrightarrow{t}_h (h, A, 0) \\
\text{LET} \\
\hline
(h, A, \mathbf{let } x = v \mathbf{ in } c) \xrightarrow{t}_h (h, A, c[v/x]) \\
\text{HEAD-STEP} \\
\hline
\frac{(h, A, c) \xrightarrow{t}_h (h', A', c')}{(h, A, T[t := K[c]]) \rightarrow (h', A', T[t := K[c']])} \\
\text{FORK} \\
\hline
\frac{t' \neq t \quad t' \notin \text{dom } T}{(h, A, T[t := K[\mathbf{fork}(c)]]) \rightarrow (h, A, T[t := K[0]][t' := c])}
\end{array}$$

Figure 5 Remaining operational semantics step rules

entire execution graph G , with E_{at} the set of all atomic operations on ℓ in G , and *init* the $\mathbf{begin_atomic}(\ell, \Sigma)$ event. For E_{ex} , take the accesses of ℓ in P' .

□

A.3. A Verification Approach for YC20: Soundness

In this section, we prove that if a program is safe under the operational semantics, it has no undefined behavior under YC20 (Moiseenko et al. 2025) semantics. We again focus on proving data-race-freedom.

For the remainder of this section, we fix an instrumented program and we assume that it is safe under the operational semantics.

Lemma 7. *If the program obtained by inserting redundant nonatomic read-write pairs $[e] :=_{\text{na}} [e]_{\text{na}}$ into a YC20 program is data-race-free, then the original program is data-race-free as well.*

We apply YC20 semantics to the instrumented program by treating the $\mathbf{begin_atomic}$ and $\mathbf{end_atomic}$ commands like redundant nonatomic read-write pairs.

A.3.1. Grounded C20 executions

Definition 3. *We say an atomic access event e on a location ℓ is grounded with respect to a $\mathbf{begin_atomic}(\ell, \Sigma)$ event if e as well as e 's rf^+ predecessors are enabled under Σ .*

Definition 4. *We say an atomic access event e on a location ℓ is grounded if it has exactly one hb-maximal hb-preceding $\mathbf{begin_atomic}(\ell, \Sigma)$ event e' and e is grounded with respect to e' .*

Definition 5. *We say a C20 execution is grounded if every atomic access event in this execution is grounded.*

Definition 6. *We say an atomic access event e on a location ℓ is weakly grounded under some grounding order (a total order on the events of the execution) if at least one of the following is true:*

- At least one $\mathbf{begin_atomic}(\ell, _)$ event precedes e in grounding order and e is grounded with respect to the most recent preceding one in grounding order
- both of the following are true:
 - for any rf^+ predecessor e' of any transitive-reflexive grounding-order-predecessor of e , if e' is a release event then e' precedes e in the grounding order
 - e is a write or fence event or e 's rf predecessor exists and is before e in the grounding order and is also weakly grounded or is a nonatomic access

Definition 7. *We say a C20 execution is weakly grounded if there is a single total order on the events of the execution (called its grounding order) consistent with happens-before such that every atomic access event of the execution is weakly grounded.*

Lemma 8. *If an Execute step goes from G to G' , and G is grounded, then G' is weakly grounded. For the grounding order, take some arbitrary total order on the events of G consistent with happens-before, followed by the newly added event.*

Lemma 9. *During a Re-Execute step, a non-determined committed event e is not a release event.*

Proof. By contradiction; assume e is a release event. e must have an uncommitted po-predecessor e' ; otherwise, e would be determined. But since e is a release event, there is an rpo edge

from e' to e . But only determined events are allowed to have outgoing rpo edges. \square

Lemma 10. *During a Re-Execute step, the order in which the events are added is consistent with happens-before.*

Proof. By contradiction. Assume that a Guided Step that adds an event e also adds a happens-before edge from e to some existing event. Since e is po-maximal, it must be that this edge is a synchronizes-with edge, which implies that e is a release event with a reads-from edge to some existing event. This implies e is a committed event. Since it is not a determined event, we obtain a contradiction by Lemma 9. \square

Lemma 11. *If a Re-Execute step goes from G to G' using committed events C , and G is grounded, then G' is weakly grounded. For the grounding order, take some arbitrary order on the determined events consistent with the happens-before order of G' , followed by the other events in the order in which they are added by the Guided Steps.*

Proof. We prove, by induction on the number of preceding Guided Steps, that for the event e added by a Guided Step, all rf^+ -predecessors that are release events were added earlier. If e is an uncommitted event with an rf-predecessor e' , we have that e' was added earlier, so by the induction hypothesis we have the goal. If e is a committed event, its rf^+ -predecessors are also committed events, since the committed set is rf-complete. By Lemma 9 we have the goal. \square

A.3.2. Proving one XMM step We fix a weakly grounded C20 execution graph G and a corresponding grounding order.

Definition 8 (Execution Prefix). *We say a subgraph of the execution is an execution prefix if it is prefix-closed with respect to happens-before, i.e. if an event is in the prefix, then all events that happen before it are also in the prefix.*

Definition 9 (Data race). *A data race is a pair of accesses of the same location, not ordered by happens-before, at least one of which is a write and at least one of which is nonatomic. For the purposes of this definition, we treat allocations, deallocations, and conversion to or from atomic mode as nonatomic writes.*

If an execution prefix is data-race-free, then it is well-defined at each access of a location within that prefix whether at that event the location is in nonatomic mode or in atomic mode, and, if it is in atomic mode, what its atomic specification is, based on the hb-maximal conversion event that happens before the access. We say the execution prefix is mode-well-formed if nonatomic accesses occur only on locations in nonatomic mode, and atomic accesses occur only on locations in atomic mode.

We say a configuration γ of the operational semantics corresponds to an execution prefix P , denoted $P \sim \gamma$, if all of the following hold:

- the thread pool matches the final configurations of the threads of the prefix
- the prefix has no data races. It follows that each location's final mode (atomic or nonatomic) and (in the case of nonatomic locations) final value within the prefix is well-defined.

- the prefix is mode-well-formed
- the prefix is grounded.
- the nonatomic heap maps each allocated cell whose final mode is nonatomic to the value written by its final write.
- the atomic heap maps each allocated cell ℓ whose final mode is atomic with atomic specification Σ to the atomic specification assigned to it by its latest (i.e. hb-maximal) **begin_atomic** event e , and to the tied resource obtained by, starting from $(\Sigma, \rho_0, \varepsilon)$, first producing the tied postconditions and then consuming the tied preconditions of the atomic access events on ℓ in the prefix that happen-after e .

Notice that the configuration corresponding to an execution prefix P , if it exists, is unique. We denote it by γ_P .

Lemma 12. *In any reachable configuration of the operational semantics, for each location ℓ , one of the following holds:*

- Not yet allocated: $h(\ell) = \perp$ and $A(\ell) = \perp$
- In nonatomic mode: $\exists v. h(\ell) = v \vee h(\ell) = \emptyset$ and $A(\ell) = \perp$
- In atomic mode: $h(\ell) = \emptyset$ and $\exists \Sigma, \omega. A(\ell) = (\Sigma, \omega)$
- Deallocated: $h(\ell) = \emptyset$ and $A(\ell) = \perp$

Proof. By induction on the number of steps. \square

We say an execution prefix P is *valid* if it corresponds to a configuration γ and γ is reached by executing the events of P in any order consistent with happens-before.

We define P_n as the prefix constituted by the first n events in grounding order.

A.3.3. Proof of main lemma In this sub-subsection, we prove the main lemma, which says that, for all n , all sub-prefixes of P_n are valid. By induction on n . We fix an n and we assume all sub-prefixes of all P_m with $m < n$ are valid. The case where $n = 0$ is trivial; assume $n > 0$.

In the following two lemmas, let e be the grounding-order-maximal event in P_n , and let t be the thread of e .

Lemma 13. *P_n is data-race-free.*

Proof. By contradiction. By the induction hypothesis, we have P_{n-1} is data-race-free, so there must be some event $e' \in P_{n-1}$ in some thread t' that races with e . Consider the prefix P obtained by removing from P_{n-1} all events that happen-after e' . Since P is valid, we have a γ such that $P \sim \gamma$ and γ is reachable by executing the events of P in any order. Assume e' was executed last. So in γ , t' has just executed e' and t is about to execute e . By case analysis on e and e' . \square

Lemma 14. *P_n is grounded.*

Proof. By the induction hypothesis, we have that P_{n-1} is grounded. It remains to prove that e is grounded. Assume it is weakly grounded. By $\gamma_{P_{n-1}}$ not being stuck, we have that e 's operation o is enabled and that the current mode of e 's location ℓ is atomic, with some atomic specification Σ .

- We prove that e 's rf^+ predecessors are enabled under Σ . e 's immediate rf-predecessor e' , if any, is in P_{n-1} and is therefore grounded; by groundedness of e' the goal follows for e 's indirect rf^+ -predecessors.

- We exploit sufficiency of e 's tied precondition. Let E_{ex} be the set of all atomic access events on ℓ in P_{n-1} that happen-after the hb-maximal **begin_atomic** event on ℓ in P_n . We construct an atomic location trace with E_{at} consisting of e , all rf^+ predecessors of e , and all rf^* predecessors of the events in E_{ex} . By sufficiency, we have that e is enabled in Σ .

□

Lemma 15. *For every sub-prefix P of P_n , and for every order on the events of P consistent with hb, a configuration γ such that $P \sim \gamma$ is reachable by executing the events in this order.*

Proof. By induction on the size of P . Fix a P and fix such an order. Let e , in thread t , be the final event in this order. Apply the induction hypothesis to $P' = P \setminus \{e\}$ to obtain some γ' such that $P' \sim \gamma'$. It suffices to prove that executing e in γ' reaches a configuration γ such that $P \sim \gamma$. By case analysis on e . We elaborate one case.

- **Case** e is an atomic operation. Assume $\text{lab}(e) = (t, \ell, v, o)$. By the fact that γ' is reachable and, by the safety of the program, therefore not stuck, we know $\gamma'.A(\ell) = (\Sigma, \omega)$ for some Σ and ω , and we can, starting from γ' , consume e 's tied precondition and produce its tied postcondition to obtain γ . It remains to prove that $\Sigma, t, v, o \models \omega$. Let E_{ex} be the set of atomic accesses on ℓ in P' that happen-after the hb-maximal **begin_atomic**($\ell, _$) event e' in P' . Take as the set E_{at} of the atomic location trace the rf^{-1} closure of $E_{\text{ex}} \cup \{e\}$, i.e. the events of $E_{\text{ex}} \cup \{e\}$ as well as all rf^+ -predecessors of those events.

□

A.3.4. Proving an YMM trace

Theorem 3. *The program is grounded and data-race-free under YC20.*

Proof. By induction on the number of YMM steps, i.e. the number of (Re-)Execute steps. Base case: the empty execution is grounded and data-race-free. Induction step: assume G is grounded and data-race-free, and assume the step goes from G to G' . From Lemmas 8 and 11 we know G' is weakly grounded. By Lemma 15, we obtain that G' is grounded and data-race-free. □