CQS: A FORMALLY-VERIFIED FRAMEWORK FOR FAIR AND ABORTABLE SYNCHRONIZATION

Nikita Koval
JetBrains Research
nikita.koval@jetbrains.com

Dmitry Khalanskiy
JetBrains
dmitry.khalanskiy@jetbrains.com

Dan Alistarh
IST Austria
dan.alistarh@ist.ac.at

May 23, 2023

ABSTRACT

Writing concurrent code that is both correct and efficient is notoriously difficult. Thus, programmers often prefer to use synchronization abstractions, which render code simpler and easier to reason about. Despite a wealth of work on this topic, there is still a gap between the rich semantics provided by synchronization abstractions in modern programming languages—specifically, *fair* FIFO ordering of synchronization requests and support for *abortable* operations—and frameworks for implementing it correctly and efficiently. Supporting such semantics is critical given the rising popularity of constructs for asynchronous programming, such as coroutines, which abort frequently and are cheaper to suspend and resume compared to native threads.

This paper introduces a new framework called `CancellableQueueSynchronizer` (CQS), which enables simple yet efficient implementations of a wide range of fair and abortable synchronization primitives: mutexes, semaphores, barriers, count-down latches, and blocking pools. Our main contribution is algorithmic, as implementing both fairness and abortability efficiently at this level of generality is non-trivial. Importantly, all our algorithms, including the CQS framework and the primitives built on top of it, come with *formal proofs* in the Iris framework for Coq for many of their properties. These proofs are modular, so it is easy to show correctness for new primitives implemented on top of CQS. From a practical perspective, implementation of CQS for native threads on the JVM improves throughput by up to two orders of magnitude over Java's `AbstractQueuedSynchronizer`, the only practical abstraction offering similar semantics. Further, we successfully integrated CQS as a core component of the popular Kotlin Coroutines library, validating the framework's practical impact and expressiveness in a real-world environment. In sum, `CancellableQueueSynchronizer` is the first framework to combine expressiveness with formal guarantees and solid practical performance. Our approach should be extensible to other languages and families of synchronization primitives.

1 Introduction

Providing the "right" set of programming abstractions to enable efficient and correct concurrent code is a question as old as the field of concurrency [1, 2]. One of the most basic primitives is the *mutex*, which allows access to the critical section to at most one thread, via `lock()` and `unlock()` invocations. Standard libraries, e.g., the Java concurrency library [3], provide more general primitives, such as the *semaphore*, which allows at most a fixed number of threads to be in the critical section simultaneously, the *barrier*, which allows a set of threads to wait for each other at a common program point, and the *count-down latch*, which allows threads to wait until a given set of operations is completed.

Although basic versions of the above primitives exist in most specialized libraries, programmers often require *stronger semantics* from synchronization abstractions, which are supported by modern programming languages such as Java, C#, Go, Kotlin and Scala. One particularly desirable property is *fairness* [4], by which the order of critical section traversals should respect the FIFO order of arrivals, to avoid starvation. A second key property is *abortability*, which enables a thread to *cancel* its request, due to a time-out or user-specified behavior. Abortability and scalability are especially important in the context of *coroutines* [5, 6], the number of which can be in the millions simultaneously, and which can be frequently cancelled. Coroutines are also significantly cheaper to suspend and resume compared to

native threads: internally, they are scheduled on a thread pool, so when a coroutine is suspended, the corresponding thread immediately picks up another one and executes it instead, so the native thread never blocks. In such a setting, efficient cancellation is vital, while fairness becomes less expensive and more natural. Coroutines are now a key component of modern programming languages such as Java, C++, Go, Scala, and Kotlin.

Implementing fairness and cancellation efficiently in a concurrent context is known to be very challenging, and there is a long line of work proposing highly non-trivial designs [7, 8, 9, 10, 11, 12]. Modern languages and libraries typically either restrict the generality of the semantics, providing more efficient *unfair* synchronization, or implement complex constructs, which may lead to correctness and performance issues. This paper addresses the question of implementing *fair* and *abortable* synchronization primitives in a way that is general, efficient, and easy to reason.

For intuition, we begin with the observation that most of the synchronization operations we focus on are inherently *blocking*: threads attempt to acquire a shared resource or synchronize, and may have to wait for the resource to become available. For example, the mutex `lock()` operation either acquires the lock instantly or adds the currently running thread to a queue of waiting operations and suspends. The `unlock()` invocation resumes the first waiting `lock()` request, handing the lock over. To our knowledge, the only practical abstraction to implement such synchronization primitives in full generality is the `AbstractQueuedSynchronizer` in Java [3], which maintains a FIFO queue of suspended requests in a way reminiscent of the state-of-the-art CLH mutex [13]. While the `AbstractQueuedSynchronizer` has been extremely influential, its design does not scale to high contention. Our goal is to provide a design that is general enough to support a wide range of abstractions, but also efficient enough to support modern usage scenarios.

Our Contribution. We introduce a new framework called `CancellableQueueSynchronizer` (CQS), which enables simple and efficient implementations of a wide range of *fair* and *abortable* synchronization and communication primitives, such as mutexes, semaphores, barriers, count-down latches, and blocking pools. We show that CQS can implement this wide range of synchronization primitives, for which we provide both formal proofs and extensive experimental validation, showing significant practical improvements over the state of the art.

Conceptually, the goal of the `CancellableQueueSynchronizer` framework is to efficiently maintain a FIFO queue of waiting threads, corresponding to operations to be completed. To this end, CQS provides two main operations: (1) `suspend()`, which adds the current thread as a waiter into the queue and suspends, and (2) `resume(result)`, which tries to retrieve and resume the first waiter with the specified result. One key advantage of the CQS semantics is that it allows operations to invoke `resume(..)` before `suspend()`: we actively use this property for implementation simplicity and better performance. Despite the relative simplicity of the `CancellableQueueSynchronizer` API, it allows us to support a rich set of synchronization and communication primitives.

The data structure behind `CancellableQueueSynchronizer` uses techniques from modern concurrent queue implementations [14, 15], leveraging the `Fetch-and-Add` instruction for better scalability. In brief, our solution is based on a logically-infinite array, equipped with two position counters, indexing `suspend()` and `resume(..)` operations, respectively. Each operation starts by incrementing its counter via `Fetch-and-Add`, thus reserving the cell in the first-come-first-served order. The rest of the synchronization is performed in the cell: `suspend()` stores the current thread, while `resume(..)` wakes up the suspended thread.

The main novelty behind `CancellableQueueSynchronizer` is the efficient built-in support for *aborting/cancelling* operations. We emulate the infinite array with a linked list of fixed-sized *cell segments*, so each cell stores a waiting operation. On thread cancellation, the cell state should be reclaimed to avoid memory leaks, but also segments full of cancelled requests should be physically removed from the linked list. One naive way to implement such functionality is by linearly searching the waiting queue for the corresponding segment, which is then unlinked [3]. However, this would have a worst-case linear time in the queue size, making frequent cancellations lead to significant overheads. While for threads this approach is sufficient (since their number is small and they rarely abort), for coroutines, of which millions can exist at the same time and which cancel frequently, significant complexity improvements are required. We propose a more efficient design, where segments form a concurrent doubly-linked list, enabling constant time removals via careful pointer manipulations. This also allows us to support different cancellation modes: in case of *simple* cancellation, `resume(..)` is allowed to fail if the waiter located in the corresponding cell was cancelled, whereas *smart* cancellation provides a mechanism to efficiently skip a sequence of aborted requests but requires complex mechanisms to ensure that some thread is always resumed.

Formal Proofs in Iris/Coq. The complexity of the resulting CQS implementation renders manual correctness proofs quite challenging and error-prone. We provide *modular formal proofs* for all the presented primitives in Coq [16] using the Iris separation logic [17]. We formally specify the CQS operations and then demonstrate that they obey the semantics corresponding to the primitives we consider. One property we do not show formally is the FIFO order,

which is notoriously difficult to approach in Iris but can be demonstrated through classical proofs. We emphasize the complexity of our formalization task, as only a few similar real-world implementations are formally verified [18, 19, 20, 21, 22]. Our proofs for CQS span approximately 8000 lines of Coq code, often requiring non-trivial reasoning. Yet, the proofs are *modular*, so they can be employed as the basis for proving new synchronization primitives implemented on top of CQS, reducing formalization effort. Specifically, proving each higher-order CQS-based primitive presented in the paper on this basis takes only around 500 lines.

Evaluation. We integrated the CQS framework as part of the standard Kotlin Coroutines library and used it to implement several fundamental synchronization primitives. To validate performance, we implemented `CancellableQueueSynchronizer` on the JVM for native threads and compared it against the state-of-the-art `AbstractQueuedSynchronizer` framework in Java [3], which aims to solve the same problem, and, to our knowledge, is the only practical abstraction that provides similarly general semantics. We present different versions of mutex and semaphore, barrier and count-down-latch primitives, and two versions of blocking pools. Our algorithms outperform existing implementations in almost all scenarios and are sometimes faster by orders of magnitude.

In particular, our semaphore implementation outperforms the standard Java solution, which is implemented via `AbstractQueuedSynchronizer` [3], up to `4x` in the uncontended case where the number of threads does not exceed the number of permits, and up to `90x` in a highly-contended scenario. For the count-down-latch implementation, our solution shows up to `7x` speedup compared to the Java library, while the barrier synchronization is faster by up to `4x`. For blocking pools, which share a limited set of resources, our approach is faster than the Java library implementation by up to `150x`. In some cases, the *fair* synchronization primitives we present even outperform the *unfair* variants in the Java standard library. Finally, results show that the cancellation support of CQS is more efficient than the one of the `AbstractQueuedSynchronizer` framework. Our analysis shows that these improvements come mainly because from the superior scalability of our design.

2 Basic CQS Algorithm

In this section, we describe the key ideas behind the `CancellableQueueSynchronizer` algorithm in an iterative fashion, using a simple non-abortable mutex construct as an example. We then focus on the complexities of supporting cancellation/abortability in the next section.

Thread Management. We manipulate threads to suspend and resume operations. While our main application is coroutines, we will use threads for illustration, as they may be more familiar to the readers. Listing 1 presents the API we use in the paper. We emphasize that our approach can be directly adapted to any concurrency model, such as coroutines, futures, or continuations. [1] Our implementations for Java native threads and Kotlin coroutines (Section 6) support this claim.

Our API assumes that the currently-running thread can be obtained by calling `currentThread()`, and suspended by invoking `park(..)`. While suspended, the thread can be aborted via `cancel()` call, becoming unable to resume. In that case, the `onCancel` lambda provided in `park(..)` is executed. If a thread is cancelled in an active state, the cancellation takes effect with the following `park(..)` invocation.

```
interface Thread {
 fun park(onCancel: lambda () -> Unit[2]): Any
 fun unpark(result: Any): Bool
 fun cancel() // invoked by user
}
fun currentThread(): Thread
```

Listing 1: Thread management API.

To resume a thread, the `unpark(result)` function should be called. It returns `true` if the resumption succeeds, so the corresponding `park(..)` invocation completes with the specified result. Otherwise, if the thread is already cancelled, `unpark(result)` returns `false`. Notably, `unpark(result)` can be called before `park(..)` – in this case, the following `park(..)` invocation immediately completes without suspension, returning the provided result.

Environment. For simplicity, we assume the sequentially-consistent memory model, which matches our implementation, as all real-world weak memory models provide sequential consistency for data-race-free programs. In addition to plain reads and writes, we use atomic `Compare-and-Swap (CAS)`, `Get-and-Set`, and `Fetch-and-Add (FAA)` instructions, which are available in all modern programming languages. We also assume that the runtime environment

[1] Many languages support asynchronous programming either explicitly via `Future`-s, or implicitly via the `async/await` construct that internally manipulates continuation objects.

[2] `Unit` is a type with only one value: the `Unit` object. This type corresponds to the `void` type in Java.

supports garbage collection (GC). Reclamation techniques such as hazard pointers [23] or hazard eras [24] can be used in environments without GC.

High-Level Algorithm Overview. At the logical level, the `CancellableQueueSynchronizer` maintains a first-in-first-out (FIFO) queue of waiting requests and provides two main functions:

- `suspend(): T`, which adds the current thread as a waiter into the queue and suspends, and
- `resume(result: T): Bool`, which tries to retrieve and resume the next waiter, passing the specified value of type `T`.

A key advantage is that the framework allows to invoke `resume(..)` before `suspend()` as long as it is known that `suspend()` will happen eventually, so synchronization primitive implementations can allow such races. In Section 4, we present several algorithms that leverage this property for better performance and simplicity.

```
1  val cells = InfiniteArray()
2  var suspendIdx: Int64 = 0
3  var resumeIdx:  Int64 = 0
4
5  fun suspend(): T {
6    i := FAA(&suspendIdx, +1)
7    // Try to suspend in cells[i].
8    t := currentThread()
9    if CAS(&cells[i], null, t):
10     return park() // enqueued, suspend
11   // Read the result and finish.
12   result := cells[i]; cells[i] = TAKEN
13   return result
14 }
15 fun resume(result: T) {
16   i := FAA(&resumeIdx, +1)
17   t := cells[i]
18   if t == null: // is the cell empty?
19     // ‘suspend()‘ is coming, try to
20     // install the result and finish.
21     if CAS(&cells[i], null, result):
22       return
23     // The cell stores a thread.
24     t = cells[i]
25   // Resume the waiting request.
26   cells[i] = RESUMED
27   t.unpark(result) // t is Thread
28 }
```

Listing 2: High-level CQS implementation on top of an infinite array without abortability support.

A useful mental image of CQS is that of an infinite array supplied with two counters: one that references the cell in which the new waiter should be enqueued as part of the next `suspend()` call, and one that references the next cell for `resume(..)`. The intuition is that `suspend()` atomically increments its counter via `Fetch-and-Add`, stores the currently running thread in the corresponding cell, and suspends. Likewise, `resume(..)` increments its counter, visits the corresponding cell, and resumes the stored thread with the specified value. However, if `resume(..)` comes before `suspend()`, it simply places the value in the cell and finishes — `suspend()` grabs the value later and completes without an actual suspension. [3]

Listing 2 provides a high-level pseudocode for this simplified `CancellableQueueSynchronizer`, without abortability support. An infinite array `cells` (line 1) stores waiting threads and values inserted by racing resumptions. Counters `suspendIdx` and `resumeIdx` (lines 2–3) reference cells for the next `suspend()` and `resume(..)` operations.

When `suspend()` starts, it first gets its index and increments the counter atomically via `Fetch-And-Add (FAA)`, which returns the value right before the increment (line 6). Next, it obtains the currently running thread to be inserted into the cell (line 8) and tries to do so via `Compare-And-Swap (CAS)` (line 9). If this `CAS` succeeds, the operation parks the thread, finishing when resumed (line 10). Otherwise, a concurrent `resume(..)` has already visited the cell — thus, `suspend()` extracts the placed value, cleans the cell by placing a special `TAKEN` token (line 12), and returns the extracted value (line 13). Note that in the mutex implementation, we always pass `Unit` through CQS; other data structures, such as blocking pools discussed in Section 4.4, may pass different values.

```
1  val cqs = CQS<Unit>()
2  var state: Int = 1 // "unlocked" intially
3  fun lock() {
4    s := FAA(&state, -1)
5    if s > 0: return // was the lock acquired?
6    cqs.suspend() // suspend otherwise
7  }
8  fun unlock() {
9    s := FAA(&state, +1)
10   // Resume the first waiting
11   // request if there is one.
12   if s < 0: cqs.resume(Unit)
13 }
```

Listing 3: Basic mutex algorithm without abortability support using the CQS framework.

[3] The `suspend()` and `resume(..)` race behavior is similar to the thread parking mechanism in both our API and Java, where `unpark(..)` followed by `park()` results in the latter operation returning immediately.

Symmetrically, `resume(..)` increments `resumeIdx` first (line 16). It then checks whether the cell is empty (line 18), in which case it tries to place the resumption value directly into the cell (line 21). If the attempt fails, a waiter is already stored in the cell, so the algorithm re-reads it (line 24). After the waiter is extracted, the operation stores a special `RESUMED` token in the cell to avoid memory leaks and resumes the extracted thread (lines 26–27).

Mutex on Top of CQS. To illustrate how primitives should use CQS, consider the simple mutex implementation from Listing 3. The rough idea is to maintain a `state` field (line 2) that stores 1 if the mutex is unlocked, and $w \leq 0$ if the mutex is locked. In the latter case, the negated value of w is the number of waiters on this mutex.

Initially, the mutex is unlocked and its `state` equals 1. When a `lock()` operation arrives, it atomically decrements the state, setting it to 0 (line 4), so the logical state becomes "locked". Since the previous logical state was "unlocked", the operation completes immediately (line 5). However, if another `lock()` arrives after that, it changes `state` to -1, keeping the logical state as "locked" and incrementing the number of waiters. Since the mutex was already locked, this invocation suspends via CQS (line 6). Likewise, `unlock()` increments `state`, either making the mutex "unlocked" if the counter was 0, or decrementing the number of waiters (line 9). In the latter case, `unlock()` resumes the first waiter via CQS (line 12). It is worth emphasizing that `lock()` and `unlock()` contain only five lines of easy-to-follow code in total.

Non-Blocking Operations. Synchronization primitives typically provide non-blocking variants of operations, such as the `tryLock()` sibling of `Mutex.lock()`, which succeed only when the operation does not require suspension. However, supporting them becomes non-trivial when `resume(..)` comes before `suspend()`, so the data (e.g., the lock permit) is stored in CQS and cannot be extracted without suspension; thus, the non-blocking sibling cannot access it.

To solve the problem, we introduce a special *synchronous* resumption mode, so that `resume(..)` always makes a rendezvous with `suspend()` and does not leave the value in CQS, failing when this rendezvous cannot happen in bounded time. We view this as an extension to `CancellableQueueSynchronizer` and present it in Appendix A.

Infinite Array Implementation. The last building block of the basic CQS implementation is the emulation of an infinite array. Since all cells are processed in sequential order, the algorithm only requires having access to the cells between `resumeIdx` and `suspendIdx` and does not need to store an infinite number of cells. We follow the approach behind the channels implementation in Kotlin [25], maintaining a linked list of cell segments, each containing a fixed number of cells, as illustrated in Figure 1.

Each segment has a unique id and can be seen as a node in a Michael-Scott queue [26]. Following this structure, we maintain only those cells that are in the current active range (between `resumeIdx` and `suspendIdx`) and access them similarly to an array. Specifically, we change the current working segment after completing operations equal to the number of cells in each segment.

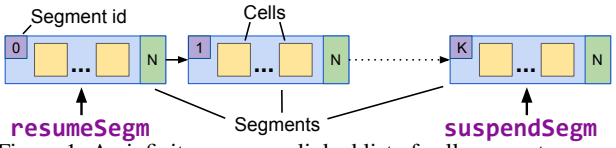

Figure 1: An infinite array as a linked list of cell segments.

Despite conceptual simplicity, the implementation of this structure is non-trivial, as shown in [25]. We discuss the implementation and the required changes to the CQS algorithm in Appendix B.

3 Cancellation Support

In this section, we extend the basic construct above with cancellation support. We assume that threads can be aborted via `Thread.cancel()` call, which bounds the following `unpark(..)` to fail. Additionally, the `onCancel` *cancellation handler* provided in the `park(..)` call[4] is invoked when the thread aborts. We will use this functionality later in this section.

We support two cancellation modes: *simple* and *smart*. Intuitively, the difference is that in the *simple* cancellation mode, `resume(..)` fails if the thread in the corresponding cell has been cancelled, whereas the *smart* cancellation enables efficient skipping a sequence of aborted requests.

[4]Since in practice we manipulate threads or coroutines, cancellation should be handled via an existing mechanism. In Java, for example, aborted threads throw `InterruptedException`, which can be caught and processed by the user. Moreover, some coroutines libraries, such as Kotlin Coroutines [6], already support an API similar to the one we use.

3.1 Simple Cancellation

The *simple* cancellation mode is relatively straightforward — when a waiter becomes cancelled, the `resume(..)` operation that processes this cell is bound to fail. Thus, the code for `resume(..)` in Listing 2 should return `true` if `t.unpark(result)` at line 27 succeeds, and `false` on failure, indicating that the thread has already been aborted. Figure 2 shows the corresponding cell life-cycle.

An important technical detail is that aborted threads should be physically removed from the waiting queue to allow the garbage collector to reclaim the related memory. Thus, we specify a *cancellation handler* that replaces the aborted `Thread` with a special `CANCELLED` marker, according to the diagram in Figure 2. In addition, we must remove segments full of cancelled cells from the linked list to avoid memory leaks; we discuss the corresponding part of the algorithm in Appendix B.

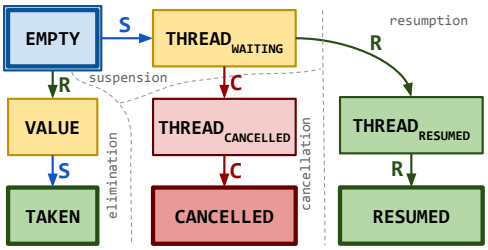

Figure 2: Cell life-cycle with *simple* cancellation. When a thread becomes cancelled by a successful `Thread.cancel()` invocation (tagged with "C"), its content changes to `CANCELLED` to avoid memory leaks, and the corresponding `resume(..)` fails. The edges marked with `S` and `R` correspond to the transitions by `suspend()` and `resume(..)`.

Mutex with Simple Cancellation. Please recall the mutex algorithm presented in Listing 3. With simple cancellation, `resume(..)` fails when the resuming thread is already aborted (`THREAD`$_{\texttt{CANCELLED}}$ or `CANCELLED` state). In this case, the corresponding `lock()` request is no longer valid, and the `unlock()` invocation, which performs `resume(..)` on this cell, incremented the counter (which must have been decremented by a `lock()` operation earlier). Thus, the balance is met, and `unlock()` should restart.

Limitations. One issue with the cancellation logic above is that it requires the `resume(..)` operation to process all the cancelled cells. Consider N `lock()` operations which execute and then immediately abort — the following call to `unlock()` increments `state` and unsuccessfully invokes `resume(..)` exactly N times. This leads to $\Theta(N)$ complexity, which is, nevertheless, amortized by `Thread.cancel()` invocations. Ideally, however, `unlock()` should require $O(1)$ time under no contention and should not "pay" for the cancelled requests.

Another problem is that it is sometimes infeasible to wait until a *resume(..)* operation observes that the waiter is cancelled. We often wish to immediately learn about a waiter being cancelled and change the state correspondingly. As an example, consider a readers-writer lock and the following execution: (1) a reader comes and takes a lock, (2) a writer arrives and suspends, (3) then, another reader arrives and also suspends, because it should take a lock after the suspended writer. After that, (4) the suspended writer becomes cancelled, so the second reader should be resumed and take the lock. However, with simple cancellation, the effect of cancellation is postponed until another operation tries to resume the cancelled waiter, so the reader does not wake up. Making cancellations take effect immediately is critical in this context.

3.2 Smart Cancellation

A better option would be to skip cancelled waiters in `resume(..)` and install a cancellation handler that de-registers the operation when it aborts. For mutex, this could be incrementing the `state` field. However, a naive approach where `resume(..)` simply skips aborted threads would be incorrect.

The Problem. Figure 3 illustrates a potential problematic execution with such a mutex. Assume it is initially locked and two threads start. The first thread invokes `lock()`, placing itself in the CQS, and immediately aborts; however, the `state` is not incremented back yet. After that, the second thread calls `unlock()`, which increments the `state` counter (so it becomes 0) and intends to wake up a waiting `lock()` operation. The corresponding `resume(..)` sees the first cell in `CANCELLED` state and places its value in the next empty cell. The execution switches back to the first thread, and the cancellation handler of the aborted `lock()` increases the counter to 1. The resulting state is shown in the figure. Finally, two `lock()` calls by both threads are performed (they are under the dashed red line). One of them decrements

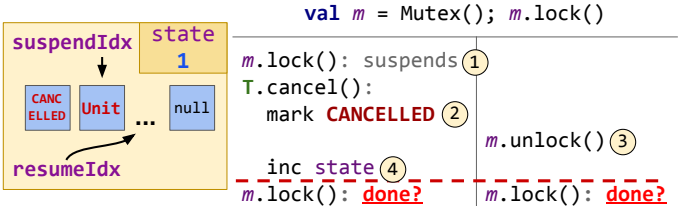

Figure 3: An incorrect execution of mutex with naive cancellation strategy, where the cancellation handler increments `state` back, while `resume(..)` simply skips cancelled cells.

`state` to 0 and enters the critical section; the other suspends via CQS and, observing the value in the cell, also proceeds to enter the critical section, thus breaking the mutex semantics.

The REFUSE State. Notice that the naive version above would work fine in cases where the cancellation handler does not change the mutex state from "locked" to "unlocked" (thus, `state` stays non-positive). The problem occurs when the "last" waiter becomes cancelled, and a concurrent `resume(..)` tries to complete it. In this case, `resume(..)` must be informed that there is no longer any waiter in the `CancellableQueueSynchronizer` that could receive the value.

To signal this, a new REFUSE state is added to the cell life-cycle; see Figure 4 on the right for the updated cancellation part. This state signals that an operation attempted to abort, but determined that there is an upcoming `resume(..)` and the aborted waiter was the last one in the CQS. Thus, the `resume(..)` that inevitably visits the cell should be refused by CQS and will no longer attempt to pass the value to any waiter.

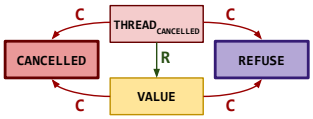

Figure 4: Cell life-cycle for *smart* cancellation, with a special REFUSE state. The suspension, elimination, and resumption parts stay unchanged (see Figure 2).

Smart Cancellation API. Users who develop primitives on top of CQS with smart cancellation should implement `onCancellation()` and `completeRefusedResume(value)` functions, whose semantics are described in Listing 4. When a waiter is cancelled (the cell state changes to $\texttt{THREAD}_{\texttt{CANCELLED}}$), the cancellation handler invokes `onCancellation()`, which tries to logically remove the waiter from the data structure. If the `resume(..)` operation that sees this cell can safely skip it and still match with another non-cancelled `suspend()`, the operation returns `true`, and the cell state becomes CANCELLED. Otherwise, when the cell state becomes REFUSE, and the corresponding `resume(..)` should be refused, `onCancellation()` should return `false`. This way, the refused `resume(..)` invokes `completeRefusedResume(...)` to complete the operation.

```
1 // Invoked on cancellation and returns
2 // 'true' if the cell should enter to
3 // CANCELLED state and 'false' when it
4 // should  transition to REFUSE.
5 fun onCancellation(): Bool
6 // Defines how to process a refused
7 // resume(..) with the specified value
8 fun completeRefusedResume(value: T)
```

Listing 4: Smart cancellation API.

Note that the behavior of `resume(..)` depends on whether the aborted thread moves the cell to CANCELLED or REFUSE state. However, if `resume(..)` observes $\texttt{THREAD}_{\texttt{CANCELLED}}$ state (the cell stores a `Thread` instance while `unpark(..)` fails), the expected behavior can not yet be predicted. We resolve this race by *delegating* the rest of the current `resume(..)` to the cancellation handler, replacing the thread instance with the resumption value — see the corresponding transition from $\texttt{THREAD}_{\texttt{CANCELLED}}$ to VALUE in Figure 4. After that, when the cancellation handler changes the cell's state to CANCELLED or REFUSE, it receives the value and completes the resumption correspondingly. In this case, the value passed to `resume(..)` can be out of the data structure for a while but is guaranteed to be processed eventually. Note that `resume(..)` never fails when using the smart cancellation mode.

Mutex with Smart Cancellation. Consider the mutex example again. Listing 5 presents the `onCancellation()` and `completeRefusedResume(..)` implementations for the basic algorithm from Listing 3; the rest stays the same.

When a `lock()` request aborts, the `onCancellation()` operation increments `state` (thus, decrementing the number of waiters). However, when the increment changes `state` to 1 ("unlocked"), the operation must return `false` to refuse the upcoming `resume(..)`. After that, the `resume(..)` that comes to the cell sees it in REFUSE state and invokes `completeRefusedResume(..)`. For the mutex, the lock is already successfully returned at the moment of incrementing `state` in `onCancellation()`, so this function does nothing. However, when CQS is used to transfer elements (see blocking pools in Subsection 4.4 as an example), the refused element should be returned back to the data structure via `completeRefusedResume(..)`.

```
1  fun onCancellation(): Bool {
2   // Increment the counter back.
3   s := FAA(&state, +1)
4   // s < 0: still in the "locked" state;
5   // s = 0:
        the mutex has become "unlocked",
6   //       refuse the upcoming resume(..).
7   return s < 0
8  }
9  fun completeRefusedResume(permit: Unit)
       {
10  // Do nothing, the mutex has already
11  // been moved to the "unlocked" state.
12 }
```

Listing 5: Cancellation handling in the *smart* mode for the basic mutex algorithm from Listing 3.

```
1  fun resume(value: T): Bool {
2   i := FAA(&resumeIdx, +1)
3   while (true): // modify the cell state
4    cur := cells[i]
5    when {
6     cur == null:
7       if CAS(&cells[i], null, value):
8         return true
9     cur is Thread:
10      if cur.unpark(value):
11        cells[i] = RESUMED
12        return true
13      // The thread is cancelled.
14      if cancellationMode == SIMPLE:
15        return false
16      // In smart cancellation, delegate
17      // this resume(..) completion
18      // to the cancellation handler.
19      if CAS(&cells[i], cur, value):
20        return true
21    cur == CANCELLED:
22      // Fail with simple cancellation.
23      if cancellationMode == SIMPLE:
24        return false
25      // Skip the cell in the smart mode
26      return resume(value)
27    cur == REFUSE:
28      completeRefusedResume(value)
29      return true
30   }
31 }
```

```
32 fun cancellationHandler(s: Segment,
33                          i: Int) {
34  // Which cancellation mode do we use?
35  if cancellationMode == SIMPLE:
36    // Mark the cell state to
37    // CANCELLED and finish.
38    s[i] = CANCELLED
39    s.onCancelledCell()
40    return
41  // Smart cancellation mode is used.
42  markCancelled := onCancellation()
43  if markCancelled:
44    // Mark the cell as CANCELLED.
45    old := GetAndSet(&s[i], CANCELLED)
46    // Did it store an aborted thread?
47    if old is Thread:
48      s.onCancelledCell()
49    else: // old is a value of type T
50      // A concurrent resume(..) has
51      // delegated its completion.
52      resume(old)
53  else:
54    // Move the cell state to REFUSE.
55    old := GetAndSet(&s[i], REFUSE)
56    // Did it store an aborted thread?
57    if old is Thread: return
58    // A concurrent resume(..) has
59    // delegated its completion;
60    // old is a value of type T.
61    completeRefusedResume(old)
62 }
```

Listing 6: Pseudocode for `resume(..)` that supports all cancellation modes and the corresponding cancellation handler. The `suspend()` implementation stays the same. The user-specified operations are highlighted in yellow. The `onCancelledCell()` operation, highlighted with green, informs the segment about a new cancelled cell — we have to remove segments full of cancelled cells to avoid memory leaks; the details are discussed in Appendix B.

The `resume(..)` Operation. Listing 6 presents a pseudocode for `resume(..)` that supports all cancellation modes and for the cancellation handler — the function that is invoked when `Thread` becomes cancelled; it is set in the `park(..)` invocation (see Listing 1). For simplicity, we assume that CQS uses an infinite array in `resume(..)`; the changes required for support of cancellation in its emulation are discussed in Appendix B.

Like before, `resume(..)` increments `resumeIdx` first (line 2). After that, the corresponding cell should be modified — this logic is wrapped with a `while(true)` loop (lines 3–29); the current cell state is obtained in the beginning of it (line 4). When the cell is empty (line 6), `resume(..)` tries to set the resumption value to the cell (lines 7–8). If the corresponding `CAS` succeeds, this `resume(..)` finishes immediately. If the `CAS` fails, the cell modification procedure restarts.

When the cell stores a suspended thread (line 9), `resume(..)` tries to complete it (line 10). If successful, the cell value is cleared for garbage collection, and the operation finishes (lines 11–12). Otherwise, the thread has been cancelled. In the simple cancellation mode, `resume(..)` simply fails (lines 14–15). With the smart cancellation, `resume(..)` tries to replace the cancelled waiter with the resumption value, thus, delegating its completion to the cancellation handler, and finishes on success (line 19–20). On failure, one of the branches below will be entered.

When the cell is in `CANCELLED` state (line 21), `resume(..)` either fails in the simple cancellation mode (lines 23–24), or skips this cell in the smart one, invoking `resume(..)` one more time (line 26). In Appendix B, we describe how to skip a sequence of `CANCELLED` cells in $O(1)$ under no contention, with the infinite array implemented as a linked list of segments.

In the remaining case, when the cell is in the `REFUSE` state (line 27), this `resume(..)` should be refused, and `completeRefusedResume(..)` is called (line 28). After that, the operation successfully finishes (line 29).

The Cancellation Handler. The cancellation handler can be specified as a parameter of the `park(..)` call (see Listing 1) and is invoked when the thread becomes aborted. Here, the `cancellationHandler(..)` function accepts

the segment and the location index of the cell inside it — we know them at the point of invoking `park(..)`, so the handler has access to the cell and can update its state to `CANCELLED` or `REFUSE`.

In the first case, when the simple cancellation mode is used (lines 35–40), the cell state is always updated to `CANCELLED` and a special `onCancelledCell()` function is invoked on the segment (lines 38–39). This `onCancelledCell()` function signals that one more cell in this segment was cancelled and removes the segment if all the cells become cancelled (see Appendix B for details).

With smart cancellation, `onCancellation()` is invoked first (line 42). If it succeeds (returns `true`), then the cell state can be moved to `CANCELLED`. However, a concurrent `resume(..)` may come and replace the aborted thread with its resumption value, see the cell state diagram in Figure 4. Therefore, we put the `CANCELLED` token via an atomic `GetAndSet` operation, which returns the previous cell state (line 45). If a thread instance was stored in the cell (line 47), `resume(..)` has not come there: the handler signals about a new cancelled cell, removing the segment if needed (line 48), and finishes. Otherwise, if a resumption value was stored in the cell, the cancellation handler completes the corresponding resumption by invoking `resume(..)` with this value (line 52).

In case `onCancellation()` returns `false` (line 53), the matching `resume(..)` should be refused. Thus, the cell state moves to `REFUSE` via an atomic `GetAndSet` (line 55). If the cell stored the cancelled thread, the handler finishes (line 57). Otherwise, a concurrent `resume(..)` has replaced it with the resumption value — we complete it with `completeRefusedResume(..)` (line 61).

4 Synchronization Primitives on Top of CQS

To show the expressiveness of the `CancellableQueueSynchronizer` framework, we present several algorithms developed on top of it. Starting with the barrier, we present a new count-down-latch algorithm, then several semaphore algorithms, and finish with blocking pools.

4.1 Barrier

A simple but popular synchronization abstraction is the *barrier*, which allows a set of parallel threads wait for each other at a common program point, via a provided `arrive()` operation.

Algorithm. Listing 7 on the right presents the algorithm on top of CQS. The implementation is straightforward: it maintains a counter of the parties who arrived (line 2) and increments it in the beginning of the `arrive()` operation (line 5). All but the last `arrive()` invocations suspend (line 6), while the latter one resumes all those who previously arrived (line 7).

```
1 val cqs = CQS<Unit>()
2 var remaining: Int = parties
3
4 fun arrive() {
5   r := FAA(&remaining, -1)
6   if r > 1: return cqs.suspend()
7   repeat(parties - 1) { cqs.resume(Unit) }
8 }
```

Listing 7: Barrier algorithm via CQS.

Once the last thread arrives, all the waiters should be resumed. However, if any of these waiters becomes cancelled, the barrier contract is violated — fewer waiters will be successfully resumed and overcome the barrier. Unfortunately, solving this problem would require an ability to *atomically* resume a set of waiters (so either all the waiters are resumed or none), but no real system provides such a primitive. Thus, similarly to the implementation in Java, we do not support cancellation. However, instead of breaking the barrier when a thread is cancelled, we ignore cancellation. The intuition behind this design is that even if a waiter has been cancelled, it has successfully reached the barrier point and should not block the other parties from continuing.

4.2 Count-Down-Latch

The next synchronization primitive we consider is the *count-down-latch*, which allows waiting until the specified number of operations are completed. It is initialized with a given count, and each `countDown()` invocation decrements the number of operations yet to be completed. Meanwhile, the `await()` operation suspends until the count reaches zero.

Basic Algorithm. The pseudocode of our count-down-latch implementation is presented in Listing 8. Essentially, the latch maintains two counters: `count`, representing the number of remaining operations (line 5), and `waiters`, which stores the number of pending `await()`-s (line 7).

```
1  val cqs = CQS<Unit>(
2   cancellationMode = SMART
3  )
4  // initialized by user
5  var count: Int = initCount
6  // the number of waiters
7  var waiters: Int = 0
8
9  fun countDown() {
10  r := FAA(&count, -1)
11  // Has the counter reached zero?
12  if r <= 1: resumeWaiters()
13 }
14
15 fun await() {
16  if count <= 0: return
17  w := FAA(&waiters, +1)
18  // Is DONE_BIT set?
19  if w & DONE_BIT != 0: return
20  // Suspend until count reaches zero
21  cqs.suspend()
22 }
23 fun resumeWaiters() = while(true) {
24  w := waiters
25  // Is DONE_BIT set?
26  if w & DONE_BIT != 0: return
27  // Set DONE_BIT and resume waiters.
28  if CAS(&waiters, w, w | DONE_BIT):
29    repeat(w) { cqs.resume(Unit) }
30    return
31 }
32
33 fun onCancellation(): Bool {
34  w := FAA(&waiters, -1)
35  // Move the cell to CANCELLED if the
36  // bit is unset; otherwise, to REFUSE.
37  return w & DONE_BIT == 0
38 }
39
40 fun completeRefusedResume(token: Unit) {
41  // Ignore cancelled await()-s.
42 }
43
44 const DONE_BIT = 1 << 31
```

Listing 8: Count-down-latch implementation on top of CQS with smart cancellation. When manipulating with `DONE_BIT`, we use bitwise "and", "or", and "left shift" operators, denoted as **&**, **|**, and **<<**, respectively.

The `countDown()` function is straightforward: it decrements the number of remaining operations (line 10), resuming the waiting `await()`-s if the count reached zero (line 12).[5] Meanwhile, `await()` checks whether the counter of remaining operations has already reached zero, immediately completing in this case (line 16). If `await()` observes that `count` is positive, it increments the number of waiters (line 17) and suspends (line 21).

Given that `resumeWaiters()`, which is invoked by the last `countDown()`, can be executed concurrently with `await()`, they should synchronize. For this purpose, `resumeWaiters()` sets the `DONE_BIT` in the `waiters` counter (line 28), forbidding further suspensions and showing that this count-down-latch has reached zero. Thereby, `await()` checks for this `DONE_BIT` before suspension and completes immediately if the bit is set (line 19).

Cancellation. The simplest way to support cancellation is to do nothing: the algorithm already works with the simple cancellation mode, where `resume(..)`-s silently fail on cancelled `await()` requests (line 29). This strategy results in resuming cancelled waiters, which makes `resumeWaiters()` work in a linear time on the total number of `await()` invocations, including the aborted ones.

Smart cancellation, on the other hand, makes it possible to optimize `resumeWaiters()` so that the number of steps is bounded by the number of non-cancelled `await()`-s. The `onCancellation()` function is invoked when a waiter becomes cancelled. It attempts to decrement the number of waiters (line 34), making `resume(..)` skip the corresponding cell in the CQS. However, if the `DONE_BIT` is already set at the moment of the decrement, a concurrent `resumeWaiters()` is going to resume this cancelled waiter. The corresponding `resume(..)` call should be ignored, so `onCancellation()` returns `false`, while `completeRefusedResume(..)` does nothing (lines 40–42).

4.3 Semaphores

The barrier and count-down latch algorithms described above do not actually require waiting requests to be resumed in FIFO order. However, this property is critical for some primitives such as the mutex or the semaphore. While the mutex allows at most one thread to be in the critical section protected by `lock()` and `unlock()` invocations, the semaphore is a generalization of mutex that allows the specified number of threads to be the critical section simultaneously by taking a permit via `acquire()` and returning it back via `release()`.

In fact, the semaphore algorithm is almost the same as the one for the mutex, presented under the CQS framework presentation in Listing 3 (the basic version) and Listing 5 (the cancellation part). The only difference is that the `state` field is initialized with K instead of 1, when K is the number of threads allowed to be in the critical section concurrently. We present the implementation details in Appendix C.

[5] We allow the number of `countDown()` calls to be greater than the initially specified one. However, we could check in `countDown()` that the counter is still non-negative after the decrement, throwing an exception otherwise.

4.4 Blocking Pools

While the previous algorithms use `CancellableQueueSynchronizer` only for synchronization, it is also possible to develop *communication* primitives on top of it. Here, we discuss two blocking pool implementations. When using expensive resources such as database connections or sockets, it is common to reuse them — this usually requires an efficient and accessible mechanism. The *blocking pool* abstraction maintains a set of elements that can be retrieved in order to process some operation, after which the element is placed back in the pool:

- `take()` either retrieves one of the elements (in an unspecified order), suspending until an element appears if the pool is empty;
- `put(element)` either resumes the first waiting `take()` operation and passes the element to it, or puts the element into the pool.

Intuitively, the blocking pool contract reminds the semaphore one. Similarly to the semaphore, it transfers resources, with the only difference being that the semaphore shares logical non-distinguishable permits while blocking pool works with real elements. The rest is almost the same. In Appendix C, we present two pool implementations: queue-based and stack-based. Intuitively, the queue-based implementation is faster since queues can be built on segments similar to CQS and leverage `Fetch-And-Add` on the contended path. In contrast, the stack-based pool retrieves the last inserted, thus the "hottest" element. Please note that both algorithms we discuss are *not* linearizable and can retrieve elements out-of-order under some races. However, since pools do not guarantee that the stored elements are ordered, these queue- and stack-based versions should be considered bags with specific heuristics; these semantics matches practical applications.

5 Correctness and Progress Guarantees

In this section, we discuss correctness and progress guarantees for both CQS operations and the primitives we built on top of the framework.

5.1 Formal Proofs of Correctness in Iris/Coq

Correctness is formally proven in the state-of-the-art concurrent higher-order separation logic Iris [17] using its Coq formalization [16]. Here, we highlight the key ideas behind the proofs and discuss their limitations. **The source code of the proofs is available on GitHub [27]. We complement this with a detailed outline in Appendix E.**

Operation Specifications. Iris is a framework designed for reasoning about the safety of concurrent programs, and several non-trivial algorithms have already been formally proved using it [18, 19, 20, 21, 22, 28, 29]. When constructing formal proofs, one should provide a specification for each of the data structure operations. In the Iris logic, operations manipulate *resources*, which are pieces of knowledge about the system-wide state and can be held by threads or the data structure itself. These resources are *logical* and do not affect the program execution. A specification describes which resources are required for the operation to start and how they change when it finishes.

Consider again the mutex as an example. Intuitively, we parameterize it with a resource R, which serves as an exclusive right to be in the critical section and, thus, to invoke `unlock()`. Initially, this resource R is held by the mutex object. The specification ensures that:

1. when the `lock()` operation finishes, the resource R is transferred to the caller thread, and
2. when `unlock()` starts, the corresponding thread must provide R.

If the resource R is unique, the specification still holds; however, as it cannot be held by multiple threads by construction, the mutual exclusion contract is satisfied.

All our specifications are defined in a similar manner. For example, to specify the semaphore contract, we simply need to maintain K non-distinguishable copies of R; thus, allowing at most K threads to enter the critical section. However, the actual specifications in Coq contain many additional details, mainly due to support for cancellation semantics. Please refer to the proofs outline in Appendix E and the source code [27] for details.

Modularity. Our Iris proofs are *modular*: specifications treat each operation separately and do not concern the state of the system as a whole, locally manipulating logical resources instead. As a result, the proof of CQS itself spans 8000 lines of Coq; by comparison, the proof of the barrier, including its definition, takes only 400 lines, the semaphore proof requires less than 300 lines, and the proofs for the count-down-latch and blocking pools take up to 700 lines each. Modularity dramatically reduces the effort for someone wishing to formally verify their CQS-based primitive.

Limitations. One main limitation is that the existing formal specifications do not highlight the FIFO semantics, allowing the waiting operations to complete in any order. Instead, these specifications verify high-level properties, such as "at most one thread can be in the critical section" for the mutex. This limitation stems from the modularity of proofs and the fact that the user code parameterizes the cancellation handler in CQS. The fairness of end-to-end structures on top of the CQS is easy to see by the analogy with the state-of-the-art linearizable queues [14, 15], but proofs of such form are not modular. While the modular Iris proofs are powerful enough to show fairness, this requires significant effort even for simple data structures such as the classic Michael-Scott queue [20], and constructing them for non-trivial and, especially, higher-order structures like CQS is currently impractical. Most importantly, a modular proof of fairness of structures on top of the CQS would require placing highly involved contracts on the cancellation handler as well as the uses of `suspend` operations that may interact with it, making it significantly more difficult to prove the correctness of primitives on top of the CQS for the end user.

Another limitation of the provided Iris specifications is that they do not assert the lack of memory leaks. In particular, they do not prevent us from always storing the whole infinite array. Nevertheless, the lack of memory leaks follows by construction, as we always physically remove segments full of canceled cells. Beyond that, we have thoroughly tested our implementation for the absence of memory leaks via the Lincheck framework [30], which enables model checking of concurrent algorithms on the JVM.

Finally, we assume a strong sequentially-consistent memory model. We find this assumption reasonable as almost all the operations that manipulate shared data are atomic in the presented algorithms, while considering relaxed memory may significantly increase the proofs complexity [31, 32]. We also rely on the SC-DRF (sequential consistency for data-race-free programs) property of all real-world weak memory models, such as C++11 and JMM, which makes reasoning in the strong memory model sufficient. However, we plan to extend our proofs to support the release-acquire semantics [32] and, thus, match the LLVM memory model for languages such as C/C++ and Rust.

5.2 Progress Guarantees

Similarly to the dual data structures formalism [33], we reason about progress independently of whether the operation was suspended. When we say that some blocking operation is lock- or wait-free, we mean that it performs all the synchronization with this progress guarantee, either completing immediately or adding itself to the queue of waiters followed by suspension.

Unfortunately, the progress guarantees cannot be mechanized in our Iris proofs. The reason for this is that, at the time of writing, there are two forms of specifying program behavior in Iris. The first way is to use *(partial) weakest preconditions*, which do not ensure that an operation terminates. In fact, an infinite loop satisfies any such specification. The second less popular form uses the *total weakest precondition* [17], which requires that every operation must terminate in a bounded number of steps. This type of specification can be used to show wait-freedom of algorithms, but is not applicable to our case, as some of the operations guarantee only lock-freedom.

We do not consider the lack of formal proof of progress guarantees a major issue. Although it is possible to write such proofs (see [34] for a comprehensive analysis), we find it much easier to discuss this question separately. In essence, most of the presented primitives including the CQS framework itself guarantee wait-freedom under no cancellation and at least lock-freedom when requests may abort. We provide a detailed analysis in Appendix D.

6 Evaluation

Our main practical contribution is integrating CQS, along with the mutex and semaphore implementations, into the standard Kotlin Coroutines library [6]. Other presented synchronization and communication primitives are implemented in tests, enabling their fast development when needed.

To validate performance, we implemented `CancellableQueueSynchronizer` on the JVM and compared it against the state-of-the-art `AbstractQueuedSynchronizer` framework for implementing synchronization primitives in Java [3]. The latter provides similar semantics to CQS, and is the only practical framework that addresses the same general problem. Notably, CQS-based algorithms are significantly more straightforward to reason. For fair performance evaluation, we use threads as waiters in CQS; it should benefit the Java implementation, which is well-optimized for this case.

Our implementations for coroutines in Kotlin and native threads in Java confirm the flexibility of our design, let alone matching the real-world semantics.

Experimental Setup. Experiments were run on a server with 4 Intel Xeon Gold 6150 (Skylake) sockets; each socket has 18 2.70 GHz cores, each of which multiplexes 2 hardware threads, for a total of 144 hardware threads.

We used `OpenJDK 15` in all the experiments and the `Java Microbenchmark Harness (JMH)` library [35] for running benchmarks. When measuring operations, we also add some uncontended work after each operation invocation — the work size is geometrically distributed with a fixed mean, which we vary in benchmarks. In our CQS implementation, we have chosen the segment size of 64 based on minimal tuning.

6.1 Barrier

We compare the CQS-based barrier implementation with the standard one in Java. In addition, we add a baseline counter-based solution, which is organized in the same way as ours, but performs active waiting instead of suspension, spinning in a loop until the `remaining` counter becomes zero.

Benchmark. Each of the threads performs barrier point synchronizations followed by some uncontended work. This process is repeated a fixed number of times. We measure a single synchronization phase, a set of `arrive()`-s with additional work for each thread. Without any synchronization, the execution time is expected to stay the same independently of the number of threads.

Results. The experimental results are presented in Figure 5. We evaluated all three algorithms on various numbers of threads and with three average work sizes — `100`, `1000`, and `10000` uncontended loop iterations on average. The graphs show an average time per operation, so lower is better.

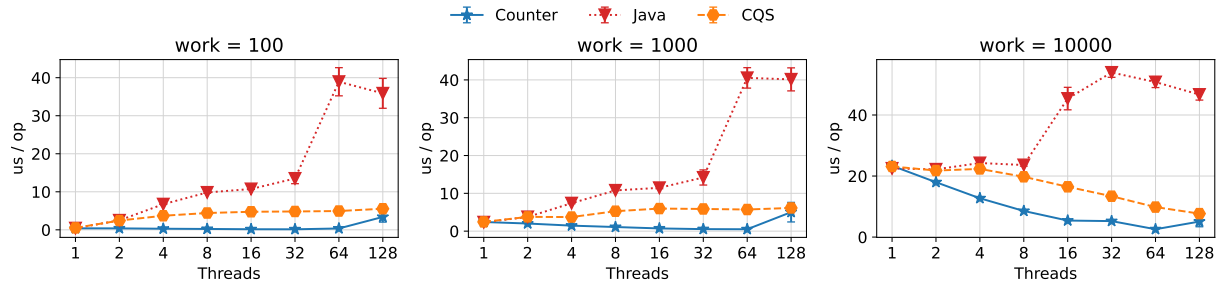

Figure 5: Evaluation of `CancellableQueueSynchronizer`-based barrier implementation against the standard one available in Java and a simple counter-based solution with active waiting. The plots show average time per synchronization phase, **lower is better**.

Since one of the main synchronization costs is thread suspensions and resumptions, the counter-based solution with active waiting is predictably the fastest. Nevertheless, our CQS-based algorithm shows similar performance trends, due to all the operations being based on `Fetch-And-Add`. In contrast, the solution from the standard concurrency library in Java shows significantly less scalability—we find the reason for such performance degradation in using a mutex under the hood; surprisingly, it does not use `AbstractQueuedSynchronizer` directly. As a result, we find our simple CQS-based algorithm to provide superior performance.

6.2 Count-Down-Latch

Next, we evaluate our count-down latch implementation against the one in Java's concurrency package, which is built on top of the `AbstractQueuedSynchronizer` framework.

Benchmark. We consider a workload with a fixed number of `countDown()` invocations distributed among threads, each followed by additional uncontended work. Besides, we add a baseline that does not invoke `countDown()` and only performs the work. Thus, comparing with this baseline we can measure the overhead caused by the count-down-latch synchronization.

Results. Figure 6 shows the evaluation results with different additional work sizes (50 uncontended loop iterations on average on the left, 100 in the middle, and 200 on the right). It is apparent that the CQS implementation significantly outperforms the standard one from Java, by up to $4\times$. Compared to the baseline, it follows the same trend, providing an extremely small overhead on the right graph, where the work is 200 uncontended loop cycles.

Similar to our CQS-based algorithm, the implementation in Java maintains a counter of remaining `countDown()` invocations. However, they update this counter in a `CAS` loop: the algorithm reads the current counter value and tries to replace it with the reduced by one via `CAS`, restarting the process on failure. We find this difference the main reason for the superior scalability of our solution.

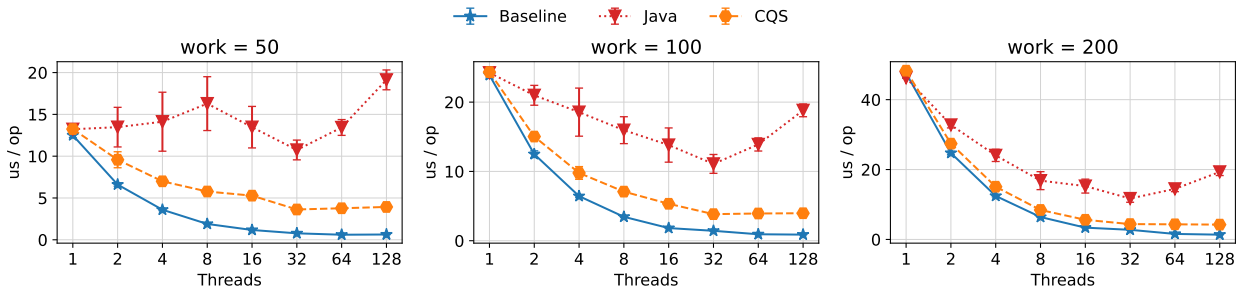

Figure 6: Evaluation of the CQS-based count-down-latch implementation against the standard one in Java. The baseline illustrates how the operation time would change if `coundDown()` took no time. **Lower is better.**

6.3 Mutex and Semaphores

Since the semaphore is a generalization of the mutex, we equate its implementation with $K = 1$ permits as mutual exclusion. We compare our algorithm against alternatives from the standard Java library, unfair implementations of mutex and semaphore in Java, and the state-of-the-art fair CLH and MCS lock algorithms. In Section 2 we also mention that implementing non-blocking `Mutex.tryLock()` and `Semaphore.tryAcquire()` operations would require extending CQS with a special *synchronous* resumption mode, leaving the details to Appendix A. We included both semaphore implementations in the experiment to show that the complexity introduced by this `synchronous` resumption mode does not affect performance.

Benchmark. Consider the workload of many operations to be executed by the specified number of threads with the parallelism level restricted via semaphore. Thus, each operation invocation is wrapped with the `acquire()-release()` pair. When the parallelism level equals `1`, the semaphore is de facto a mutex, so we can compare our semaphore against other mutex algorithms. As before, the operations are simulated with uncontended geometrically distributed work. In addition, we perform some work before acquiring a permit, thus, simulating a preparation phase for the operation guarded by the semaphore. We used `100` uncontended loop iterations on average for both pieces of work; the results for other work sizes do not differ significantly and, therefore, are omitted.

Results. The results against both fair and unfair versions of the standard `ReentrantLock` and `Semaphore` primitives in Java, as well as against the classic CLH [13] and MCS [36] fair locks, are shown in Figure 7. Our semaphore implementation with the *synchronous* resumption mode in CQS, which enables `tryAcquire()` implementation, is denoted with the suffix ¡¡Sync¿¿.

In the mutex scenario, all the fair algorithms show the same performance, while Java's unfair mutex and semaphore are predictably faster, as unfairness significantly reduces context switches under high contention.

In the semaphore scenario, our solution outperforms both fair and unfair Java implementations by up to `4x` when the number of threads does not exceed the number of permits (so suspensions do not happen). The main reason is that our solution leverages `Fetch-and-Add` to update the number of available permits, which can be negative, indicating the number of waiters. In contrast, the implementation in Java must ensure that this number stays non-negative, so it has to perform this update in a `CAS` loop, reading the current number of available permits, trying to decrement it via `CAS` if there is a permit to acquire, and restarting if the `CAS` fails.

With the increase in the number of threads, our algorithm is almost on par with the unfair version when the number of permits $\geq$`16`, and significantly outperforms the fair one in all scenarios. In particular, our semaphore implementation outperforms the standard fair algorithm in Java by up to `90x` in a highly-contended scenario. More scalable queue design behind CQS is the key to achieving such an outstanding performance. Notably, the complexity introduced by the synchronous resumption is negligible and does not affect results.

6.4 Blocking Pools

We implemented both queue- and stack-based pools and compared them against the existing `ArrayBlockingQueue` (both fair and unfair) and `LinkedBlockingQueue` collections from the standard Java library. Notably, they do not leverage the `AbstractQueuedSynchronizer` framework, as it serves only for synchronization, while CQS enables communication out-of-the-box. Relatively, their solutions provide linearizability, while our pools may be non-linearizable when threads abort. This experiment considers all data structures as solutions for pools of shared resources.

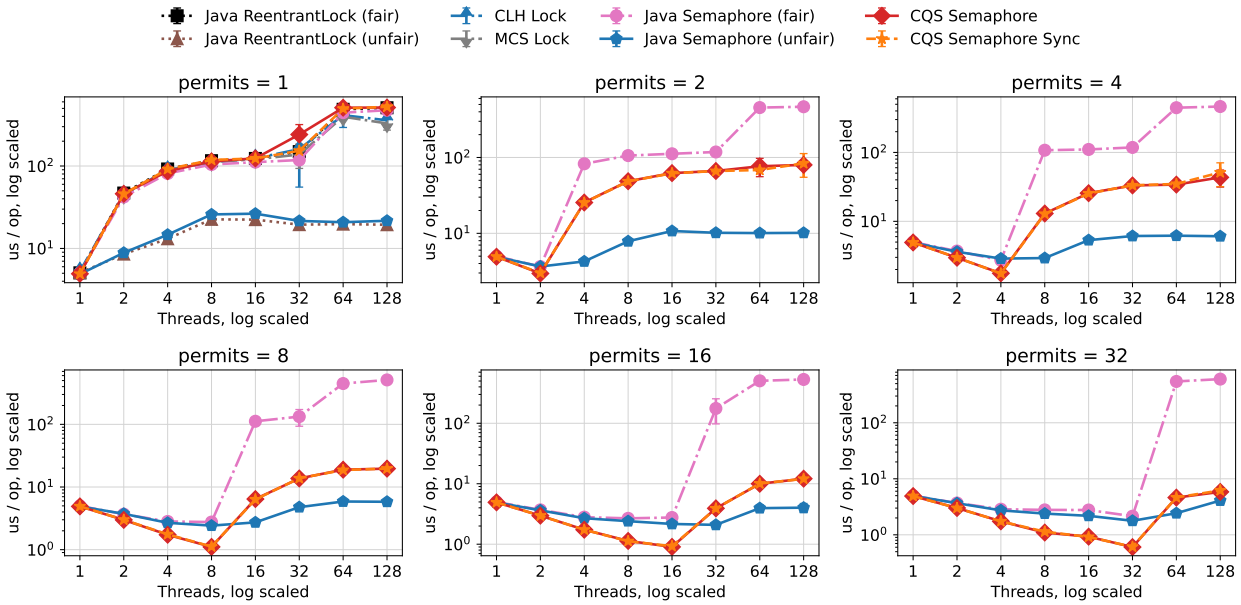

Figure 7: Evaluation of CQS-based semaphore implementations compared to the standard ones in Java, including the unfair variants. In addition, we compare our semaphores against the standard fair and unfair lock implementations in Java and the classic CLH and MCS fair mutexes. **Lower is better.**

Benchmark. We use the same benchmark as for semaphores. In essence, we run many operations on the specified number of threads with a shared pool of elements. Each operation performs some work (`100` uncontended loop iterations on average in our experiment) first, then takes an element, performs some other work with this element (`100` more loop iterations on average in our experiment), and returns it to the pool at the end. The results with other work amounts are omitted but were examined and do not differ significantly.

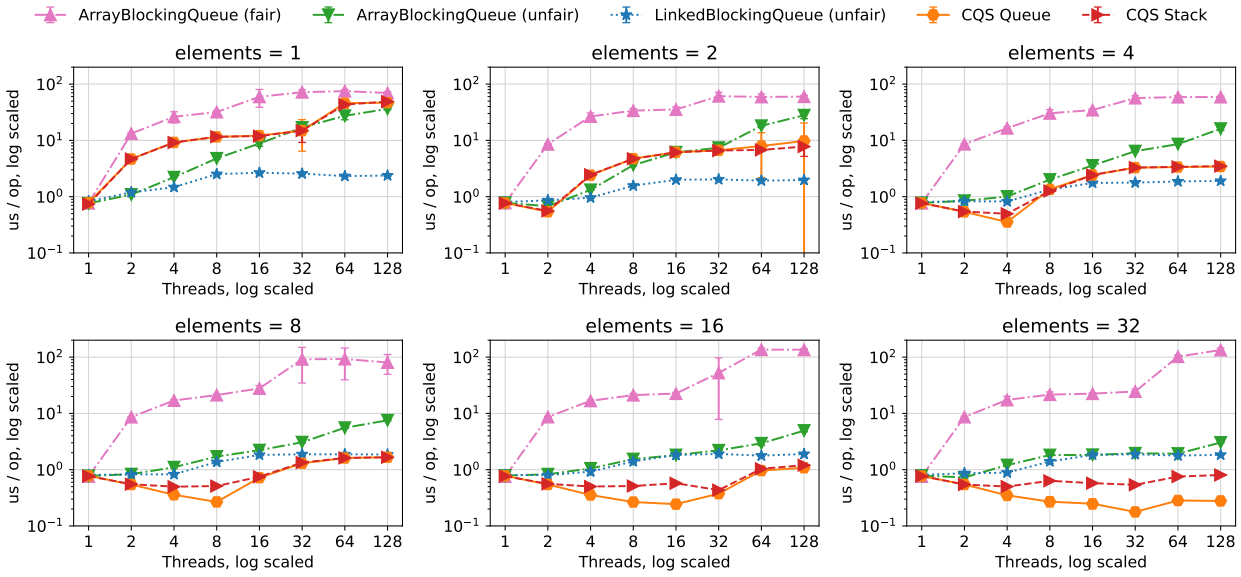

Figure 8: Evaluation of the presented queue- and stack-based pool algorithms with various numbers of shared elements against the existing `ArrayBlockingQueue` (both fair and unfair versions) and `LinkedBlockingQueue` collections from the standard Java library. **Lower is better.**

Results. Figure 8 shows results with different numbers of elements shared in the pool. First, our queue-based version shows better results on larger numbers of elements, which is expected as the queue perform a `FAA` on the contended path instead of `CAS` in the stack-based solution; the latter often fails under high contention, resulting in the operation restart.

Compared to the fair `ArrayBlockingQueue`, both of our implementations are more performant by up to `100`× times. The synchronization behind `ArrayBlockingQueue` uses coarse-grained locking, while our solution is non-blocking for storing elements and managing the queue of waiting requests.

The *unfair* `LinkedBlockingQueue` is more scalable than the *unfair* version of `ArrayBlockingQueue`, and they slightly outperform our *fair* implementations on a large number of threads with a small number of shared elements, which is when our solutions suspend a lot. However, both our solutions consistently outperform these unfair primitives by up to `10`× times when at least `8` elements are shared, showing the same or better performance when the number of threads does not exceed the number of elements.

6.5 Abortability Support

Up to this point, we have primarily focused on situations where suspended requests do not get aborted. While cancellation performance might not always be crucial, as it typically occurs due to a more resource-intensive coroutine or thread interruption, removing aborted waiters from the queue in constant time remains essential. This is particularly true for coroutines, where thousands may be waiting on a mutex or semaphore. The CQS framework fulfills this need by physically removing aborted threads in $O(1)$ under no contention. In contrast, Java's `AbstractQueuedSynchronizer` takes linear time in the queue size to remove an interrupted thread.

To assess the practical impact of constant-time cancellation, we conducted a benchmark comparison between CQS (with both `SIMPLE` and `SMART` cancellation modes) and `AbstractQueuedSynchronizer`. Initially, we populate the synchronization framework with a specified number of suspended threads. After that, we measure the time required to suspend and instantly abort, without parking the native thread. Based on the initial queue size, we anticipate that the performance of Java's solution will degrade, while CQS should consistently deliver the same level of performance. Figure 9 displays the results, which confirm our hypothesis. In particular, CQS outperforms Java's solution by `1.9x` on an empty queue, and by `65x` when the queue contains 1000 waiters.

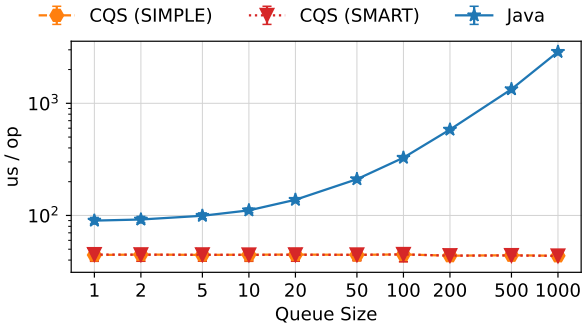

Figure 9: Comparison of CQS' and Java's `AbstractQueuedSynchronizer` cancellation mechanisms. **Lower is better.**

Regrettably, it is not feasible to compare the cancellation mechanisms of CQS and `AbstractQueuedSynchronizer` under concurrent conditions at the time of writing the paper. The reason is that the latter gets into a deadlock in its internal `cleanQueue()` function.

7 Related Work

Our work is part of a wider effort of formalizing and implementing expressive, safe, and efficient support for asynchronous operations in modern programming languages [37, 38, 39, 40, 41]. In this context, we provide contributions at the level of algorithms, semantics, and formal proofs, with Kotlin/JVM as a practical application. Specifically, we perform one of the first thorough explorations of how fairness and abortability semantics can be efficiently supported at the data structure level, and present one of the first formally-verified such designs for this type of data structure. Importantly, our approach enables high-performance implementations in a range of practical applications, and could serve as a basis for standard library implementations in modern languages.

We emphasize that few real-world implementations of similar complexity are formally verified [18, 19, 20, 21, 22]. In line with prior work, we do not formally prove full linearizability, which is notoriously difficult to approach in Iris but can be demonstrated through classical proofs. There are successful linearizability proofs of data structures of comparable complexity using the approach of *contextual refinement* [20, 28] using the ReLoC proof framework. Iris itself permits making specifications *logically atomic*, which can also represent linearizability [42]. We find this approach much less applicable to ensuring linearizability of a *framework*, which depends heavily on the behavior of the code passed to it.

At the algorithmic level, our CQS implementation builds on ideas from both the classic Michael-Scott queue [26] and the highly-efficient LCRQ queue design of Afek and Morrison [14]. The latter was also used by Izraelevitz and Scott [4] to build blocking synchronous queues, and by Koval et al. [43, 25] to build channels. Relative to these latter modern works, CQS supports much more general semantics, requiring significant changes to the design, in particular,

to support cancellations. Specifically, CQS is general enough to provide full support for coroutines, while staying flexible and efficient, whereas these prior design focus on narrower applications, such as blocking queues.

To our knowledge, the only abstraction that provides similarly-general semantics is the `AbstractQueuedSynchronizer` in Java [3], which CQS outperforms by a wide margin due to superior algorithmic design, complemented by formal proofs. More precisely, the `AbstractQueuedSynchronizer` framework combines the classic CLH [13] lock algorithm to maintain the queue of suspended requests with an integer counter, which represents the synchronization primitive state and is updated by `CAS` operations. In contrast, the CQS enables more efficient state updates via `Fetch-And-Add`-s, also maintaining the queue of waiters with `FAA`-s on the contended path; thus, providing a more scalable solution.

8 Discussion

We have presented a new `CancellableQueueSynchronizer` framework enabling efficient implementations for a whole range of fundamental synchronization primitives in a fair and abortable manner. We observed that the interplay between fairness and cancellation semantics can raise subtle semantic and correctness questions. We found formalization extremely useful when identifying correctness issues in our implementation, notably w.r.t. cancellation semantics. A practical consequence of our work is efficient support for such primitives in the context of Kotlin Coroutines, which we show to generally outperform existing designs offering similar semantics in a wide range of scenarios. Specifically, our algorithms on top of CQS outperform existing Java implementations in almost all scenarios and can be faster by orders of magnitude. Surprisingly, the CQS-based primitives frequently surpass even the *unfair* versions of primitives from the standard Java library in our experiments, thanks to the superior scalability of our design.

We believe that CQS could serve as a basis for more complex semantics, designs, and primitives (e.g., fair readers-writer locks and synchronous queues), enabling efficient synchronization not only for Kotlin Coroutines but for other languages and platforms as well, such as C++, Rust, and Go. We plan to investigate this in future work, along with proof extensions to the release-acquire memory model semantics [32].

References

[1] Edsger W Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer, 2001.

[2] Donald E Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966.

[3] Doug Lea. The `java.util.concurrent` synchronizer framework. *Science of Computer Programming*, 58(3):293–309, 2005.

[4] Joseph Izraelevitz and Michael L Scott. Generality and speed in nonblocking dual containers. *ACM Transactions on Parallel Computing (TOPC)*, 3(4):1–37, 2017.

[5] Gilles Kahn and David MacQueen. Coroutines and networks of parallel processes. 1976.

[6] Kotlin Coroutines. `https://github.com/Kotlin/kotlin-coroutines`, 2022.

[7] Prasad Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 295–304, 2003.

[8] Hyonho Lee. Fast local-spin abortable mutual exclusion with bounded space. In *International Conference On Principles Of Distributed Systems*, pages 364–379. Springer, 2010.

[9] Robert Danek and Wojciech Golab. Closing the complexity gap between fcfs mutual exclusion and mutual exclusion. *Distributed Computing*, 23(2):87–111, 2010.

[10] Adam Alon and Adam Morrison. Deterministic abortable mutual exclusion with sublogarithmic adaptive rmr complexity. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 27–36, 2018.

[11] George Giakkoupis and Philipp Woelfel. Randomized abortable mutual exclusion with constant amortized rmr complexity on the cc model. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 221–229, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Abhijeet Pareek and Philipp Woelfel. Rmr-efficient randomized abortable mutual exclusion. In *International Symposium on Distributed Computing*, pages 267–281. Springer, 2012.

[13] Peter Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*, pages 165–171. IEEE, 1994.

[14] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM.

[15] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. *SIGPLAN Not.*, 51(8):16:1–16:13, February 2016.

[16] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 205–217, 2017.

[17] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[18] Morten Krogh-Jespersen, Thomas Dinsdale-Young, and Lars Birkedal. Verifying a concurrent data-structure from the dartino framework in iris. 2016.

[19] Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. Verifying concurrent search structure templates. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–196, 2020.

[20] Simon Friis Vindum and Lars Birkedal. Contextual refinement of the michael-scott queue (proof pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 76–90, 2021.

[21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[22] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M Frans Kaashoek, and Nickolai Zeldovich. Gojournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th Symposium on Operating Systems Design and Implementation (OSDI). Virtual, 423–439*, 2021.

[23] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15:491 – 504, 07 2004.

[24] Pedro Ramalhete and Andreia Correia. Brief announcement: Hazard eras — non-blocking memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 367–369, 2017.

[25] Nikita Koval, Dan Alistarh, and Roman Elizarov. Fast and scalable channels in kotlin coroutines. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 107–118, New York, NY, USA, 2023. Association for Computing Machinery.

[26] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[27] Cqs formal proofs. `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs`, 2023.

[28] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. Mechanized verification of a fine-grained concurrent queue from meta's folly library. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 100–115, 2022.

[29] Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W O'Hearn, and Francesco Zappa Nardelli. Applying formal verification to microkernel ipc at meta. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 116–129, 2022.

[30] Nikita Koval, Maria Sokolova, Alexander Fedorov, Dan Alistarh, and Dmitry Tsitelov. Testing concurrency on the jvm with lincheck. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 423–424, 2020.

[31] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. Rustbelt meets relaxed memory. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, 2019.

[32] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in iris. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[33] William N Scherer III, Doug Lea, and Michael L Scott. Scalable synchronous queues. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 147–156. ACM, 2006.

[34] Xiao Jia, Wei Li, and Viktor Vafeiadis. Proving lock-freedom easily and automatically. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, page 119–127, New York, NY, USA, 2015. Association for Computing Machinery.

[35] JMH - Java Microbenchmark Harness. `https://openjdk.java.net/projects/code-tools/jmh/`, 2021.

[36] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[37] Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. Pause'n'play: Formalizing asynchronous c#. In *European Conference on Object-Oriented Programming*, pages 233–257. Springer, 2012.

[38] Semih Okur, David L Hartveld, Danny Dig, and Arie van Deursen. A study and toolkit for asynchronous programming in c#. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1117–1127, 2014.

[39] Aleksandar Prokopec and Fengyun Liu. Theory and practice of coroutines with snapshots. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 3:1–3:32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[40] Philipp Haller and Heather Miller. A reduction semantics for direct-style asynchronous observables. *Journal of Logical and Algebraic Methods in Programming*, 105:75–111, 2019.

[41] Zak Cutner and Nobuko Yoshida. Safe session-based asynchronous coordination in rust. In *International Conference on Coordination Languages and Models*, pages 80–89. Springer, 2021.

[42] Various authors. `https://gitlab.mpi-sws.org/iris/examples/-/tree/0260d3d08e2f56bbccd44c3d56436baea30da4c9/theories/logatom`, 2022.

[43] Nikita Koval, Dan Alistarh, and Roman Elizarov. Scalable FIFO channels for programming via communicating sequential processes. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*, volume 11725 of *Lecture Notes in Computer Science*, pages 317–333. Springer, 2019.

A Synchronous Resumption

In section 2 we briefly mentioned that supporting non-blocking variants of blocking operations, such as the `tryLock()` one in the mutex, would require introducing a special `synchronous resumption` mode. Indeed, the classic CQS framework does not provide a way to implement such operations correctly. Therefore, first, we describe the problem and introduce the *synchronous resumption* mode in the absence of cancellation in Subsection A.1. After that, we extend it with abortability support in Subsection A.2.

A.1 Extension to the Basic CQS

In Section 2, we presented a mutex algorithm on top of CQS. We copy it in Listing 9 for convenience. While this mutex algorithm is simple, extending it to allow a `tryLock()` operation is not obvious. Specifically, `tryLock()` should attempt to acquire the lock and return `true` or `false` depending on whether it succeeded, attempting to change the logical state from "unlocked" to "locked" and never manipulating CQS. However, the `unlock()` implementation in Listing 9 relies on leaving a right to acquire the lock (a "permit") as metadata in a CQS cell. A naive implementation of `tryLock()`, which tries to atomically update `state` from `1` ("unlocked") to `0` ("locked"), would not observe that the permit in CQS, resulting in an incorrect execution.

```
1  val cqs = CQS<Unit>()
2  // 1 - unlocked, ≤ 0 - # of waiters
3  var state: Int = 1 // "unlocked"
       intially
4  fun lock() {
5    s := FAA(&state, -1)
6    // Is the lock just acquired?
7    if s > 0: return
8    cqs.suspend() // suspend otherwise
9  }
10 fun unlock() {
11   s := FAA(&state, +1)
12   // Resume the first waiting
13   // request if there is one.
14   if s < 0: cqs.resume(Unit)
15 }
```

Listing 9: The basic mutex algorithm without cancellation support using the CQS framework. This is a copy of the algorithm in Listing 3.

To illustrate, consider the execution in Figure 10. First, a new mutex is created, and `lock()` is invoked. Then, two parallel threads start. The thread on the right also invokes `lock()`, decrementing `state` first. Since the mutex is already acquired, it then invokes `cqs.suspend()`. However, the execution switches to the thread on the left between the `state` decrement and the `suspend()` invocation. Then, the `unlock()` invocation changes `state` from `-1` to `0` and invokes `cqs.resume(..)`. Since the conjugate `suspend()` has not been invoked yet, it puts `Unit` in the first cell and completes. Thus, the mutex is actually in the "unlocked" state, while the permit to acquire this mutex is stored not in the `state` field but in the first cell of the CQS. Therefore, the following invocation of `tryLock()` fails since `state` equals `0`, while the `lock()` call succeeds — it goes to the first cell and completes immediately.

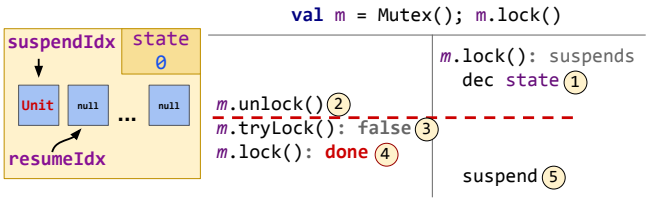

Figure 10: Incorrect behaviour of the mutex from Listing 9 extended with `tryLock()` that tries to atomically change `state` from 1 (unlocked) to 1 (locked, no waiters).

`ASYNC` and `SYNC` Resumption Modes. The problem stems from the `unlock()` behaviour when it intends to pass the lock to a parallel `lock()` request — another `lock()` operation that happens after this `unlock()` may illegally acquire the lock from the CQS cell due to a race.

To prevent such a race, we should avoid leaving "permits" in cells "unattended". As a solution, two resumption modes are proposed — *asynchronous* and *synchronous*. The *asynchronous* mode (`ASYNC` in code) is the standard mode where `resume(..)` puts the value into an empty cell and completes immediately, thus transferring this value asynchronously. In contrast, the *synchronous* mode (`SYNC` in code) forces `resume(..)` to wait until the value is taken, and marks the cell as `broken` after some bounded time if the value is still not taken by a concurrent `suspend(..)`. The idea is similar to breaking cells in modern queues [14, 15]. In this case, both `resume(..)` and `suspend()` operations manipulating this cell fail, so `suspend(..)` returns `null` while `resume(..)` returns `false`. The intuition is that allowing broken cells keeps the balance of paired operations, such as `lock()` and `unlock()`, so they should simply restart. Figure 11 describes the modified cell life-cycle.

Modifications to `suspend()` and `resume(..)`. Listing 10 shows the updated versions of `suspend()` and `resume(..)` operations; the key changes from the basic algorithm in Listing 2 in Section 2 are highlighted with yellow. The semantics change in that both these operations may fail if the cell is broken; thus, `suspend()` can return `null`, while `resume(..)` returns `true` on success and `false` on failure.

```
1  // The resumption mode is specified
2  // by the user when constructing CQS.
3  val resumeMode: ASYNC or SYNC
4
5  val cells = InfiniteArray()
6  var suspendIdx: Int64 = 0
7  var resumeIdx:  Int64 = 0
8
9  fun suspend(): T {
10  i := FAA(&suspendIdx, +1)
11  // Try to suspend in cells[i].
12  t := currentThread()
13  if CAS(&cells[i], null, t):
14    return park() // enqueued, suspend
15  // Read the result and finish
16  // if the cell is not broken.
17  result := GetAndSet(&cells[i], TAKEN)
18  // Was the cell broken?
19  if result == BROKEN: return null
20  // The cell stored a value.
21  return result
22 }
```

```
23 fun resume(result: T): Bool {
24  i := FAA(&resumeIdx, +1)
25  t := cells[i]
26  if t == null: // is the cell empty?
27    // 'suspend()' is coming, try to
28    // install the result and finish.
29    if CAS(&cells[i], null, result):
30      // Finish in ASYNC mode.
31      if resumeMode == ASYNC: return true
32      // Synchronous resumption. Wait
33      // until the value is taken.
34      repeat(MAX_SPIN_CYCLES):
35        if cells[i] == TAKEN: return true
36      // The value has not been taken.
37      return !CAS(&cells[i], result,
38                  BROKEN)
39    // The cell stores a thread.
40    t = cells[i]
41  // Resume the waiting request.
42  cells[i] = RESUMED
43  t.unpark(result) // t is Thread
44 }
```

Listing 10: High-level CQS implementation without cancellation support but with support of both *asynchronous* and *synchronous* resumption modes. The key changes from the basic algorithm in Listing 2 in Section 2 are highlighted with yellow.

The `suspend()` operation does not change significantly. As before, it increments `suspendIdx` first (line 10), obtains the currently running thread (line 12), and tries to install it if the cell is empty (line 13). However, if the cell is not empty, it could have been broken and, thus, does not necessarily contain a value. Therefore, the algorithm replaces the current cell state with `TAKEN` via an atomic `GetAndSet(..)` operation (line 17), failing if the cell was in the `BROKEN` state (line 19). If the cell did contain a value, it is returned as the result of `suspend()` (line 21).

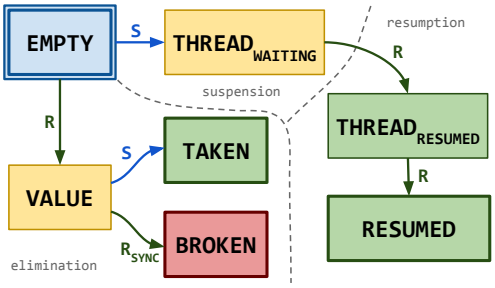

Figure 11: `CancellableQueueSynchronizer` cell life-cycle with *synchronous resumption mode* (the introduced transition is marked with R_{SYNC}) but without cancellation support. The edges marked with `S` correspond to the transitions by `suspend()`, and those marked with `R` represent transitions by `resume(..)`.

As for the `resume(..)` operation, like before, it increments `resumeIdx` first (line 24). When the cell is empty (line 26), `resume(..)` tries to set the resumption value to the cell (line 29). If the corresponding `CAS` fails, the cell stores a thread, which is read later at line 40. Otherwise, the value is successfully set, and further logic depends on the choice of resumption mode. In the asynchronous mode, `resume(..)` finishes immediately (line 31). In the synchronous mode, it waits in a bounded loop waiting for the value to be taken and finishes if it happens (lines 34–35). If the value has not been taken, the operation attempts to break the cell and fail, completing successfully if the value was taken after all and the corresponding `CAS` fails (line 37).

When the cell contains a thread instance, `resume(..)` cleans the cell (line 42), resumes the thread (line 43), and finishes.

Mutex Algorithm with `tryLock()`. Listing 11 contains a correct mutex implementation extended with the `tryLock()` operation. As discussed, the synchronous resumption mode is used (line 1) and both `lock()` and `unlock()` operations are wrapped in an infinite loop (lines 5 and 10) so that they restart if `suspend()` and `resume(..)` fail (lines 8 and 13). The remainder is the same as in the previous algorithm in Listing 9.

```
1  val cqs = CQS<Unit>(resumeMode = SYNC)
2  var state: Int = 0
3
4  fun tryLock(): Bool = CAS(&state, 1, 0)
5  fun lock(): Unit = while(true) {
6    s := FAA(&state, -1)
7    if s > 0: return
8    if cqs.suspend() == Unit :return
9  }
10 fun unlock(): Unit = while (true) {
11   s := FAA(&state, 1)
12   if s == 0: return
13   if cqs.resume(Unit): return
14 }
```

Listing 11: Basic mutex algorithm with `tryLock()` on top of CQS, without cancellation. This implementation is also correct with the simple cancellation mode presented in Subsection 3.1.

A.2 Cancellation Support

To support the *synchronous* resumption mode with cancellation, we need to ensure that `resume(..)` never leaves the value in CQS without making a rendezvous with `suspend()`. In the previous Subsection A.1, we figured out how the elimination part, when `resume(..)` comes to the cell before `suspend()`, should be modified. However, with *smart* cancellation, there is another way to leave the value in CQS — when the cell contains an aborted thread, the `resume(..)` implementation in Listing 6 in Section 3 can delegate its completion by replacing this cancelled thread with the resumption value (lines 19–20). With the *synchronous* resumption, we forbid this behaviour and wait in a spin-loop until the state changes to either `CANCELLED` or `REFUSE`.

The `resume(..)` Modification. The resulting pseudocode of the `resume(..)` operation is presented in Listing 12. The logic for `suspend()` stays the same as in Listing 10, while the cancellation handler is identical to the one in Listing 6.

As usual, the operation starts with incrementing `resumeIdx` (line 2). After that, if cell is in the empty state, the algorithm tries to perform elimination similarly to Listing 10 (lines 6–15). When the cell is in the `CANCELLED` state, the `resume(..)` operation either fails in the simple cancellation mode, or skips the cell with smart cancellation (lines 29–34). When the cell is in the `REFUSE` state, the user-specified `completeRefusedResume(..)` function is called and the operation finishes (lines 35–37).

When the cell contains a suspended thread, the algorithm first tries to resume it, finishing on success (lines 13–15). Otherwise, this thread is already aborted, and the logic depends on the cancellation mode. In simple cancellation, this `resume(..)` invocation fails (lines 17–18).

In contrast, with smart cancellation, the behaviour depends on whether the state changes to `CANCELLED` or `REFUSE`. Thus, in order not to leave the value in CQS when the *synchronous* resumption mode is used, we modify the algorithm and wait in a loop until the state changes (line 23, highlighted with yellow). Aside from this, with asynchronous resumption, the algorithm stays the same and delegates this resumption completion to the cancellation handler (lines 27–28).

The Modified Cell Life-Cycle Diagram. For readability, we present the full version of the cell life-cycle diagram in Figure 12, which supports both asynchronous and synchronous resumption as well as simple and smart cancellation modes.

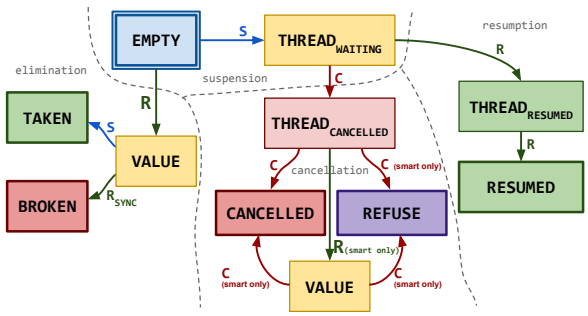

Figure 12: The full version of the cell life-cycle diagram supports both asynchronous and synchronous resumption and simple and smart cancellation modes.

```
1  fun resume(result: T): Bool {
2   i := FAA(&resumeIdx, +1)
3   while (true): // modify the cell
4    cur := cells[i]
5    when {
6     cur == null:
7       // Try to perform elimination
8       // similarly to Listing 10.
9       if CAS(&cells[i], null, result):
10        return true
11    cur is Thread:
12      // Try to resume the thread.
13      if cur.unpark(result):
14        cells[i] = RESUMED
15        return true
16      // The thread is cancelled.
17      if cancellationMode == SIMPLE:
18        return false
19      // Smart cancellation is used.
20      // With SYNC resumption, wait
21      // until the cell state changes
22      // to either CANCELLED or REFUSE
23      if resumeMode == SYNC: continue
24      // Delegate this resume(..)
25      // completion to the cancel-
26      // lation handler in ASYNC mode.
27      if CAS(&cells[i], cur, result):
28        return true
29    cur == CANCELLED:
30      // Fail with simple cancellation
31      if cancellationMode == SIMPLE:
32        return false
33      // Skip the cell in SMART mode.
34      return resume(result)
35    cur == REFUSE:
36      completeRefusedResume(result)
37      return true
38 }
```

Listing 12: Pseudo-code for `resume(..)` that supports all cancellation modes and both *asynchronous* and *synchronous* resumption. The key change from the algorithm with cancellation support in Listing 6 in Section 3 and Listing 10 in Appendix A.1 is highlighted with yellow.

B Infinite Array Implementation

The `CancellableQueueSynchronizer` framework is built on an infinite array, the cells of which are processed in sequential order. To emulate this infinite array, we follow the approach behind the implementation of the channels in Kotlin [25], maintaining a linked list of cell segments, each containing a fixed number of cells, as illustrated in Figure 13 (we repeat the illustration of the structure from Section 2).

Each segment has a unique id and can be seen as a node in a Michael-Scott queue [26]. Following this structure, we maintain only those cells that are in the currently active range (between `resumeIdx` and `suspendIdx`) and access them similarly to an array. Specifically, we change the current working segment once every `SEGM_SIZE` operations, where `SEGM_SIZE` is the number of cells in each segment.

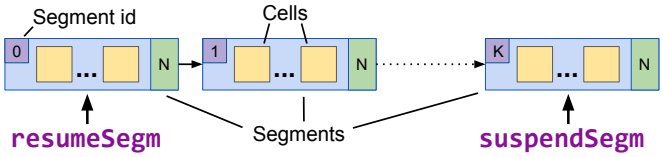

Figure 13: An infinite array as a linked list of cell segments.

Despite conceptual simplicity, the implementation of this structure is non-trivial, as shown in [25]. In this section, we discuss the implementation details and the required changes to the CQS algorithm, providing the formal proofs in Appendix E. We highlight that the infinite array implementation is not a part of our contribution, but providing all the technical details is necessary to make this paper self-contained and deliver formal proofs.

B.1 Basics Algorithm without Cancellation

Listing 13 presents a pseudo-code for `suspend()` in terms of these segments; the changes to `resume(..)` are symmetrical, so they are omitted. Instead of maintaining head and tail pointers, as is usually done in concurrent queues [26], we maintain `suspendSegm` and `resumeSegm` pointers related to the last segment used by `suspend()` and `resume(..)`, respectively; initially, they reference the same segment of `id = 0` (lines 1–6)

Initially, `suspend()` reads its last used segment (line 11) and increments `suspendIdx` (line 12). Then, it locates the required segment by following the chain of `next` pointers starting from the one already read before the increment, updating `suspendSegm` if required (line 17). The `findAndMoveForwardSusp(..)` implementation is straightforward — it finds the required segment first (lines 27–34), creating new segments if needed (lines 30– 33), and then updates `suspendSegm` to the one that was found if it has not been updated to it or one of the later segments yet (lines 36–41).

The changes for `resume(..)` are symmetric and, therefore, are omitted. The only notable difference is that `findAndMoveForwardResume(..)` should clean the `prev` pointer to the previous segment in the doubly-linked list structure to ensure that all the processed segments are available for garbage collection.

B.2 Segment Removal for Cancellation

Once a cell is in the `CANCELLED` state, it no longer stores the reference to the cancelled waiter, so the garbage collector is free to collect this reference. However, we also want it to collect the cell itself, in order to avoid memory leaks. Since we emulate the infinite array via a concurrent linked list of segments, the segments full of cancelled cells must be physically removed from the list. This way, we can guarantee that the memory complexity depends only on the number of non-cancelled cells, even in the case where all but one cell in each segment are cancelled, since the segment size is constant. The only exception is cells in the `REFUSE` state, as they are not considered as cancelled, so the corresponding segments cannot be removed. However, the `REFUSE` state indicates that there is a concurrent `resume(..)`, which is going to process this cell eventually — the number of such `resume(..)`-s is bounded by the number of threads. In sum, the total memory complexity should be $O(N + T)$, where N is the number of non-cancelled waiters, and T is the number of threads. (We assume the segment size to be constant; otherwise, the complexity should be multiplied by the segment size.)

Listing 14 presents the pseudocode of the segment removal part of the algorithm, including the required changes to the `resume(..)` and `findAndMoveForwardResume(..)` functions. The `suspend()` implementation stays almost the same as in Listing 13 above, with a single addition: a cancellation handler that invokes `s.onCancelledCell()` should be specified in the `park(..)` call. The changes to `findAndMoveForwardSuspend(..)` are symmetric to the ones for `findAndMoveForwardResume(..)`.

The Removal Algorithm Overview. In order to remove a segment in $O(1)$ (under no contention), we add a `prev` pointer to the `Segment` structure, which references the nearest non-removed segment on the left, or equals `null` if all of them are removed or processed (e.g., when the segment being removed is the head of the list). By maintaining the

```
1  var suspendSegm : Segment
2  var resumeSegm  : Segment
3  constructor {
4   s := Segment(id = 0)
5   suspendSegm = resumeSegm = s
6  }
7
8  fun suspend(): T {
9   // Read the current ‘suspendSegm‘
10  // before incrementing the counter.
11  s := suspendSegm
12  idx := FAA(&suspendIdx, 1)
13  // Find the required segment
14  // and update ‘suspendSegm‘.
15  id := idx / SEGM_SIZE
16  i := i % SEGM_SIZE
17  s = findAndMoveForwardSusp(s, id)
18  // Process the cell s[i].
19  t := currentThread()
20  if CAS(&s[i], null, t):
21    return park()
22  result := s[i]; s[i] = TAKEN
23  return result
24 }
```

```
25 fun findAndMoveForwardSusp(start: Segment,
26                           id: Long): Segment {
27  cur := start
28  // 1. Find the required segment.
29  while cur.id < id:
30    if cur.next == null:
31      // Create a new segment if needed.
32      s := Segment(id = cur.id+1)
33      CAS(&cur.next, null, s)
34    cur = cur.next
35  // 2. Move ‘suspendSegm‘ forward if needed
36  while (true):
37    s := suspendSegm
38    // Already up-to-date?
39    if s.id ≥ id: break
40    // Try to update
41    if CAS(&suspendSegm, s, cur): break
42  // 3. Return the found segment.
43  return cur
44 }
```

Listing 13: Implementation of the suspend operation that manipulates a linked list of fixed-size segments. The key change is highlighted — the current suspendSegm should be read before the suspendIdx increment, which guarantees that the required segment can be found by following next pointers and all the preceding segments are not needed anymore, so suspendSegm can be safely moved forward. For simplicity, this implementation supports only the asynchronous (default) resumption mode.

prev pointer, we can perform physical removal by linking the previous and the next segments to each other. However, doing so correctly requires non-trivial tricks in a concurrent environment.

Once a segment is removed, we should guarantee that it is no longer reachable by the next and prev references starting from suspendSegm and resumeSegm. For this purpose, we split the removal procedure into two parts: *logical* and *physical*. We assume that the segment is logically removed if all the cells are in the CANCELLED state and neither suspendSegm nor resumeSegm references it (lines 63–66). At the same time, we need to guarantee that they cannot start referencing logically removed segments, making them “alive” again and causing memory leaks.

To solve this problem, we maintain the number of cancelled cells alongside the number of pointers that reference this segment in a single integer field (line 61). By storing these numbers in a single register, we are able to modify them atomically — to emphasize this, the corresponding code is wrapped with atomic block in the pseudocode. Thus, there are two ways for a segment to become cancelled. First, if neither suspendSegm nor resumeSegm references it, and the cancelled counter reaches SEGM_SIZE, the segment becomes logically removed and the following remove() call should remove it physically. In the second case, all the cells are already cancelled and the number of pointers reaches zero when suspendSegm or resumeSegm updates. In this case, the corresponding code must check whether the previously referenced segment became logically removed and invoke remove() if needed.

Correspondingly, when suspendSegm or resumeSegm need to be updated, they should increment the number of pointers that reference the new segment. However, if this new segment is already logically removed, the increment fails and the update should be restarted. The corresponding logic is provided in the tryIncPointers() function (lines 72–75). Similarly, when suspendSegm or resumeSegm stop referencing some segment, they decrement the number of pointers. The corresponding decPointers() function returns true if the segments becomes logically removed (lines 79–81).

The only exception from this is the attempt to remove the tail segment. We forbid removing the tail segment, as doing so would make it more difficult to ensure that each segment has a unique id throughout the list’s lifetime. Therefore, we ignore the attempts to remove the tail segment — see the first statement in remove() (line 86). Thus, if a segment was the tail of the list at the time of logical removal, the following remove() call does nothing, and the physical removal of this segment is postponed until it stops being the tail.

The resume(..) Operation. While the logic of suspend() stays the same, since the segment requested by it has non-CANCELLED cells and is not removed at that point, the resume(..) operation requires some modifications (lines 1–15). First, the semantics of findAndMoveForward(..) are slightly changed. Since the requested segment

```
fun resume(value: T): Bool = while(true) {
  r := resumeSegm
  i := FAA(&resumeIdx, 1)
  id := i / SEGM_SIZE
  s := findAndMoveForwardResume(r, id)
  // All the previous segments are processed.
  if s.id != id:
    if cancellationMode == SIMPLE: return false
    // Update resumeIdx to skip the sequence of
    // cancelled segments in smart cancellation
    CAS(&resumeIdx, i + 1, s.id * SEGM_SIZE)
    continue
  // Process the cell s[i % SEGM_SIZE]
  ...
}
// Returns the first non-removed segment with
// id equal to or greater than the specified,
// creating new segments if needed.
fun findAndMoveForwardResume(start: Segment,
                             id: Long): Segment {
  while (true):
    s := findSegm(start, id)
    // Try to update ‘resumeSegm‘, and
    // restart if the found segment is
    // removed and is not the tail one.
    if moveForwardResume(s): break
    s.prev = null; return s
}

fun findSegm(start: Segment, id: Long): Segment {
  cur := start
  while cur.id < id || cur.removed():
    if cur.next == null:
      // Create a new segment if needed.
      newSegm := Segment(id = cur.id + 1)
      if CAS(&cur.next, null, newSegm):
        // Is the previous tail removed?
        if cur.removed(): cur.remove()
    cur = cur.next
  return cur
}
fun moveForwardResume(to: Segment): Bool {
  while (true):
    cur := resumeSegm
    // Do we still need to update ‘resumeSegm‘?
    if cur.id >= to.id: return true
    // Try to inc pointers to ‘to‘.
    if !to.tryIncPointers(): return false
    // Try to update ‘resumeSegm‘.
    if CAS(&resumeSegm, cur, to):
      // Dec pointers to cur.
      if cur.decPointers(): cur.remove()
      return true
    // The ‘resumeSegm‘ update has failed,
    //  dec pointers to ‘to‘ back.
    if to.decPointers(): to.remove()
}
```

```
class Segment {
 // Initialized with (2, 0) for the first
 // segment; stored in a single 32-bit Int.
 var (pointers, cancelled) = (0, 0)

 fun removed(): Bool = atomic {
   return cancelled == SEGM_SIZE &&
          pointers = 0
 }
 fun onCancelledCell() = atomic {
   cancelled++; if removed(): remove()
 }
 // Increments the number of pointers;
 // Fails if the segment is removed.
 fun tryIncPointers(): Bool = atomic {
   if removed():  return false
   pointers++; return true
 }
 // Decrements the number of pointers and
 // returns ‘true‘ if the segment becomes
 // logically removed, ‘false‘ otherwise.
 fun decPointers(): Bool = atomic {
   pointers--; return removed()
 }

 // Physically removes the current segment
 fun remove() = while(true) {
   // The tail segment cannot be removed.
   if next == null: return
   // Find the closest alive segments
   // on the left and on the right.
   prev := aliveSegmRight()
   next := aliveSegmLeft()
   // Link ‘next‘ and ‘prev‘.
   next.prev = prev
   if prev != null: prev.next = next
   // Are ‘prev‘ and ‘next‘ still alive?
   if next.removed() && next.next != null:
     continue
   if prev != null && prev.removed():
     continue
   return // this segment is removed.
 }

 fun aliveSegmLeft(): Segment {
   cur := prev
   while cur != null && cur.removed():
     cur = cur.prev
   return cur // ‘null‘ if all are removed
 }
 fun aliveSegmRight(): Segment {
   cur := next
   while cur.removed() && cur.next != null:
     cur = cur.next
   return cur // tail if all are removed
 }
}
```

Listing 14: Pseudocode for the segment removal algorithm. On the left, the modified `resume(..)` is presented. In addition, `findAndModeForwardResume(..)` is split into two parts, the important changes of which are highlighted with yellow. The required changes to `Segment` are shown on the right. The `onCancelledCell()` function, highlighted with green, is called when a cell from this segment moves to `CANCELLED` state.

can be removed, it now returns the first *non-removed* segment with id equal to or greater than the requested one, creating new segments if needed. Thus, if the requested segment is not found (line 7), we know that the required cell is in the CANCELLED state, and so can process it correspondingly: fail in the simple cancellation mode (line 8) or efficiently skip a sequence of removed segments and restart the operation in the smart cancellation mode (lines 11–12). Otherwise, when the requested segment is successfully found, we process it as usual; see the Listing 6 for details. One more notable change is that we must clean the `prev` pointer in `findAndMoveForwardResume(..)` to avoid memory leaks (line 27) — all the previous segments are either processed or going to be processed by concurrent `resume(..)`-s.

The `findAndMoveForwardResume(..)` Operation. We split the operation into two parts. First, the `findSegm(..)` function finds the first non-removed segment with id equal to or greater than the requested one, creating new segments if needed (line 22). Once the segment is found, we try to make `resumeSegm` point to it — this part can fail if the found segment becomes logically removed in the meantime, and the procedure restarts in this case (line 26).

Essentially, the `findSegm(..)` logic stays the same with two small modifications. First, it skips the logically removed segments in the search procedure (the highlighted query at line 32). Second, once the tail of the list is updated, it checks whether the old tail should be removed (line 38).

As for the `moveForwardResume(..)` operation, we need to increment and decrement the numbers of pointers there. Thus, we first try to increment the number of pointers to the new segment (line 48), returning `false` and causing `findAndMoveForwardResume(..)` to restart on failure. If the increment of the number of pointers succeeds, the operation tries to update `resumeSegm` to the new one (line 50). If the update succeeds, the number of pointers to the old segment (`cur` in the code) should be decremented, removing the segment physically if needed (line 52). If the `resumeSegm` update fails, the operations decrements the number of pointers to the new segment back (removing it if needed) and restarts.

The `Segment.onCancelledCell()` Operation. The `onCancelledCell()` operation is called when the cell moves to the CANCELLED state, see Listing 6. It increments the number of cancelled cells and checks if this led to the segment becoming logically removed, in which case it invokes `remove()` (lines 67–69).

The `Segment.remove()` Operation. The last part is the `remove()` operation itself. If the segment that is being removed is the tail, the removal is postponed and delegated to either `findSegmentResume(..)` or `findSegmentSuspend(..)` that will update the tail and check whether the old one should be removed (line 86).

Otherwise, the algorithm finds the first non-removed segment to the right (line 89) by following `next` pointers, and the first non-removed segment on the left (line 90) by following `prev` pointers. After that, we link the segment on the right with the segment on the left by updating its `prev` pointer (line 92). If a non-removed segment on the left was not found, `prev` is updated to `null`. Otherwise, we link such a segment with the segment on the right by updating the `next` pointer (line 93). If all segments on the right are logically removed, we manipulate the tail one.

As a result, we successfully linked the segments found on the left and on the right with each other. However, they could have been removed in meantime. Therefore, we check if they became removed and re-start the removal if they did (lines 95–98). Otherwise, the removal procedure is completed. It is worth noting that it is possible that concurrent `remove()`-s keep the reference to our removed segment and can accidentally re-link it with some other segment(s). However, due to checks that the segments found on the left and on the right are non-removed after the linking procedure (lines 95–98), we can guarantee that even if such an accident occurs, the `remove()` that led to this error will fix the problem. Thus, we know that the segment will be removed eventually.

C Semaphore and Blocking Pools: Implementation Details

In this section, we describe the semaphore and blocking pools algorithms in detail. Additionally, for semaphore, we also cover the version with *synchronous* resumption mode, which is briefly introduced in Section 2 and discussed in Appendix A. In essence, the synchronous resumption mode is needed to support non-blocking variants of blocking operations, such as `Mutex.tryLock()` and `Semaphore.tryAcquire()`. Please see Appendix A for details.

C.1 Semaphore

The semaphore algorithm is similar to the mutex one discussed during the CQS details presentation; Listing 3 presents the basic mutex version without cancellation support, while Listing 5 fills the gap presenting the cancellation handler for the *smart* cancellation mode. When implementing semaphore, the only significant difference is that instead of a single "unlocked" state, the `state` counter stores the number of available permits. Listing 15 shows the corresponding pseudocode.

In the presented implementation, we use the asynchronous resumption (line 2) and the smart cancellation (line 3) modes; other variants are omitted and can be easily constructed based on the one we present. The `state` field (line 6) stores either the number of permits when positive or 0, or the number of waiters when negative. Similarly to the mutex algorithm, `acquire()` decrements this counter and suspends if needed, while `release()` increments it and resumes the first waiter if there is one.

The `acquire()` Operation. First, the `state` counter is decremented (line 9). If the counter was positive — thus, the number of available permits was positive, — the operation has successfully taken a permit and completes immediately (line 11). Otherwise, no permits are available, so it suspends in the `CancellableQueueSynchronizer` (line 13).

The `release()` Operation. First, in decrements the `state` counter (line 16). If the counter was non-negative, then no operation is waiting in the CQS, so the permit is successfully returned to the semaphore and the operation completes. Otherwise, the next waiter should be resumed (line 18).

Cancellation. When a waiting `acquire()` is cancelled, it increments the `state` counter (line 21). If the counter was negative, this increment successfully decremented the number of waiters, so the cancellation succeeds and `onCancellation()` returns `true`. Otherwise, if the counter was non-negative, there already is a concurrent `release()` that will resume this waiter eventually. In this case, the corresponding `resume()` should be refused, since and the permit is already returned back, the `completeRefusedResume(..)` operation does nothing (lines 29–32).

```
1  val cqs = CQS<Unit>(
2    resumptionMode = ASYNC,
3    cancellationMode = SMART
4  )
5  // Initialized with the number of permits
6  var state: Int = K
7
8  fun acquire() {
9    s := FAA(&state, -1)
10   // Is the permit acquired?
11   if s > 0: return
12   // Suspend otherwise
13   return cqs.suspend()
14 }
15 fun release() {
16   s := FAA(&state, 1)
17   // Is there a waiter to be resumed?
18   if s < 0: cqs.resume(Unit)
19 }
20 fun onCancellation(): Bool {
21   s := FAA(&state, 1)
22   // If the number of waiters was
23   // decremented, the cancellation
24   // successfully completes. Otherwise,
25   // there is a release() that is going
26   // to resume this waiter, refuse it.
27   return s < 0
28 }
29 fun completeRefusedResume(permit: Unit){
30   // The permit is already been returned
31   // to this semaphore; do nothing.
32 }
```

Listing 15: Semaphore implementation on the top of CQS with asynchronous resumption mode and smart cancellation. The initial number of permits is K.

Synchronous Resumption and `tryAcquire()`. In order to use the synchronous resumption mode, both `acquire()` and `release()` should restart when the `suspend()` and `resume(..)` invocations fail. Thus, it becomes possible to implement an additional `tryAcquire()` operation that attempts to take a permit if it is available — it simply decrements the `state` counter if it is positive or fails otherwise.

C.2 Blocking Pools

While the barrier, count-down-latch, and semaphore algorithms use `CancellableQueueSynchronizer` only for synchronization, it is also possible to develop *communication* primitives on top of it. In this section, we consider simple blocking pool implementations.

When working with expensive resources such as database connections, sockets, etc., it is common to reuse them, which usually requires an efficient and accessible mechanism. The *blocking pool* abstraction maintains a set of elements that can be retrieved to process some operation, after which the element is placed back in the pool. Operations `put(element)` and `take()` are provided:

- `put(element)` either resumes the first waiting `take()` operation and passes the element to it, or puts the element into the pool;
- `take()` takes one of the elements from the pool (in an unspecified order), or suspends if it is empty: later `put(e)` operations resume waiting `take()`-s in the first-in-first-out order.

In this paper, we consider two pool implementations: queue-based and stack-based. Intuitively, the queue-based implementation is faster since it can be built on segments, similarly to the `CancellableQueueSynchronizer`, and uses `Fetch-And-Add`-s on the contended path [14, 15]. In contrast, the stack-based pool retrieves the last inserted, thus the "hottest", element. Please note that both algorithms presented in this section are not linearizable and can retrieve elements out-of-order under some races. However, since pools do not guarantee that the stored elements are ordered, these queue and stack-based versions should be considered as bags with specific heuristics; these semantics matches practical applications.

We start with an abstract solution that does not rely on queues, stacks, or other containers. After that, we provide solutions for queue-based and stack-based pools on top of this abstract construct.

Abstract Blocking Pool. Intuitively, the blocking pool contract reminds of a semaphore. It, like the semaphore, transfers resources, with the only difference that semaphore transfers logical non-distinguishable permits while blocking pool works with actual elements. The rest, however, is almost the same. Listing 16 presents the abstract blocking pool implementation on top of `CancellableQueueSynchronizer` with asynchronous resumption and smart cancellation (lines 2 and 3). Like in the semaphore, the algorithm maintains the `size` counter (line 5, in the semaphore this counter is called `state`), which represents the number of elements in the pool if it is non-negative, and the negated number of suspended `take()` requests otherwise.

The `put(..)` operation increments `size` first (line 8), and either resumes the next waiter if the counter was negative (line 11) or adds the element to the pool structure via `tryInsert(..)` function if there was no waiter in the pool (line 15). In our design, `tryInsert(..)` can fail if a concurrent `take()` comes between the counter increment and the `tryInsert(..)` call — both operations should restart in this case.

```
1  val cqs = CQS<E>(
2    resumeMode = ASYNC,
3    cancellationMode = SMART
4  )
5  var size: Int = 0
6
7  fun put(element: E) = while (true) {
8   s := FAA(&size, +1)
9   if s < 0: // is there a waiting take()?
10    // Resume the first waiter and complete.
11    cqs.resume(element); return
12  else:
13    // Try to insert the element. Can fail due
14    // to a race with a concurrent retrieve()
15    if tryInsert(element): return
16 }
17 fun take(): E = while(true) {
18  s := FAA(&size, -1)
19  if s > 0:
20    // Try to retrieve an element. Can fail
21    // due to a race with a concurrent  put(e
       )
22    e := tryRetrieve()
23    if e != null: return e
24  else:
25    return cqs.suspend() // no elements
26 }
27 fun onCancellation(): Bool {
28  // Similar to the semaphore algorithm.
29  s := FAA(&size, 1)
30  return s < 0
31 }
32 fun completeRefusedResume(element: E) {
33  if !tryInsert(e): put(e)
34 }
35 // When tryInsert(e) fails, the conjunctive
36 // tryRetrieve() fails as well, and vice versa
37 fun tryInsert(element: E): Bool
38 fun tryRetrieve(): E?
```

Listing 16: Abstract blocking pool implementation that maintains the `size` counter of available elements (if positive) or waiting retrievals (if negative). All the waiting `take()` operations are processed in first-in-first-out order, while the pool itself can use any concurrent data structure under the hood. In Listing 17, we present two solutions: based on a concurrent stack, which returns the "hottest" element, and based on a queue, which is relatively more efficient.

The take() operation decrements the size counter first (line 18), and either tries to retrieve an element from the pool structure via tryRetrieve(..) if the size was positive (lines 22–23) or suspends in the CQS if there is no element to retrieve (line 25).

As for the cancellation logic, the onCancellation() implementation is identical to the one in semaphore: it increments the size counter (line 29) and returns true if the number of waiters was decremented (so the counter was negative) or returns false if there is an upcoming resume(..) that should be refused. To complete the refused resume, the algorithm tries to insert the element back into the pool structure via tryInstert(..) and on its failure performs a full put(..) — see the completeRefusedResume(..) implementation (lines 32–34)

Queue-Based Pool. In order to complete the pool implementation, we need to specify the tryInsert(..) and tryRetrieve() functions. The pool with a queue under the hood is shown on the left side of Listing 17. Our implementation is based on an infinite array (line 3), which can be emulated in a way similar to how it is done in the CancellableQueueSynchronizer framework.

The tryInsertQueue(..) (we added Queue and Stack suffixes to distinguish the implementations) operation increments its insertIdx counter (line 9) and tries to atomically change the corresponding slot in the infinite array from null to the given element via CAS (line 14). If this CAS fails, it means that a concurrent tryRetrieveQueue(), which already discovered the preceding size increment, came to the same array slot and broke it (line 22) — tryInsertQueue(..) returns false in this case.

The tryRetrieveQueue() operation increments its retrieveIdx counter (line 18) and tries to retrieve an element from the corresponding infinite array slot (line 22). If the slot is empty, it breaks it by atomically replacing null with the BROKEN token and returns false, causing the paired tryInsertQueue(..) to fail as well (line 14).

Stack-Based Pool. The pool with a classic Treiber stack under the hood is presented on the right side of Listing 17. Here we face a similar race when put(..), which has already incremented the size counter but has not inserted the element yet, interferes with a concurrent take() that tries to retrieve an element. We use an approach similar to breaking slots in the previously discussed queue-based pool. The difference is that, instead of breaking slots, tryRetrieveStack() inserts a "failed node" if the stack is empty or contains other failed nodes (lines 45–49); otherwise, it removes the top node with an element (lines 53–54). On the opposite side, tryInsertStack() checks that the stack does not have these "failed nodes", removing one and failing if they exist (lines 32–34); otherwise, it inserts a node with the specified element (line 38).

D Progress Guarantees

Here, we discuss the progress guarantees of both CQS suspend(..) and resume(..) operations and the primitives from Section 4 built on top of the CancellableQueueSynchronizer framework. Notably, we consider both *asynchronous* and *synchronous* resumption modes, where the first is the default one described on the main body, and the synchronous resumption aims at supporting non-blocking operations, such as Mutex.tryLock() or Semaphore.tryAcquire() — it is briefly introduced in Section 2 and discussed in full detail in Appendix A.

Similarly to the dual data structures formalism [33], we reason about progress independently of whether the operation was suspended. Thus, when we say that some blocking operation is lock- or wait-free, we mean that it performs all the synchronization with this progress guarantee, either completing immediately or adding itself to the queue of waiters followed by suspension. Specifically, we analyze the part of the operation prior to Thread.park(..) call, if one ever occurs.

D.1 The CQS Operations

First, we discuss the suspend() and resume(..) of the CancellableQueueSynchronizer framework itself, followed by the analysis of the barrier, the count-down-latch, the semaphore, and the blocking pools presented in Section 4 and Appendix C.

The suspend() Operation. The suspend() operation obtains the id of the working cell by incrementing suspendIdx. It then finds the required segment in a bounded number of steps and either installs the currently running thread to the cell or returns the value already stored in it, failing if the cell is already broken by a concurrent resume(). In either case, it completes within a finite number of its own steps, and is, therefore, *wait-free*.

```
// The queue bases on an infinite
// array with two counters.
val a = InfiniteArray()
var insertIdx: Long = 0
var retrtieveIdx: Long = 0

fun tryInsertQueue(element: E): Bool
    {
  // Get an index for this insertion
  i := FAA(&insertIdx, 1)
  // Try to put the element into the
  // slot, failing if it has already
  // been broken by a paired
  // retrieval that came earlier.
  return CAS(&a[i], null, element)
}
fun tryRetrieveQueue(): E? {
  // Get an index for this retrieval
  i := FAA(&retrtieveIdx, 1)
  // Replace the slot value with ⊥.
  // If null, the slot becomes broken
  // and this retrieval attempt fails
      .
  return GetAndSet(&a[i], BROKEN)
}
```

```
class Node(val element: E, next: Node)
var top: Node? = null

fun tryInsertStack(element: E): Bool {
 while (true) {
   t := top
   // Does this stack contain
   // failed retrievals?
   if t != null && t.element == BROKEN:
     // Remove the failed node and fail.
     if CAS(&top, t, t.next): return false
   else:
     // The stack is either empty
     // or contains elements.
     if CAS(&top, t, Node(element, t)):
      return true
 }
}
fun tryRetrieveStack(): E? {
 while (true) {
   t := top
   // Does the stack have elements?
   if t == null || t.element == BROKEN:
     // The stack is either empty or
     // contains failed retrievals;
     // add one more and fail
     if CAS(&top, t, Node(BROKEN, t): return
       null
   else:
     // The stack contains elements;
     // try to remove the top one.
     if CAS(&top, t, t.next):
       return t.element
 }
}
```

Listing 17: Blocking pool specializations built on top of the solution in Listing 16 with a queue (on the left) and stack (on the right) under the hood. Intuitively, the queue-based implementation is faster since it is based on arrays and uses `Fetch-And-Add`-s on the contended path, while the stack-based pool retrieves the last inserted, thus the "hottest", element.

The `resume(..)` Operation. The behaviour of `resume(..)` depends on the cancellation mode. If no cancellation happened during the execution, `resume(..)` obtains an id of the working cell by incrementing `resumeIdx`, finds the required segment in a bounded number of steps, and either places the element in the cell (optionally waiting in a bounded loop in the synchronous resumption mode) or resumes the stored waiter. In either case, `resume(..)` is wait-free.

With simple cancellation, `Thread.cancel()` moves the cell state to `CANCELLED`, and the `resume(..)` that processes this cell fails. Therefore, `resume(..)` remains *wait-free*.

The situation is more complex in the smart cancellation mode. In this case, the progress guarantee of `resume(..)` depends on the resumption mode. In the synchronous resumption mode, `resume(..)` may wait in a spin-loop until the cell's state changes from $\texttt{THREAD}_{\texttt{CANCELLED}}$ to `CANCELLED` or `REFUSE`. Thus, `resume(..)` is *blocking*. In the asynchronous mode, `resume(..)` is *lock-free* due to a possibly infinite number of `suspend()`-s that place and immediately abort. However, the progress guarantee can degrade if the `completeRefusedResume(..)` implementation, which is specified by the user and invoked when a `resume(..)` detects that it was refused, ensures a weaker progress guarantee.

The `Thread.cancel(..)` Operation. With simple cancellation, `Thread.cancel()` moves the cell state to `CANCELLED` and potentially removes the segment if the last cell was cancelled. The segment removing procedure is lock-free, so cancellation obeys lock-freedom as well.

With smart cancellation, the handler invokes the `onCancellation()` function and can also invoke the `completeRefusedResume(..)` procedure — both of them are specified by the user. In addition, the handler can call `resume(..)`

in the asynchronous (default) resumption mode. The `resume(..)` operation is at best lock-free, so the overall cancellation is lock-free if the functions specified by the user guarantee lock-freedom as well, and is bounded by their progress guarantees otherwise.

D.2 Barrier

Since our implementation does not support cancellation and the asynchronous resumption mode is used, it is guaranteed that both `suspend()` and `resume(..)` synchronizations are wait-free. The rest of the `arrive()` operation is also wait-free, which should be obvious from the code. Therefore, our implementation guarantees *wait-freedom*.

D.3 Count-Down-Latch

Since `suspend()` is wait-free and does not fail, the `await()` operation is obviously *wait-free* as well. The cancellation, however, is *lock-free* due to possible segment removing.

As for the `countDown()` operation, it performs `Fetch-And-Add` at line 10 and invokes `resumeWaiters()` if the count has reached zero at line 12. Thus, the progress guarantee for `countDown()` is completely dependent on the `resumeWaiters()` function. Surprisingly, even with the infinite loop wrapper, the number of failed `CAS`-s to set the `DONE_BIT` at line 28 is bounded by the number of concurrent `await()` invocations, and thus, by the parallelism level in general. If new `await()` invocations happen when `resumeWaiters()` is invoked, since the `count` is already zero, they complete immediately and neither change the `waiters` field nor suspend. As a consequence, `resume(..)` can skip a bounded number of cancelled cells and is wait-free. In sum, `resumeWaiters()` along with the `countDown()` operation are *wait-free*.

D.4 Semaphore

Consider the case where no cancellation happens during the execution. In this case, both `suspend()` and `resume(..)` are *wait-free*, so `acquire()` and `release()` are also wait-free. However, when synchronous resumption is used, concurrent `suspend()` and `resume(..)` can lead to failing each other. Therefore, the operations may restart and interfere infinitely with synchronous resumption, so only *obstruction-freedom* is guaranteed.

Cancellation weakens the progress guarantees. With asynchronous resumption, `resume(..)` is only *lock-free* since there can be an infinite sequence of `suspend()`-s followed by successful cancellations, so any given `resume(..)` may not finish while the system makes progress. Since the cancellation handler can invoke `resume(..)`, it is, therefore, also *lock-free*. With the synchronous resumption, the `resume(..)` operation is *blocking*, while the cancellation part is *lock-free* due to a possible segment removal.

D.5 Blocking Pools

The queue-based pool provides wait-free `tryInsert(e)` and `tryRetrieve()` functions, while in the stack-based version, they ensure lock-freedom. However, additions and removals can interfere in an obstruction-free way due to the slot breaking in the queue-based version and publishing "failed nodes" in the stack-based one. Nonetheless, they always complete in a bounded number of steps when all other threads are paused. Therefore, all the operations, including the cancellation that can invoke `put(..)` as a part of `completeRefusedResume(..)`, are *obstruction-free*.

E Formal Specification and Proofs for CQS

This section outlines the formal proofs for the Coq formalization of `CancellableQueueSynchronizer`; the specifications and proofs of the presented algorithms on top of CQS are discussed in Section F. **The proofs themselves are available on GitHub [27]**. The corresponding files are referenced throughout.

Providing formal proofs of correctness for concurrent data structures is currently rare, and even more so for algorithms and data structures employed in a realistic production setting. (This is in spite of Iris certainly being powerful enough to express such proofs.) Notable exceptions include the verification of a concurrent queue used in the Dartino framework [18], proofs for algorithms used in real-world databases and filesystems [19], the contextual refinement of a concurrent queue similar to one in the Java standard library [20], and the recent cases of the Meta company verifying several of its internally-used concurrent data structures [29, 28]. We also highlight the proofs for a wide range of libraries used throughout the Rust ecosystem [21].

We suspect that the main reason for the dearth of such proofs is the high complexity barrier, preventing users from using separation logic to encode the intuition behind the data structure design. Also, in our experience, obtaining formal proofs for complex data structures is not obvious: in total, the proofs of the claims of this paper span more than 10'000 lines of Coq code, much of which required non-trivial reasoning.

We found the formalization process quite useful, as it identified subtle correctness issues in our implementation, especially in the case of the cancellation operation. We note that the proofs below do not attempt to show the FIFO property: proving such properties is known to be very challenging in our framework, and can be approached via classical proofs.

Reader Guide. This section outlines the basic ideas, definitions, and rationale behind our proofs in Coq, and functions essentially as "liner notes" for the formal proof. The experienced reader may wish to directly examine the proof text, perhaps in conjunction with Section E.6. Although we strive to justify our definitions and choices, we understand that some readers may find it difficult to internalize the fine details in this section. This is due to the fact that proofs of such massive algorithms as CQS are typically hard to follow and understand. That is the reason why we decided to prove the framework in Coq, which guarantees correctness of our proofs: manual proofs would provide too big of a surface for error for our liking.

E.1 Structure of the Proofs

Resources and Invariants. The discussion here is a high-level description of the notions on which Iris operates. Its purpose is to provide the reader with just enough intuition to be able to follow the outline provided in this paper. A more detailed and technical discussion can be found in the description of Iris itself [17].

There are two basic notions at the heart of the proofs: *resources* and *invariants*.

A *resource* is an entity that only exists in the logical realm, does not affect the code execution in any way, and is used to keep track of our knowledge about the state of the system. An example of a resource is an exclusive right to write to a particular memory location. Each executing thread keeps a collection of resources that it can use to perform various operations.

An *invariant* is a collection of resources that is always owned by a data structure itself, as opposed to some particular thread. This notion is not to be confused with a loop invariant, which is a broadly similar, but meaningfully distinct concept. The resources stored in an invariant can be used by any thread at any time as long as no thread can ever observe the invariant not holding: in particular, it is allowed for a thread to borrow the resources from an invariant for the duration of an atomic operation, but not for longer. (Note that if the proofs were performed in a weak memory model, this notion would have to be significantly more elaborate.)

Resources can be allocated; some resources can be deallocated; some can be duplicated, split into fractions, combined to form other resources, etc. In this outline of the proof, the inner workings of these operations are omitted due to the sheer scope of the formal proof and the number of resources that needed to be defined; instead, we postulate where needed the existence of resources with the required properties or even imply it. We feel justified to focus on the general picture due to the fact that Coq has performed an automatic verification of the validity of our claims. For example, we often say that a particular data structure "knows" that there only exists a fixed number of copies of a particular resource; such knowledge is itself represented as a resource that is stored in an invariant associated with that structure, but presenting the proofs in accordance with this would, in our view, obscure the general view in favor of minutiae.

Specifications of Methods. The proof of each method is provided in the form of specification of how its behavior affects the available resources. Specifications have the following form: "If an expression e is executed by a thread that owns A, then the call does not break any invariants and, when it completes, it returns a value v and provides the calling thread with B", where A and B are (groups of) resources and v can appear in the definition of B.

The specification can be parameterized with some values (usually the arguments to the method), which can appear in definitions of e, A, and B.

As an example of a specification, we consider `GetAndSet`, also commonly known as `swap()`, which always successfully writes v to a memory location ℓ and returns the value that was stored there at the moment of the write, can be given as follows: "If $\texttt{GetAndSet}(v)$ is executed by a thread that owns the exclusive knowledge about the memory location ℓ containing x, then the call returns x and produces the exclusive knowledge about ℓ containing v". The "exclusivity" here is mentioned because if some other parts of the system knew that ℓ contained x, the method could not be correctly executed, as it would violate the knowledge owned by the other parties.

There are some weaknesses to this form of specification: if a method never finishes and instead hangs without breaking any invariants, then the specification is still correct. In fact, a simple infinite loop that does not access any state satisfies any specification. This is an important reason for why we address the progress guarantees separately from correctness proofs.

Specifications of Logically Atomic Methods. An additional special case is that of methods that need to perform their operations atomically in order to be correct. For example, consider the specification of the `GetAndSet` operation given above. That specification is highly impractical, as `GetAndSet` usually operates on shared state, so it is not possible to provide only one thread with the exclusive knowledge of the contents of a memory cell for the whole duration of `GetAndSet`: an attempt by any other thread to access the cell in the meantime would be invalid, as only one thread has any knowledge about that memory.

To deal with this issue, a separate form of specifications exists: "If e is executed and has access to A, then at some point in time it atomically consumes A and provides B; after it finishes, it returns a value v", where v can appear in the definition of B and A can be parameterized by some values that can also appear in the definition of B. "Having access" here means obtaining A (possibly several times) and immediately providing it back.

It is possible then to provide a useful specification of `GetAndSet`: "If `GetAndSet`(v) is executed and has access to knowledge that ℓ contains some value (we call the value it has at this moment in time x), then at some point it replaces this knowledge with the fact that ℓ contains v; after the operation finishes, it returns x". To simplify the nomenclature, we instead say that `GetAndSet`(v) atomically replaces the value in ℓ with v and returns the initial value; this section is meant to define what specifically we mean by `GetAndSet`(v) being atomic even though, as usually defined, it can access a memory location several times, which is atomic only logically and not physically.

Specifications of Logical Operations. In addition to specifications of code, we recognize some operations that do not require any code to execute and only operate on resources and invariants. For example, there could exist an invariant that owned an instance of either an exclusive (that is, one-of-a-kind) resource C or a D; then a thread that owns the C could obtain an instance of D: given that C is exclusive and the thread owns it, the invariant must be holding D, so the thread can then swap the D for its C without violating the invariant, and, most importantly for the point raised here, without executing any code.

E.2 Futures

Throughout the paper, we used the notion of threads that allow parking, unparking, and canceling them. However, in HeapLang, the default language to describe computations in the Iris framework, threads are a very light concept that essentially just describes code running in parallel with arbitrary interleavings. It doesn't have a notion of parking, unparking, or cancellation.

Therefore, in order to perform formal proofs of operations that support cancellation, we need to introduce some model of blocking computations that is general enough to be adapted to any practical language or library, independently of whether they are built on threads or coroutines, while at the same time supporting all operations on threads that we used.

The model we chose is that of `Futures`, a structure that allows passing a unique value to it, checking whether the value is present, or canceling the computation that would provide the value. The following is a description of the model, including its pseudocode.

Example: the Mutex. Consider the `lock()` operation in mutex. Intuitively, it either takes the lock immediately or registers as a waiter and then is resumed by an `unlock()` operation. We can split `lock()` into two phases at the point of suspension. This idea is inspired by the dual data structures formalism [33], originally designed for synchronous queues, where these two phases are named "registration" and "follow-up".

Unlike the dual data structures formalism, we make suspension *explicit* by returning a special `Future` instance as a result of a blocking operation. With this change, `lock()` in mutex returns `Future<Unit>`. See Listing 18 below.

```
interface Mutex {
 fun lock(): ~~Unit~~ Future<Unit> { ... }
 fun release() { ... }
}
```

Listing 18: Mutex API via `Future`-s.

```
1  interface Future<R> {
2   fun get(): R? or ⊥ // R - completed with R
3                      // null - not completed
4                      // ⊥ - cancelled
5   fun cancel(): Bool // true - cancelled
6                      // false - completed
7  }
8  // Use this Future without suspension.
9  class ImmediateResult<R>(
10  val result: R // the operation result
11 ) : Future<R> {
12  override fun get() = result
13  override fun cancel() = false
14 }
15 // Use this Future when suspending.
16 class Request<R>(
17  val cancellationHandler: () -> Unit
18 ) : Future<R> {
19  var result: R? or ⊥ = null // ⊥ => cancelled
20
21  fun complete(r: R): Bool = CAS(&result, null, r)
22
23  override fun get() = result
24  override fun cancel(): Bool {
25   if CAS(&result, null, ⊥): // mark as cancelled
26     cancellationHandler() // invoke the handler
27     return true // successfully cancelled
28   return false // already completed
29  }
30 }
```

Listing 19: Implementations of `Future` for both suspending and immediate completing situations.

The `lock()` operation completes regardless of whether the lock has been successfully acquired or the request was put into the waiting queue. If the lock has not yet been acquired, calling `get()` on this `Future` returns `null` instead, but after the lock is transferred to the waiting `lock()` operation, `get()` starts returning `Unit`, indicating that the blocking part of the `lock()` operation has completed with the result `Unit`.

Implementation of `Futures`. Since it is possible for a potentially blocking operation to complete immediately, we have two `Future` implementations presented in Listing 19: `ImmediateResult` is returned when the operation completes without suspension, while `Request` is returned when the operation suspends.

Though most synchronization primitives return `Unit` as a result of blocking request, there are plenty of data structures, such as blocking queues, where operations also manipulate some data. Thus, we make our `Future` generic in type parameter `R` (line 1). In addition, we provide a way to cancel the waiting request via the `cancel()` operation (line 5). When the operation is not completed yet, `cancel()` succeeds and returns `true`, and `get()` starts returning ⊥. Also, the specified cancellation handler (line 17) is invoked in this case.

The implementation in Listing 19 is certainly not the only one that will ensure the correct work of the provided data structures. All the proofs were performed against a generalized specification of the provided code, not against the actual code. As long as a set of fairly liberal requirements (listed below) is fulfilled, it's possible to implement such an interface in a wide variety of various programming languages and libraries, and the proofs will be immediately applicable.

For example, the blocking code throughout the paper relies on Java-like behavior of aborted threads throwing `InterruptedException`, which can be caught and processed by the user, which corresponds to canceling a future. Likewise, some coroutines libraries, such as Kotlin Coroutines [6], already support an API similar to the one in `Request`.

The actual requirements that are placed on the `Future` implementation are as follows:

- A Future cannot be both cancelled and completed.
- Both cancellation and completion must happen in a logically atomic manner: there must be a single atomic operation that transfers a pending Future to one of the terminal states.
- At most one call to the cancellation handler may ever happen. If this property does not arise from the implementation of `Future`, it is easy to achieve this by replacing the cancellation handler with a version that checks whether the cancellation handler was already invoked and only invoking the original one if it was not.
- The right to complete a Future must be exclusive, that is, it must not ever be accessible by third parties. Specifically, the implementation of the CQS would be certainly incorrect if a Future stored there could be completed by something other than a call to `resume(v)`.
- There must exist an exclusive right to perform the acquisition of the logical resources stored in the completed Future. For example, in the case of a mutex, among the calls to `future.get()` that return the unit value and not `null` or $\bot$, only a single one of them actually has the right to enter the critical section.
- If the Future was ever completed or cancelled, it stays that way.

Specification. There are several logical resources introduced for the specification (see file `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/util/future.v`) :

- The *completion permit*. This is a fractional resource: it can be split into several parts, but in order to perform some operations with the completion permit, the whole resource is required. There can not exist more than one completion permit for any given Future at any time.
- The *cancellation permit*, with the same properties as the completion permit.

  Despite the name, the cancelling permit is also used as the exclusive right to acquire the logical resources stored in the Future. A separate permit could be introduced for this, but that would not affect the proof of the CQS but could complicate the specification of the Future.
- Knowledge that the Future was completed with some value v. This contradicts the knowledge that it was completed with some other value v', with any fraction of a completion permit, or the knowledge that it was cancelled. Such knowledge is freely duplicable.
- Knowledge that the Future was cancelled; also freely duplicable and contradicting the existence of any fraction of the cancellation permit.

Additionally, each Future is parameterized with some logical resource; we say that a Future is *R-passing* if its parameter is R. The following operations are supported on Futures:

- Creation of a completed Future with `ImmediateResult(..)`. When provided with an R, this method creates an R-passing Future, providing its cancellation permit and the knowledge that the Future was completed.
- Creation of an empty Future with `Request()`. It creates an R-passing Future and provides its completion and cancellation permits.
- `complete(`$\bot$`)` atomically consumes the cancellation permit and returns either `true` and the knowledge that the Future is cancelled or `false` along with the untouched cancellation permit and the knowledge that the Future was already completed.
- The `complete(v)` method obeys two specifications.

  It can be called as an atomic function that accepts the completion permit and an instance of R and behaves symmetrically to `complete(`$\bot$`)`, successfully providing the knowledge that the Future is completed, or failing, which shows that the Future was cancelled, and giving back the instance of R and the completion permit.

  Alternatively, if it is called with the completion permit and the knowledge that the Future is already cancelled, it always returns `false` and gives back the completion permit.
- The `get()` method accepts the cancellation permit as the exclusive right to perform acquisition of R. It atomically consumes the cancellation permit and either returns `null` and provides back the cancellation permit or returns a value and provides an R and half of the cancellation permit.

The specification of cancellation is not provided, as its effects heavily depend on the behavior of the cancellation handler, and the proof as a whole would become more difficult.

Invariants. We register the following invariants:

1. Each Future is empty, completed, or cancelled.
2. If the Future is empty, then `result` stores a `null`, there exist both the completion permit and the cancellation permit, and there does not exist the knowledge that the Future was cancelled or completed.
3. If the Future is completed, then `result` stores the value that the Future was completed with, along with either a copy of R or a half of the cancellation permit. The cancellation permit exists, as does the knowledge that the Future is completed.
4. If the Future is cancelled, `result` stores $\bot$, and there exist both the completion permit and the knowledge that the Future was cancelled.

Execution. The correctness of the methods can be verified by observing the effect of the atomic operation underlying each of them and checking that the resources entering and leaving the ownership of the Future are kept in balance.

E.3 The Underlying Concurrent Linked-List

The infinite array, a data structure that is key for defining the CQS, is based on a concurrent linked list. Here, we discuss the part of the infinite array that is dedicated to the management of segments.

For this proof, we abstract from `moveForwardSusp(..)` and `moveForwardResume(..)` to just `moveForward(..)` that works on any pointer to segments.

E.3.1 Specification

(see file `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/concurrent_linked_list/list_spec.v`) We introduce some additional logical resources for describing the behavior of concurrent linked lists. First, we have the knowledge that a segment is *logically removed*; this resource is freely duplicable, which implies that a once-removed segment cannot stop being removed, and is physically represented as `cancelled` being equal to `SEGM_SIZE` and `pointers` being 0 simultaneously. The second resource is the knowledge that a segment pointer p (in this program, p is either `resumeSegm` or `suspendSegm`) *points to* i, which means two things: first, that p contains a reference to a segment whose `id` is i, and second, that it owns a piece of the `pointers` counter of that segment in the following sense: as long as the piece exists, the counter stores at least 1; decreasing the counter by 1 always requires the execution to relinquish a piece of the counter. Note that the existence of a piece of a counter contradicts the segment being logically removed, which follows directly from the definitions.

If a segment n was part of the linked list at some point, then segments $[0..n-1]$ also were part of the list.

The `findSegment(s, id)` operation takes as arguments a segment s with identifier $s.\texttt{id}$ and `id`, and returns some segment t such that $t.\texttt{id} \geq s.\texttt{id}$, $t.\texttt{id} \geq \texttt{id}$, and all the segments in $[\max(s.\texttt{id}, \texttt{id}); t.\texttt{id})$ are cancelled.

The `moveForward`$[p]$`(to)` operation is a logically atomic operation that, if p points to `from`, returns `true` if p now points to the maximum of `from` and `to`.`id`, and `false` if `to` is logically removed, in which case p points to `from`.

The `onCancelledCell()` operation is a logically atomic operation that is parameterized by some logical resources Φ and Ψ such that the existence of Φ implies that the `cancelled` counter in the segment is not yet equal to `SEGM_SIZE` and it is possible to correctly increase it by 1 by relinquishing the ownership of Φ, obtaining Ψ in return. The operation atomically exchanges Φ for Ψ, which means that the ability to increase `cancelled` both existed and was utilized.

Last, setting the `prev` of a segment to `null` is always valid and has no effect. This may seem like an incorrect statement, as it would mean that not having a `prev` field at all would not affect correctness even though it would lead to `remove()` not removing segments. This is true and points to another limitation of the provided formal proofs: they do not account for memory leaks and only concern themselves with invariants not being violated and specifications being met. In fact, as will be shown later, the specification for `remove()` only claims that as long as it is only called on logically removed segments, the invariants are not violated.

E.3.2 Invariants

For each segment s of a concurrent linked list, the following holds:

- If `pointers` is 0 and `cancelled` is `SEGM_SIZE`, the segment is logically removed; otherwise, it is not logically removed and either there exist some pieces of the `cancelled` counter or `pointers` is not yet `SEGM_SIZE`.

- `prev` contains either `null` or a reference to a segment s' such that $s'.\texttt{id} < s.\texttt{id}$ and all the segments between s' and s are cancelled.
- `next` contains `null` if it is the tail segment (that is, the rightmost segment that ever existed in the list); otherwise, it contains a reference to a segment s' such that $s_{\texttt{id}} < s'_{\texttt{id}}$ and all the segments between s and s' are cancelled.

  Observe that this invariant implies that the current tail segment is always accessible from any other segment by following `next` repeatedly: the chain of `next` may only end with the tail.

E.3.3 Execution

(see file `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/concurrent_linked_list/list_proof.v`)

`removed()` This method checks whether the segment is logically removed. If it is, the method returns `true` and provides the knowledge that the segment is logically removed. Otherwise, it just returns `false`.

`remove()` We claim that this operation does not violate any invariants if it is called on a logically removed segment.

First, if the segment turns out to be the tail, nothing is done. Otherwise, the segment is not the tail and can not become one anymore by definition.

Observe that `aliveSegmLeft(s)` returns either `null` or a segment s' such that $s'.\texttt{id} < s.\texttt{id}$ and all the segments between s' and s are cancelled. This holds for the initial value of `cur` due to the invariant on `prev`, and each loop iteration preserves this, according to the same invariant on `prev`, which can be easily observed.

Likewise, `aliveSegmRight(s)`, when called on a non-tail segment, returns a segment s' such that $s.\texttt{id} < s'.\texttt{id}$ and all the segments between s and s' are cancelled. Note that `null` can not be returned from this method.

Knowing this, we can easily observe that variables `prev` and `next` defined in `remove()` satisfy precisely the invariants for when a value would be a valid content of fields `prev` or `next` respectively.

Then, the operation (which does not require any additional resources to be initiated) is retried if it turns out that `prev` and `next` could be validly pointing even further.

findSegment(start, id) We establish a loop invariant that `cur` contains a reference to a segment s whose `id` is not less than that of `start`, and also all segments from $\max(\texttt{start.id}, \texttt{id})$ (inclusive) to $s.\texttt{id}$ (exclusive) are cancelled. This holds initially since there are no such segments for $s = \texttt{start}$.

If it turns out that $s.\texttt{id}$ is not less than `id` and s is not removed, then, by the loop invariant, all the requirements for the return value of the method are fulfilled.

Otherwise, it is known that either $s.\texttt{id}$ is less than `id` or s was removed. Two possibilities are then considered:

- s is not the current tail. Then, according to the invariant on `next`, all the segments with $\texttt{id} \in (s.\texttt{id}; s.\texttt{next.id})$ are cancelled. To preserve the loop invariant, it is required to show that this fact implies that all the segments with $\texttt{id} \in [\max(\texttt{start.id}, \texttt{id}); s.\texttt{next.id})$ are cancelled.

  If $s.\texttt{id}$ is less than `id`, then $s.\texttt{next.id} \leq \texttt{id}$, so this is vacuously true. Otherwise, this loop iteration was performed because s was removed. In this case, combining $[\max(\texttt{start.id}, \texttt{id}), s.\texttt{id}) \cup \{s.\texttt{id}\} \cup (s.\texttt{id}; s.\texttt{next.id})$ provides the required interval of cancelled segments.
- s is the current tail. In this case, a new segment must be appended. The only valid `id` for a segment to be appended is $s.\texttt{id}+1$, according to the invariant on `next`. An attempt is performed to append the new segment. Whether it succeeds or not, now it is known that s is not anymore the current tail, as either `next` already contained something or it does now as the result of a CAS. If the CAS succeeded, then, if the segment is logically removed, a call to `remove()` is performed, which is valid. This is desirable since a call to `remove()` that was performed after the segment became logically removed could have finished without performing any work, having observed that the segment was the tail.

  In any case, now that s is not the current tail, the case reduces to the first one.

tryIncPointers() This method atomically checks the current contents of the fields `pointers` and `cancelled` in the current segment and, if either of them is not in the terminal state, returns `true` and increments `pointers`, providing the newly-created piece of the `counter`; otherwise, it leaves everything unchanged and returns `false` and provides the knowledge that the segment is logically removed.

decPointers() This method, when called with a piece of the `pointers` counter, consumes that piece, decrementing the counter and returning `true` if the segment became logically cancelled as a result, or `false` otherwise. This is fairly easy to observe from the definitions. The tricky part is that the call does not violate the invariants, but this follows from the fact that the existence of a piece of the counter implies that the current value is nonzero.

moveForward(to) The i that the segment pointer currently points to is read into `cur`. If its `id` is at least `to.id`, then the moment of that reading is the point where the method performs its atomic action, and `true` is returned.

Otherwise, an attempt is made to acquire a piece of the `pointers` counter of segment `to`. If it fails, the segment must have been cancelled, so the whole method returns `false`. If it succeeds, this means that the call obtains a piece of `to.pointers`. `CAS` is then attempted.

If the `CAS` fails, the piece of `to.pointers` is used to call `to.decPointers()`, which may inform us that `to` is logically cancelled, in which case `to.remove()` is called.

If the `CAS` succeeds, then the piece of `to.pointers` is transferred to the segment pointer so that it logically points to `to.id`. A piece of `from.pointers` is acquired instead and then used to call `from.decPointers()`, which, likewise, can show us that a call to `from.remove()` could be valid.

`onCancelledCell()` The difficult part of this proof was finding a suitable specification; the correctness of the execution follows from that. Given a Φ, in order to perform the `Fetch-And-Add`, this method atomically obtains the right to increase `cancelled` along with the knowledge that it was not yet `SEGM_SIZE`. Increasing the `cancelled` yields a Ψ, which is provided to the caller. Then, if the segment is logically removed, `remove()` is called.

Setting `prev` to `null` No invariants are violated by this: it is always valid for a `prev` of any segment to contain `null`.

E.4 Infinite Array Specification

The CQS does not actually need a true infinite array, which would use an unbounded amount of memory. Instead, the data structure that is employed after all is able to discard groups of cancelled cells and even lose access to cells from the prefix of the "array" that are no longer needed. Thus, when the data structure underlying the CQS is called an "infinite array", the term is used loosely and for the lack of a more fitting name.

The code listings are defined (in order to avoid excessive abstractions that would detract from the general idea) with the ad-hoc "infinite array" operations being interwoven with the logic of the CQS. However, it could just as easily have been abstracted into a separate data structure by grouping a segment and an index in that segment into an entity called a *cell*: for example, (s, i) is the i'th cell in segment s, but its index in the infinite array is $s.\texttt{id} \cdot \texttt{SEGM\_SIZE} + i$. This is the approach taken in the Coq formalization to be able to prove the infinite array operations independently from the CQS. The operations provided by the ad-hoc infinite array are then the following (see file `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/concurrent_linked_list/infinite_array/array_interfaces.v`) :

- Array creation that allocates n segment pointers (with $n = 2$ in case of the CQS, the pointers being `resumeSegm` and `suspendSegm`)—performed by allocating the concurrent linked list.
- Reading a segment pointer.
- Moving a segment pointer forward—done with a call to `moveForward(..)`.
- Finding a cell with the given `id`—performed with a call to $s' =$ `findSegment(s, id / SEGM_SIZE)` and later checking whether the requested segment identifier corresponds with the requested one and returning $(s', \texttt{id} \bmod \texttt{SEGM\_SIZE})$ if so and $(s', 0)$ otherwise.
- Cancelling a cell—done with a call to `onCancelledCell()` on the underlying segment;
- Setting `prev` of the underlying segment of a cell to `null`;
- Checking the index of a cell—equal to $s.\texttt{id} \cdot \texttt{SEGM\_SIZE} + i$, where the cell is (s, i);
- Accessing the contents of a cell.

The code listings can easily be rewritten in terms of these operations (which would lead to `SEGM_SIZE` not ever being mentioned outside of the infinite array abstraction) that, when inlined, would result in the code that is present currently. The specifications of the listed operations mirror those of the operations that they are wrapping.

Notable additions that the infinite array performs in the logical realm when compared to the general concurrent linked list are the following (see file `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/concurrent_linked_list/infinite_array/array_spec.v`):

- Introduction of a *cancellation handle for the i'th cell*. This logical resource is provided for each cell and is the Φ that is passed to `onCancelledCell()` when called on an infinite array segment. In accordance to the behavior required from Φ, the existence of the cancellation handle for the cell i from a segment s prevents `cancelled` from being equal to `SEGM_SIZE`, and thus implies that s is not yet logically removed.

  There can only exist a single cell cancellation handle for any given cell.

- The Ψ acquired from `onCancelledCell()` is another logical resource, the knowledge that the i'th cell was cancelled. This knowledge is mutually incompatible with the existence of the cancellation handle for the i'th cell.

  This resource is freely duplicable.

- A logical operation of accessing an infinite array cell for the first time is introduced. This operation provides each caller either with both the exclusive right for writing to the cell and the cell cancellation handle, or with the evidence that the caller was not in fact the first one to attempt this operation; this evidence is called the knowledge that the *cell is owned*. The user of the infinite array themselves decide what logical resource is used to signify that the cell is owned.

The implementations of these logical entities (see file `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/concurrent_linked_list/infinite_array/array_proof.v`) are purely technical and do not provide the reader with a deeper understanding of the subject matter, so their existence is simply postulated in this text.

E.5 Infinite Array Iterators

An additional construct is built on top of the infinite arrays—*iterators*. An iterator is a pair of a segment pointer and a counter; for example, `resumeSegm` and `resumeIdx` form an iterator, as do `suspendSegm` and `suspendIdx`.

Iterators provide a single operation `step()`, an example of which is shown in 20. In the actual code, the operation is inlined and simplified, but due to how crucial it is to the proof, it is considered separately.

```
fun stepResume(): (Bool, Cell) {
  r := resumeSegm
  i := FAA(&resumeIdx, 1)
  s := findAndMoveForwardResume(r, i/SEGM_SIZE)
  if s.id == i/SEGM_SIZE: return (true, Cell(s, i))
  else: return (false, Cell(s, 0))
}
```

Listing 20: An implementation of an array step for the dequeue iterator. This is not used in the actual code and is only introduced as an abstraction to separate the parts of the formal proof.

Specification (not in a separate file due to being seemingly non-generalizable) Each iterator is parameterized by some logical resource R; we say that an iterator is *bounded by R*.

Each iterator introduces a logical resource that is parameterized by a nonnegative number i. We call this resource the *i'th permit from the iterator*. This permit has two important properties:

- It is exclusive: at most one permit exists for any given iterator for any number i.
- It implies that the iterator's counter is at least $i + 1$.

The `step()` operation acquires an instance of an R and returns one of two possible results:

- A pair of `true` and a cell c with index i. In this case, the i'th permit from this iterator is provided.
- A pair of `false` and a cell c with index j. In this case, the i'th permit from this iterator is provided for some $i < j$, and all the cells in $[i; j)$ are known to be cancelled.

Last, a logical operation of accessing the bounding resource is available. This operation allows the caller to observe that there are at least $i + 1$ copies of R stored in the iterator if there exists the i'th permit. The correctness of this operation follows directly from the invariants that follow.

Invariants The state of the invariant is described by the number n currently stored in its counter. Then the following is true:

- The iterator stores n copies of R.
- The segment pointer *points to* (in the sense of containing the reference to a segment and owning a piece of its `pointers` counter) some segment i such that all the cells in $[n; i \cdot \texttt{SEGM\_SIZE})$ are cancelled. Note that it is possible for $i \cdot \texttt{SEGM\_SIZE}$ to be n or less, in which case no knowledge about cancelled cells is present.
- There exists the i'th permit for all $i \in [0; n)$.

Additionally, n is nondecreasing with respect to time, that is, the iterator can only go forward.

Proof of the Step Operation. (see file `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/concurrent_linked_list/infinite_array/iterator/iterator_impl.v`) First, the segment pointer is read. The counter contained some number n at the moment, and, according to the invariant, the segment pointer contained a reference to some segment r such that all cells in $[n; r.\texttt{id} \cdot \texttt{SEGM\_SIZE})$ are cancelled.

Next, the counter is incremented. If at that moment it contained i, then the i'th permit from the iterator is created; it is also known that $i \geq n$, due to monotonicity of the iterator.

Then `findAndMoveForward(r, i/SEGM_SIZE)` is executed. According to its specification, it returns some segment s such that $s.\texttt{id} \geq r.\texttt{id}$, $s.\texttt{id} \geq i/\texttt{SEGM\_SIZE}$, and all the segments in $[\max(r.\texttt{id}, i/\texttt{SEGM\_SIZE}); s.\texttt{id})$ are cancelled.

If $s.\texttt{id}$ is $i/\texttt{SEGM\_SIZE}$, then `true` is returned, and the cell is formed correctly, obeying the provided specification. Otherwise, `false` is returned, along with a cell with index $s.\texttt{id} \cdot \texttt{SEGM\_SIZE}$. The specification of `step()` requires that we show then that $i < s.\texttt{id} \cdot \texttt{SEGM\_SIZE}$ (which easily follows from $i/\texttt{SEGM\_SIZE} < s.\texttt{id}$) and that all cells in $[i; s.\texttt{id} \cdot \texttt{SEGM\_SIZE})$ are cancelled, which requires further elaboration.

We consider two possibilities in order to prove it.

- $i/\texttt{SEGM\_SIZE} \geq r.\texttt{id}$. Then we know from the information provided by `findAndMoveForward(..)` that all the segments in $[i/\texttt{SEGM\_SIZE}; s.\texttt{id})$ are logically removed, which means that all the cell in $[i; s.\texttt{id} \cdot \texttt{SEGM\_SIZE})$ are cancelled, which is what we needed to prove.
- $r.\texttt{id} > i/\texttt{SEGM\_SIZE}$. Then `findAndMoveForward(..)` guarantees that segments in $[r.\texttt{id}; s.\texttt{id})$ are logically removed, so cells in $[r.\texttt{id} \cdot \texttt{SEGM\_SIZE}; s.\texttt{id} \cdot \texttt{SEGM\_SIZE})$ are cancelled. It is also known that all cells in $[n; r.\texttt{id} \cdot \texttt{SEGM\_SIZE})$ are cancelled and that $n \leq i$; thus, all cells in $[i; r.\texttt{id} \cdot \texttt{SEGM\_SIZE})$ are cancelled. By combining the two intervals, we obtain the desired proposition.

E.6 Proving Correctness of the CQS

This is by far the most involved part of the proof, taking about a third of the lines of the proof code on its own (see file `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/thread_queue/thread_queue.v`) .

E.6.1 Specification

The general idea is that a CQS with the enqueue resource E and the dequeue resource R is a data structure that allows callers of `suspend()` that provide an instance of E to register a Future that, upon being completed, receives an instance of R, and callers of `resume(v)` that provide an R to receive an E upon completion. In the mutex example, E is the empty resource — no special permissions are required to call `lock()` — and R is the permission to enter the critical section.

More accurately, both `resume(v)` and `suspend()` are split into a logical and a physical stage. Actually calling `resume(v)` requires providing not an R but an *awakening permit*, a logical resource introduced by this data structure; it is the awakening permit that can be acquired in exchange for R, which is done on the logical level. Likewise, a call to `suspend()` requires a *suspension permit* that can be acquired beforehand in exchange for E.

When correctly called with a suspension permit, `suspend()` returns an R-passing Future, but also provides the means of cancelling that Future: a complete (but ineffectual) cancellation permit if the Future was constructed with `Imme-`

diateResult, and a half of its cancellation permit otherwise. The other half is owned by the CQS, but is always available, so owning a half of the cancellation permit is sufficient for cancelling the Future. The complication that requires the protocol to require the canceller to access the invariants of the CQS to cancel the Future is due to the need for the CQS to reliably observe the effective state of each cell, which is affected by cancellation.

If the resumption is synchronous, both `resume(v)` and the corresponding `suspend()` may fail due to the cell being broken by the resumer; additionally, `resume(v)` may fail if the cell was "simply" cancelled. In both failure modes for `resume(v)`, instead of receiving an instance of E upon its completion, the caller instead gets an R, allowing the operation to be restarted. Likewise, when `suspend()` fails, it provides an instance of E and doesn't return a Future.

The state of the CQS is represented as the number of threads that are currently enqueued or are willing to be. We call this number the *size* of the CQS; a more specific definition will be provided later.

Users of the CQS are responsible for representing the size of the queue as physical values. In the mutex example, the size is 0 if `state` is nonnegative and $-$`state` otherwise. The CQS supports the following operations on its logical state:

Enqueue registration It is possible to provide an E, increasing the size and obtaining in exchange a suspension permit — a permission to call `suspend()`.

Dequeue registration If it is known that the size is nonzero, it is possible to decrease the size and provide an R, obtaining in exchange an awakening permit — a permission to call `resume(v)`.

Cancellation registration Used in the "smart" cancellation mode and implemented for use in `onCancellation()`, this operation can only be invoked once for each cell; this is ensured by introducing yet another set of logical resources, which don't influence the general idea and will be described later. If the CQS size at the time of this operation is 0, then cancellation registration does not change the size, provides the caller with an instance of R and logically performs the transition of the state of the cell to `REFUSED` (without physically writing anything to the cell). Otherwise, the operation decreases the size of the CQS, provides the caller with the cancellation permission for the cell in the infinite array, and performs the transition of the cell state to `CANCELLED`.

E.6.2 Per-queue Invariants

Two logical values are maintained: the *dequeue front*, and the *cell state list*. The *deque front* is the first cell about which it is not yet known that its state is going to be observed by `resume(v)` or already was. A cell is *inside the deque front* if its index is less than the deque front. *Cell state list* stores the authoritative knowledge of the state of each *relevant* cell in the infinite array; a cell is considered relevant if it is known that its state is going to be observed by a call to `suspend()` or already was.

The last cell inside the deque front must be a relevant cell, which effectively means that we forbid calling `resume(v)` unless it is known that the corresponding `suspend()` is also eventually going to be called.

It is maintained that the last cell before the deque front can not be *skippable*. A cell is skippable if it was "smartly" cancelled and `onCancellation()` returned `true`. Without this invariant, the definition of the deque front would be violated: if it is known that the last cell inside the deque front is skippable, it is going to be observed by a call to `resume(v)`, but is going to be skipped (hence the name), and thus some of the following cells are also known to be observed by `resume(v)` at a later time, thus fitting in the deque front, which contradicts the skippable cell being the last such cell.

For each *relevant* cell, there exists a single instance of a logical resource called the *suspension permit*. For each cell before the deque front, there exists a single instance of a logical resource called the *awakening permit*.

The deque front is nondecreasing, which is in line with its definition: once it is known that a cell is about to be witnessed by a call to `resume(v)`, this can not become false. Likewise, the length of the cell state list can not decrease with time.

Physically, a CQS consists of an infinite array and two iterators, one, the *enqueue iterator*, bounded by the suspension permits, and another, the *dequeue iterator*, bounded by awakening permits.

Finally, the *size* of a CQS is defined to be the number of nonskippable relevant cells outside the deque front. This is the value in terms of which the programmer-facing specifications are defined.

E.6.3 Cell States

The descriptions of cell states contained in the cell state list closely mirror the state transition systems presented in Figure 2 and Figure 4, as in most cases the logical state of the cell is adequately described by its contents. Some

superficial differences are that here, a cell stores a `Future` instead of a thread, and a `Future` filled with a value instead of just the value.

A change that is actually significant is that simple cancellation and smart cancellation are not handled separately due to the amount of shared parts between them. Instead, a single state transition system is implemented, where there are two possible transitions from $\texttt{FUTURE}_{\texttt{WAITING}}$ to a cancelled state, one for each cancellation mode. However, for each use of `CancellableQueueSynchronizer` it must be true that either all the cancelled cells are "smartly" cancelled or all the cells are "simply" cancelled. Otherwise, it would be impossible for the `resume(v)` operations to determine whether a cell in a removed segment is responsible for finalizing its own cancellation (as is done in "smart" cancellation).

A much more significant change is that the transition system described in Figure 2 and Figure 4 is presented in terms of values that the cell contains at any given moment; however, the state transition in terms of observable behavior sometimes happens before anything is written to the cell or does not occur at all. For example, it does not matter for correctness whether a resumed Future or a `RESUMED` is in the cell, the observable behavior is exactly the same in either case, so the two states are merged into one. Contrastingly, there are many more additional states required for describing smart cancellation due to the need to always distinguish cancelled cells where `onCancellation()` returned `true`, as such cells, being skippable, are important for defining the current state of the CQS. When a cell is smartly cancelled, it is initially `UNDECIDED` and stores the Future; "undecidedness" here is in regards to whether the cancellation registration will succeed. If cancellation registration fails, the cell is `REFUSED` even before the corresponding marker is written to it; otherwise, a race can happen between writing `CANCELLED` and a resumer passing a value in the asynchronous mode. `SMARTLY-CANCELLED` is the state when the cancellation registration has already succeeded, and `CLOSED` is the state when `CANCELLED` was written to the cell before the resumer managed to write its value. This race can be decided even before the cancellation registration attempt, if the resumer passes its value when the cell is still in the `UNDECIDED` state.

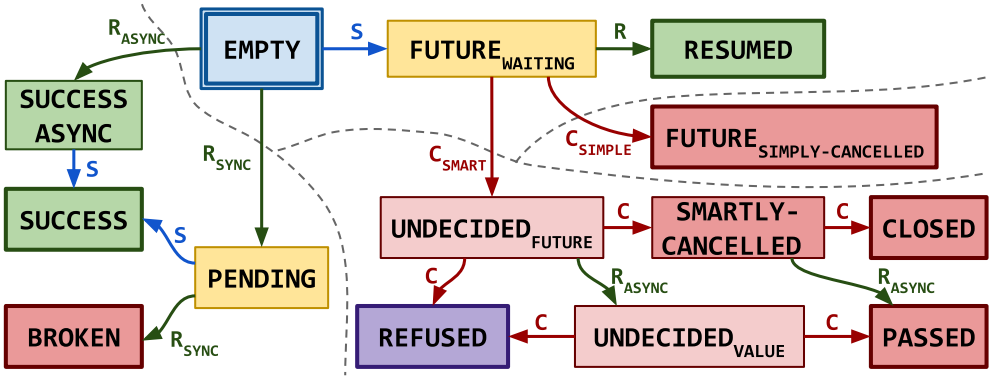

Figure 14: The state transition system for a single cell from the logical perspective.

E.6.4 Per-cell Tokens

Some of the logical resources only exist when a cell is in a particular state; each of the resources listed here is exclusive and parameterized by the cell for which it is defined.

- The *cell breaking token* represents the exclusive right of the execution of `resume(v)` to break the cell in case of elimination. This token exists in every state from the "elimination" execution path except for `BROKEN`, namely `SUCCESS-ASYNC`, `SUCCESS`, and `PENDING`. This token is generated on a successful write of a value to an empty cell and is only destroyed if the cell is broken.
- The *cancellation registration token* is generated when a Future stored in the CQS is successfully cancelled. It represents the exclusive right to call `onCancellation()`, which, in term, must use it to attempt the logical operation of cancellation registration, which destroys the cancellation registration token and instead creates a cell cancelling token.
- The *cell cancelling token*, provided by the cancellation registration, represents the right to write `CANCELLED` or `REFUSED` to the cell.

E.6.5 Requirements for User-Defined Operations

For the CQS to operate correctly, it is crucial that the operations provided by the user behave in accordance with the established invariants.

`completeRefusedResume(v)` must define some resource X such that `completeRefusedResume(v)` returns E on completion when called with it.

`onCancellation()` This method is given a cancellation registration token and must invoke the operation of cancellation registration. If the registration is successful, it must return `true`; otherwise, it must return `false` and ensure the existence of an instance of X. The operation of cancellation registration also generates a cell cancelling token and, in case of a successful cancellation, provides a cancellation handle; these resources must be passed along, which can not be reflected in the code but is required in the formal specification.

E.6.6 Per-cell Invariants and Transitions

Each *relevant* cell i is said to own various collections of logical resources depending on its state. This description heavily relies on Figure 14; so the reader is advised to consult with it periodically. In particular, the fact that some transitions are impossible (for example, the states in the elimination path are inaccessible once a Future is written to the cell) is only reflected in that figure. In the actual Coq proof, this, too, had to be specifically encoded using logical resources; we omit these details here, as they would greatly increase the size of the proof but would not provide additional clarity.

- **`EMPTY`.** By definition of a *relevant* cell, E was passed to it during the enqueue registration, and no state transitions happened yet to extract that resource, so the cell owns a E. Additionally, it is possible that the cell is already inside the dequeue front, in which case the cell also owns an R.
- **`resume(v)` arrived first.** If the call to `resume(v)` was the first to modify the cell, we have the following:
  - The cell owns the cancellation permit for the infinite array cell, meaning that the cell is never going to be cancelled;
  - The cell owns the i'th permit from the dequeue iterator.

  There are some additional resources that the cell owns, depending on its state:
  - **`PENDING`.** The cell contains some value v, as well as both E and R. This state can only be entered when in the synchronous resumption mode.
    Transition from `EMPTY` to `PENDING` happens when the resumer writes its value to the cell, providing the i'th permit from the dequeue iterator and receiving the cell breaking token. The transition happens as follows: given that the i'th permit from the dequeue iterator exists, i is clearly inside the deque front, so initially the cell owned E and R. When the resumer first interacted with the cell, its cancellation handler was initialized. Therefore, we have all the resources that the cell needs to own. Additionally, the transition to `PENDING` creates the cell breaking token, which is taken by the resumer.
  - **`BROKEN`.** The cell contains the `BROKEN` marker. It also owns either the i'th permit from the enqueue iterator or an E. This state can only be entered when in the synchronous resumption mode.
    Transition from `PENDING` to `BROKEN` happens when the resumer stops waiting for the corresponding call to `suspend()` and writes `BROKEN` to the cell, using up its cell breaking permit and getting back an R in return. The structure of this transition is trivial.
    When `suspend()` observes `BROKEN` in the cell, it gives up the i'th permit from the enqueue iterator in exchange for the E, performing no state transitions in the process.
  - **`SUCCESS-ASYNC`.** The cell contains some value v, owns an R and this cell's breaking token.
    Transition from `EMPTY` to this state happens in the asynchronous resumption mode when the resumer writes its value to the cell, relinquishing the i'th permit from the dequeue iterator in exchange for E. The cell was inside the deque front, so it owned an R; also, the transition creates the cell breaking token, which is not given to the resumer but is instead kept in the cell's ownership.
  - **`SUCCESS`.** The cell contains the `TAKEN` marker and owns the i'th permit from the enqueue iterator, as well as either E or this cell's breaking token.
    Transition from `SUCCESS-ASYNC` happens when `suspend()` writes `TAKEN`, having observed that the cell contains a value, providing its i'th permit from the enqueue iterator in exchange for R.
    Tranisition from `PENDING` happens under the same conditions.
    If the resumption is synchronous, the resumer can observe the state transition and give up its cell breaking token in exchange for E.
- **`suspend()` arrived first.** If the call to `suspend()` was the first to modify the cell, writing a Future to it, the following is true:
  - The cell owns the i'th permit from the enqueue iterator.
  - There exists a unique R-passing Future f associated with the cell.

To describe the resources the cell owns at various states, we first need to introduce a helpful ownership pattern. We say that T is owned as a *resource for resumer* if one of the following is true:

- The Future completion permit for f and T is owned;
- Half the Future completion permit, the i'th permit from the dequeue iterator, and T is owned;
- The Future completion permit for f and the i'th permit from the dequeue iterator is owned.

The logic behind this notion is that it would be unfeasible to use the state transition system to carefully track the progress that `resume(v)` has made, as that would greatly multiply the number of states. Instead, in the interesting case, the resumer goes through the following stages when working with the cell, during which the cancellation process could advance the state several times:

- It owns the i'th permit from the dequeue iterator (and the system owns the Future completion permit and some T). After observing that the cell contains a Future, the resumer trades the dequeue iterator permit from the iterator for a half of the Future completion permit (it can not take the whole permit, as then the cell state would not be able to uniquely identify the corresponding state of the Future).
- The resumer tries to complete the Future. If it succeeds, then a state transition occurs. Otherwise, the completion attempt does not have an effect. In this case, the resumer trades back its half of the completion permit without performing a state transition and either receives the i'th permit from the dequeue iterator back or gets a T.

It should be noted that, with this ownership scheme, it is always possible for the resumer to determine exactly what the cell owns; this is due to the exclusivity of the iterator permits and Future completion permits.

We also say that *resumer has finished* if the cell owns a combination of the i'th permit from the deque iterator and the completion permit for f. The intuition behind this notion is that in the part of the state transition system where a Future was written to the cell it is impossible for the resumer to perform any operation without owning some of these two resources, so if they belong to the cell, the resumer can no longer perform meaningful operations. Due to the invariants of the system, it is only possible for the resumer to have finished if the cell is inside the deque front.

- **$\texttt{FUTURE}_{\texttt{WAITING}}$.** The cell contains f. It also owns the cancellation handle of this cell, as well as E. If the cell is inside the deque front, it owns R. It owns a half of a Future cancellation permit, and also owns unit (that is, an always-true statement that bears no information) as a resource for resumer.

  The transition from `EMPTY` to $\texttt{FUTURE}_{\texttt{WAITING}}$ happens when `suspend()` writes a Future f to the cell. It provides to the cell the i'th permit from the dequeue iterator, the full completion permit and half of the cancellation permit for f; in return, it receives the knowledge that f is now part of the CQS. The non-obvious parts of the transition is that the cell owns the cancellation handle and owns the unit as a resource for resumer. The cancellation handle is allocated with the first access to the cell; the unit resource for resumer has the form where the cell owns the full completion permit and T, with the completion permit being provided by `suspend()` at the start of the transition, and T, being the unit, always trivially available.
- **`RESUMED`.** The cell contains either `RESUMED` or f. It owns the i'th permit from the dequeue iterator, knowledge that the Future was completed, the cancellation handle, and half of a cancellation permit.

  The transition from $\texttt{FUTURE}_{\texttt{WAITING}}$ to `RESUMED` happens when `resume(v)` successfully completes f. For this to happen, it must have given up its i'th permit from the dequeue iterator in exchange for the half of a completion permit when the cell was still `WAITING` (utilizing the ownership of unit as a resource for resumer). Thus, during the transition, `resume(v)` relinquishes its half of the completion permit in exchange for E. At the start, the cell owns a half of the completion permit, thus, the full completion permit is available for this operation and it is possible to safely try to complete the Future. However, in order to do that, it is also needed to provide an R. To obtain it, we observe that the interaction of `resume(v)` with the cell implies that it is inside the deque front, so R is owned by the cell at the start of the operation.
- **$\texttt{FUTURE}_{\texttt{SIMPLY-CANCELLED}}$.** The cell contains `CANCELLED` or f and owns the knowledge that the Future is cancelled. It also owns as a resource for resumer an R if the cell is inside the deque front or unit if it is not. This state is only available with the simple cancellation mode.

  Transition to $\texttt{FUTURE}_{\texttt{SIMPLY-CANCELLED}}$ from $\texttt{FUTURE}_{\texttt{WAITING}}$ happens when the Future is successfully cancelled. The half of the Future cancellation permit owned by the cell and the half in possession of the end user of the CQS are combined to perform the cancellation, which, upon success, provides the knowledge that the cell was cancelled. The cancellation handler that is owned by the cell at the start of the operation is given to the canceller so that it is able to call `onCancelledCell()`.
- **The Future is smartly cancelled.** The cell owns the knowledge that the Future was cancelled.

* **UNDECIDED.** The cell owns the cancellation handle and, if it is inside the deque front, an R. The cell may contain one of two things:
  1. f. In this case, the cell owns E and also owns the unit as a resource for resumer.
     Transition from $\texttt{FUTURE}_{\texttt{WAITING}}$ to this happens when the Future is cancelled. The cancelling operation gives up its half of the cancellation permit which, combined with the half that was stored in the cell, allow cancelling the Future. In exchange, the cancellation registration token is created and given to the cancelling operation.
  2. The cell can contain some value v. In this case we know that the resumer has finished.
     There is no transition from $\texttt{FUTURE}_{\texttt{WAITING}}$ directly to this form; instead, this is what happens in the "smart async" cancellation mode when the state is already `UNDECIDED`, so the Future is cancelled, the resumer unsuccessfully attempts to resume it and writes its value to the cell, relinquishing its permits (and finishing) in exchange for the E that was owned as a resource for resumer.
* **REFUSED.** The cell owns the cancellation handle and is known to be inside the deque front. There are three possibilities for what the cell can contain:
  1. f. Similarly to the case for `UNDECIDED`, the cell owns E and owns the unit as a resumer resource.
     Transition from the first form of `UNDECIDED` happens when cancellation registration fails due to the CQS being logically empty. The cancelling operation receives the cell cancelling token and, since the registration fails only when the cell is inside the deque front, an R in exchange for its cancellation registration token needed to invoke `onCancellation()`.
  2. Some value v. Then the resumer has finished.
     Transition from the second form of `UNDECIDED` happens when cancellation registration fails due to the CQS being logically empty. Transition from the first form of `REFUSED` happens similarly to the transition from the first to the second form of `UNDECIDED`.
  3. `REFUSED`. Then the cell owns the cancelling token and owns as a resource for resumer such a resource X provided by `onCancellation()` that `completeRefusedResume(v)` is guaranteed to provide an E if given an X.
     Transition to this from the first form happens when the cancelling operation writes `REFUSED` to the cell, giving up its cell cancelling token and the instance of X received from `onCancellation()` and thus finishing the cancellation process.
     Transition from the second form to this happens when writing `REFUSED` and giving up its cell cancelling token, but keeping the instance of X to be able to call `completeRefusedResume(v)` from the cancellation handler. Keeping the X is possible because the resumer is known to have finished, so owning X as a resource for resumer has its final form where X is not actually owned by the cell.
* **Cancellation was allowed.** When cancellation registration succeeds, a state from this group is entered; all of these states are skippable. For each of them, the awakening permit is mentioned, which raises a question of where would the awakening permit come from. The answer is that providing cells with awakening permits is the responsibility of either cancellation registration or dequeue registration.
  Recall that an awakening permit exists for each cell inside the deque front; observe also that if a skippable cell is inside the deque front, then it is known that it will be skipped by a resumer, so the deque front is increased until it encounters a nonskippable cell (which must exist, since both cancellation registration and dequeue registration only succeed for non-empty queues), possibly skipping many cancelled cells. Thus, for each skippable cell inside the deque front a new awakening permit is generated; this is the permit that is given to the cell as the result of the operation.

  SMARTLY-CANCELLED The cell stores f and owns E. It also owns as a resource for resumer the resource that is the awakening permit if the cell is inside the deque front and the unit otherwise.
  The transition from the first form of `UNDECIDED` happens as follows. The cancellation handle that was owned by the cell is given to the cancelling operation. If this cell was inside the deque front and owned an R, then that R is moved to the new end of the deque front, which must have been in the `EMPTY` or $\texttt{FUTURE}_{\texttt{WAITING}}$ state, given that the resumer could not have yet interated with that cell and the end of the deque front can not be a cancelled cell. The awakening permit is then generated and provided to the cell as described above. If the cell was outside the deque front, then the only change to the resources owned by the cell is the cancellation handle provided to the cancelling operation.

  PASSED It is known that the resumer has finished. Additionally, the cell may store some value v and own the awakening permit, or it may store `CANCELLED` and own the cancelling token.

The transition from the second form of UNDECIDED happens in almost the same way as the transition to SMARTLY-CANCELLED, with the notable difference being that since it is known that the resumer has finished, the cell is inside the deque front, so a new awakening permit was generated for it.

A transition can also happen from SMARTLY-CANCELLED when the resumer finishes, getting an E in return.

Finally, when the cancelling operation writes CANCELLED, it receives the awakening permit in exchange for its cancelling token.

CLOSED The cell stores CANCELLED, owns the cancelling token, and owns as a resource for resumer a resource that is the awakening permit if the cell is inside the deque front and the unit otherwise. The transition from SMARTLY-CANCELLED here happens when the cancelling operation successfully replaces f with CANCELLED, providing its cancelling token in exchange for E.

E.6.7 Proofs of Logical Operations

Enqueue Registration This is the simplest operation to prove. Given an E, we say that the first cell that we did not consider relevant before is now EMPTY. R does not need to be provided because the cell can not be inside the deque front, as, by an invariant, before the registration started, the last cell inside the deque front was relevant, so it could not have been the cell in question, and enqueue registration does not advance the deque front.

That the size of the CQS is increased is obvious from the definition.

A suspension permit exists for each relevant cell, so it is valid to allocate one.

Dequeue Registration By definition of the CQS size, if it is nonzero, then there exists a nonskippable relevant cell outside the deque front. We say that the first such cell has an index of $d + i$, where d is the current deque front and $i \geq 0$. Then $i + 1$ more cells than before are going to be observed by a call to `resume(v)`: in order to access the cell $d+i$, calls to `resume(v)` will have to observe the i skippable cells in addition to the new nonskippable one. Therefore, the deque front is increased by $i + 1$, and $i + 1$ awakening permits are generated.

We know that the i skippable cells were not inside the deque front at the start of this operation; observe also that a cell being skippable means that its state is, by definition, from a group of states where the cancellation succeeded and can not be PASSED (as a cell can only be in this state if it is inside the deque front); therefore, each of these cells is either SMARTLY-CANCELLED or CLOSED and owns the unit as a resource for the resumer. Now that the deque front is advanced, these cells must instead own an awakening permit as the resource for the resumer. Therefore, i of the $i + 1$ allocated awakening permits are passed to individual skippable cells to satisfy their invariants, and the last awakening permit is provided to the caller.

That the size of the CQS is decremented follows directly from the definition.

Cancellation Registration If the queue was empty, this means that every cell that is not yet cancelled is inside the deque front, including the one we attept to cancel currently. Thus, a transition is performed from UNDECIDED to REFUSED, providing an R and a cell cancelling token.

Otherwise, the queue was not empty. A transition is performed from UNDECIDED to either SMARTLY-CANCELLED or PASSED, depending on whether a value was already written to the cell, providing a cancellation handle. If the cell was outside the deque front, this change is sufficient, as the CQS size is obviously decremented due to this cell becoming skippable. Otherwise, the transition additionally requires an awakening permit and provides an R. This R is then used to perform a deque registration. The awakening permit that is obtained in such a way is used to complete the transition.

E.6.8 Execution

`resume(v)` To reiterate what was said in discussion of the specification of this method, it requires passing an awakening permit to it and either finishes with `true` and provides an E, or it finishes with `false` and provides an R, which is only possible in the synchronous resumption mode or smart cancellation mode.

First, a single step of the dequeue iterator is performed. If it is successful, the i'th cell is obtained along with the i'th permit from the dequeue iterator. The correctness of the execution then follows from the described transitions of the cell state. Otherwise, the step is unsuccessful, and the j'th cell is obtained with the i'th permit from the dequeue iterator, where cells in $[i; j)$ are known to be cancelled.

If the cancellation mode is "simple", then the i'th cell must be $\texttt{FUTURE}_{\texttt{SIMPLY-CANCELLED}}$. Given that the i'th permit from the dequeue iterator exists, the i'th cell is inside the deque front, and so stores R as a resource for the resumer. The resumer then obtains R and finishes the call to `resume(v)` with `false`.

Otherwise, the cancellation is "smart", in which case a CAS is performed to set the counter of the dequeue iterator to be at least j. If the CAS fails, it has no effect and so does not affect the correctness. If it succeeds, then it is known that the counter in the dequeue iterator must still contain $i + 1$. For this increase of the counter from $i + 1$ to j to be valid, the invariant of iterators requires that $j - i - 1$ copies of the awakening permit be provided. However, this is always possible. Observe that the permits $[i + 1; j)$ from the dequeue iterator do not yet exist. Therefore, the cells must be either `SMARTLY-CANCELLED` or `CLOSED`, so each of them must own as a resource for resumer an awakening permit if it is inside the deque front. All these cells must be inside the deque front, as for the current call to `resume(v)` to finish, it must arrive at a cell j or later. Thus, the permits $[i + 1; j)$ from the dequeue iterator are generated and exchanged for the awakening permits needed for the invariants of the iterator. The i'th permit is also exchanged for an awakening permit, which is then used to retry the whole `resume(v)` by this caller.

`suspend()` First, a step of the enqueue iterator is performed. It can not fail: otherwise, the caller would obtain the i'th permit from the enqueue iterator along with the knowledge that the cell i is cancelled; however, this can not be, as only inhabited cells can be cancelled, and each inhabited cell owns its exclusive permit from the enqueue iterator. Therefore, the execution always obtains the i'th cell along with the i'th permit from the enqueue iterator.

A new Future is then created, along with its resumption and cancellation permits. If writing the Future to the cell is successful, then the whole resumption permit and half of the completion permit are passed to the thread queue, as is described for the corresponding transition. The remaining half of the permit is then passed alongside the Future to return to the caller of `suspend()`. If the CAS was failed, but the CAS setting the contents of the cell to `TAKEN` succeeded, then an `ImmediateResult` is created with the acquired value, and the full cancellation permit is provided to the caller. Last, if writing `TAKEN` failed, then the execution acquires a copy of E in exchange for the i'th permit from the enqueue iterator.

Cancellation Handler If the cancellation mode is "simple", then writing `CANCELLED` does not affect the execution or break any invariants; also, the canceller acquires the cancellation handle, which allows it to cancel the infinite array cell.

Otherwise, the cancellation mode is "smart", the initial state is `UNDECIDED`, and the execution owns the cancellation registration token. A call to `onCancellation()` is then made, and we consider two possibilities:

- The call was successful. The execution is provided with a cell cancelling token and a cancellation handle. An attempt is made to write `CANCELLED` to the cell in place of the Future; on success, the state transitions to `CLOSED`, and otherwise, the state was `PASSED` and stays this way. In the latter case, the execution obtains an awakening permit that is then used to call `resume(v)`.
- The call was unsuccessful. The execution is provided with a cell cancelling token and a copy of X. If an attempt to write `REFUSED` in place of the Future succeeds, X is passed to the CQS so that the resumer obtains it later; otherwise, X is kept so that it can be used to call `completeRefusedResume(..)`.

E.6.9 Observable Behavior

There are a few useful properties that can be derived from the specification of the CQS.

Our main claim is that each successful call to `resume(v)` either completes exactly one Future or performs one call to `completeRefusedResume(..)` if there are no Futures left in the queue, and each unsuccessful call completes no Futures and does not perform a call to `completeRefusedResume(..)`.

To see this, observe that the only cases when a cell does not own an E are the following:

- `resume(v)` finished its execution by writing a value to the cell (in which case the corresponding call to `suspend()` observes it and completes its Future)
- The Future was successfully resumed by `resume(v)`;
- The Future was cancelled in the simple cancellation mode (and the call to `resume(v)` fails);
- In smart cancellation mode, the cancellation handler succeeded in rewriting the Future contained in the cell with a marker and took the E (in which case either the marker is `CANCELLED` and `resume(v)` attempts to work with another cell, or the marker is `REFUSED` and `completeRefusedResume(..)` is called with the knowledge

that `onCancellation(..)` returned `false`, which means that cancellation registration failed, which can only happen due to the CQS being empty);

- In smart async cancellation mode, the resumer managed to write its value to the cell before the cancellation handler wrote a marker; this case is identical to the previous one, except that the cancellation handler does the described actions on behalf of the `resume(v)`.

Since each cell only has a single E associated with it, the fact that a successful call to `resume(v)` obtains an E means that one of the aforementioned situations must have happened.

The converse is also true: both each call to `completeRefusedResume(..)` and each completion of a Future was due to a call to `resume(v)`. Observe that each call to `resume(v)` only provides a single R. Each Future requires an R to complete, and obtaining the resource X also requires some cancellation registration to have failed, in which case the caller is provided with an instance of R. For these operations to be balanced, it is required that there are no more total calls to `completeRefusedResume(..)` and completed Future than there were successful calls to `resume(v)`.

F Formal Specifications and Proofs for the Presented Primitives

In this section we outline the proofs for the presented primitives on top of `CancellableQueueSynchronizer` in a way similar to Section E. The proofs are presented in the same model as the one described in Subsection E.1, and are defined in terms of the logical model of futures described in Subsection E.2.

F.1 The Barrier Correctness

See file `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/barrier/barrier.v`.

Specification. When the barrier is initialized to wait for k parties, it produces k logical resources called *entry permits*. The `arrive()` operation consumes an entry permit and returns a *brokenness-evidence*-passing Future. *Brokenness evidence* is knowledge of the fact that the barrier is broken; it can be freely duplicated and is mutually exclusive with entry permits, meaning that it is impossible for an entry permit to exist when it is already known that the barrier is broken.

Invariants. The `remaining` counter stores some value r. We define $n = \min(\texttt{parties} - r, \texttt{parties})$, roughly corresponding to the number of parties yet to arrive.

If n is less than `parties`, then $r > 0$, and we say that the barrier has not yet been broken. The following statements hold in this case:

- at most r entry permits still exist, and the brokenness evidence does not exist yet;
- the size of the CQS is n.

If n equals to `parties`, then $r \leq 0$ — we say that the barrier has been broken in this case, and the following is true:

- no entry permits exist, and the brokenness evidence if free to create;
- the size of the CQS is 0.

The barrier stores a CQS with the unit resource as the enqueue resource E the brokenness evidence as the dequeue resource R.

Initialization. Initially, all the invariants are satisfied: either `parties` is positive, in which case `parties` entry permits are allocated and given to the user, or it is non-positive, meaning that the barrier was created broken.

The `arrive()` Operation. The method is called with an entry permit. This means that when the `Fetch-And-Add` at line 5 is performed, the barrier cannot not broken, so $n < \texttt{parties}$ and $r > 0$. This way, we can consider the following two cases:

$r > 1$: This means that we are not the last party to be arrived. The invariant is preserved with $n' = n + 1 < \texttt{parties}$. The entry permit with which the call to `arrive()` was performed is destroyed, and the enqueue registration is performed, appending to the CQS and providing the suspension permit to the caller. After the `Fetch-And-Add` at line 5, the suspension permit is used to suspend in the CQS at line 6.

$r = 1$: We are the last party. Thus, $n = \texttt{parties} - 1$ and there are n waiters in the CQS. The invariant is preserved with $n' = \texttt{parties}$. Now that all the entry permits are destroyed, it is possible to construct the brokenness evidence. Since it is freely duplicable, we create `parties` copies of it and use them to perform $\texttt{parties} - 1$ dequeue registrations, acquiring $\texttt{parties} - 1$ awakening permits.

The awakening permits are then used to perform $\texttt{parties} - 1$ calls to `resume(..)` at line 7, which is guaranteed not to fail with the chosen modes of the CQS.

Observable Behavior. The provided specification ensures that the implemented barrier is correct: as long as at least one entry permit exists, it is impossible for any of the `Futures` to complete. On the other hand, if any of the `Futures` are completed, no entry permits exist anymore.

F.2 The Count-Down-Latch Correctness

See file `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/countdownlatch/countdownlatch.v`.

Specification. The state of the latch is represented as a natural number logically equal to the number of `countDown()` invocations until the count reaches zero. When the latch is initialized, the user obtains control of this state. Once the state reaches zero, it no longer changes.

The `await()` operation returns an *openness-evidence*-passing `Future`. The openness evidence is a logical resource that signifies that the state of the latch has become zero. It contradicts any non-zero latch state and can be freely acquired from the zero state.

The `countDown()` operation atomically decrements the latch state if it was positive, and keeps it 0 otherwise.

Invariants. The state of the latch is $\max(\texttt{count}, 0)$. The latch operates in three phases, each with its own set of invariants:

- **`Closed Latch`.** When the state is non-zero, the `waiters` counter does not have `DONE_BIT` set and stores the current size of the CQS.
- **`Open Latch`.** The state is 0, but the `waiters` counter does not have the `DONE_BIT` yet, and still stores the size of the CQS. The notable difference from the previous state is that the latch is already logically open, so new calls to `await()` will immediatelly return a completed `Future`. However, both concurrent requests can add themselves to the CQS, and the waiters are eligible to abort.
- **`Finish`**. The state is 0, and `waiters` has `DONE_BIT` set. In this case, the CQS is empty.

It is impossible to go from `Finish` to any other phase, and from `Open Latch` to `Closed Latch`. The latch stores a CQS with the unit resource as the enqueue resource E and the openness evidence as the dequeue resource R.

Initialization The invariants hold initially. `waiters` is initialized with 0 (see line 7). If `initCount` is positive, then the latch is closed, and its initial state equals `initCount`; otherwise, it is already open and its state is 0. In both cases, the CQS is empty.

The `resumeWaiters()` Operation. This method is called only when the openness evidence exists, so the latch is not closed.

First, the `waiters` counter is checked to see whether the current phase is already `Finish`, by checking for `DONE_BIT` at line 26. In this case, the CQS is empty and there is nothing to do.

Otherwise, if the phase is not `Finish`, it must be `Open`, given that the openness evidence exists. The operation then attempts to set `DONE_BIT`, performing the phase transition to `Finish` (line 28). The corresponding CAS may fail due to a concurrent `DONE_BIT` setting or the `waiters` counter increment or decrement; in this case, the operation restarts. Otherwise, if the CAS succeeds, then enough copies of the openness evidence are created to perform dequeue registration for each waiter in the CQS (line 29).

The `countDown()` Operation. First, a decrement via `Fetch-And-Add` is performed (line 10). If the latch was already in `Open Latch` or `Finish` phase, this action does not have any effect, as the state was zero and stays zero when further decrementing `count`; in this case we simply obtain the freely-duplicable evidence that the latch is open.

However, if the latch was closed at the point of decrement, there are two possibilities:

- The state was larger than 1. This subtraction logically decrements the state of the latch, but has no further effect, as all the invariants still hold.
- The state equaled 1. This means that when the `Fetch-And-Add` was performed, the latch entered the `Open Latch` phase. Thus, we perform the corresponding phase transition and obtain the knowledge that the latch is closed.

Observe that if the latch is closed at the end of the operation, it means that the state was either 0 or 1, which, in turn, shows that `count` was ≤ 1. In this case, it is valid to call `resumeWaiters()` (line 12).

The `await()` Operation. Initially, `count` is checked; if the state turns out to be 0, the Future is completed immediately. This is not just done as an optimization but has an observable effect in cases where the latch was initialized with its state being 0: if no call to `countDown()` occurs, the Futures will not ever be completed despite the latch being open.

If the state was not 0, this means that the latch was initialized with a non-zero state, which ensures that it is safe to suspend in its CQS: when the latch is opened, some of the calls to `countDown()` will eventually complete the Futures.

After that, `Fetch-And-Add` is invoked (line 17). Its effect depends on whether the current phase is `Finish`. If it is, then the `Fetch-And-Add` has no effect: since `DONE_BIT` is already set, the CQS can only be empty from now on; in this case, the call observes that the bit was set and completes its Future. Otherwise, the `Fetch-And-Add` performs enqueue registration; `suspend()` is then called with the resulting awakening permit (line 21).

Cancellation. There are two modes of cancellation described. Here we show the correctness of both of them:

- **Simple.** Cancellation of a Future does not change the size of the CQS in this mode. Thus, when a call to `resumeWaiters()` succeeds in setting the `DONE_BIT`, the number of waiters that it receives includes both alive and cancelled ones. `resume()` is repeatedly invoked, and it may fail for cancelled cells, but this does not matter: the goal here is to complete all the existing Futures, not a set number of them.
- **Smart**. Given the modes of operation of the CQS, it suffices to show that `onCancellation()` and the trivial version of `completeRefusedResume(..)` are correct.

  `onCancellation()` performs a `Fetch-And-Add` (line 34). If the phase was already `Finish`, then this operation has no effect, as the CQS is empty and the `DONE_BIT` is set, in which case cancellation registration fails and the call returns `false` accordingly. Otherwise, the `Fetch-And-Add` decrements the size of the CQS, and the cancellation registration succeeds.

  `completeRefusedResume(..)` is valid since if the CQS is empty, it is safe to simply quit the resumption operation: no valuable resources are being passed that could be lost otherwise.

Observable Behavior. The specification ensures that at least `initCount` calls are needed for any of the Futures to complete.

F.3 The Semaphore Correctness

See files in `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/interruptible_semaphore`, depending on the CQS mode used.

Specification. The semaphore does not introduce any logical resources on its own; instead, it is parameterized by the type of the resource T that is needed to perform operations in the critical section guarded by the semaphore. Thus, in order to call `release()` in a valid manner, the caller must provide an instance of T, and `acquire()` returns a T-passing Future. Additionally, in order to create a semaphore that initially stores K, then K copies of T must be provided during initialization. This specification is inspired by the way to express the mutex property for locks that is commonly used in the example code for Iris.

Invariants. There are two possibilities for the state of the semaphore:

- The CQS is empty. In this case, `permits` stores some number n, and the semaphore owns n copies of T.
- The size of the CQS is $m > 0$. Then `permits` stores $-m$.

The semaphore stores a CQS with the unit resource as the enqueue resource E and T as the dequeue resource R.

Initialization. The invariants hold initially. The CQS is empty, and the semaphore owns K copies of T (which are required in order to initialize it), which is reflected in `permits` storing K.

The `acquire()` Operation. First, a `Fetch-And-Add` is performed (line 9). We consider two possibilities for the initial state of `permits`: if it contained a number greater than zero, then the semaphore owned at least one copy of T, which is taken by the call and put into an immediately complete Future (line 11). Otherwise, the semaphore did not own a single copy of T, and the call to `Fetch-And-Add` is used to perform the enqueue registration needed to call `suspend()` (line 13).

The `release()` Operation. First, a `Fetch-And-Add` is performed (line 16). If the CQS was empty, then `permits` contained a nonnegative number, in which case the operation simply passes its copy of T to the semaphore and finishes. Otherwise, `permits` contained a negative number, and the `Fetch-And-Add` performs a dequeue registration, the awakening permit from which is used to call `resume()` at line 18, which never fails given the chosen mode of the CQS.

Cancellation. With the chosen modes of the CQS, it suffices to show that `onCancellation()` and the trivial version of `completeRefusedResume(..)` are correct. The `onCancellation()` function performs a `Fetch-And-Add` at line 21, attempting to perform a cancellation registration. If the CQS was empty (and the value in `permits` is negative), then the T that is obtained as part of the cancellation registration is passed to the semaphore, like it is done in `release()`. If the CQS did contain other Futures, the size of the CQS is decremented, successfully registering the cancellation. The `completeRefusedResume(..)` invocation is valid since the call to `onCancellation()` has already provided the semaphore with T.

Observable Behavior. Choosing T to be some resource that only exists in n copies, we can ensure that the provided implementation is indeed a semaphore: if a block of code is guarded by awaiting on a Future returned from `acquire()`, it would be impossible for this code to be executed concurrently by more than n threads.

F.4 The Blocking Pools Correctness

See file `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/blocking_pool/pool.v`.

The idea of the blocking pool algorithm is similar to the one for semaphore, with the difference that pools transfer actual values instead of logical permits.

In this proof, the concurrent data structure chosen for passing values outside of the CQS is called the *outer storage* of the pool ("outer" refers to the activity happening outside of the CQS).

Specification. The pool does not use any user-visible logical resources and is parametrized by a predicate $U(..)$ that describes the values passed in the pool. A call to `put(e)` requires providing an instance of $U(e)$, and `take()` returns a $U(v)$-passing Future if it completes with a v.

Note the similarities between the semaphore specification and this one; this is explained by the fact that a semaphore can be thought of as a pool of unit values with an optimization that allows the implementation to only keep a counter of available permits instead of actually storing them in some concurrent data structure.

Requirements for the Outer Storage. (see file `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/blocking_pool/outer_storage_spec.v`) The outer storage must expose a state that is represented by a multiset of values that it contains and the number of failed retrieval attempts that are yet to be balanced by the corresponding insertions. Then, `tryInsert(..)` must atomically add its value to the multiset, returning `true`, or decrease the number of failed retrievals, returning `false`. `tryRetrieve()`, likewise, must atomically extract some value from the multiset and return it, or increase the number of failed retrievals, returning `null`.

It can be easily shown that the proposed data structures fulfill these requirements (see files `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/blocking_pool/stack_outer_storage.v`, `https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs/theories/lib/blocking_pool/queue_outer_storage.v`). For example, if the head of the stack is $\bot$, then it contains only failed retrievals; otherwise, the contents of the stack form the multiset of the contained elements. `tryInsertStack(..)` performs a single successful `CAS` during its operation, either observing that there are no failed retrievals and placing its value in the stack or deregistering one of the retrievals. `tryRetrieveStack()` behaves in a symmetrical manner. We say that the *size* of the outer storage is $n - m$ if n elements are stored in the multiset and there are m failed retrievals.

Invariants. The implementation uses internally some logical resources that are not exposed to the user of the pool, namely *insertion permits* and *retrieval permits*. We say that k_i is the number of the insertion permits in existence, and k_r is the number of retrieval permits. If the current size of the outer storage is s_o, it is known that $s_o + k_i \geq k_r$; in simple words, there can not be more retrieval permits than there would be elements in the outer storage if all insertion attempts succeeded. `size` stores $s_o - s_t + k_i - k_r$, where s_t is the current CQS size. We know that either s_t is zero or $s_o + k_i - k_r$ is, so either the CQS or the outer storage is effectively empty.

For each instance of a value v stored in the multiset representation of the outer storage, the pool owns a copy of $U(v)$. The pool stores an CQS with the unit resource as the enqueue resource E, $U(v)$ for element each element v passed through it as the dequeue resource R, and a pair of the insertion permit and $U(v)$ as X, the resource required to call `completeRefusedResume(..)`.

Initialization. Initially the invariants hold. Both the CQS and the outer storage are empty, no insertion or retrieval permits exist, and `size` is zero (line 5).

The `take()` Operation. First, `Fetch-And-Add` is performed at line 18. We consider two possibilities for the initial state of `size`.

- If it contained a number greater than zero, then $s_o + k_i - k_r$ is positive. A new retrieval permit is created, incrementing k_r. With this permit, a call to `tryRetrieve()` is attempted (line 22); regardless of whether it succeeds, both s_o and k_r are decremented: the retrieval permit is destroyed in the attempt, and either a value is taken from the logical multiset or a new failed retrieval is registered.
- If it contained a nonpositive number, then $s_o + k_i - k_r$ is zero. s_t is incremented via enqueue registration, and the resulting suspension permit is used to place a Future in the CQS (line 25).

The `put()` Operation. `Fetch-And-Add` is performed (line 8). Like with `take()`, we consider two possibilities.

- If `size` contained a nonnegative number, the CQS is empty. In this case, a new insertion permit is generated, which increments k_i. A call to `tryInsert()` is performed at line 15. The permit is destroyed, and either s_o increases because a new value was successfully placed in the multiset, or it increases because a failed retrieval was removed.
- Otherwise, `size` contained a negative number, which means that $s_t > 0$. A dequeue registration is performed, and the awakening permit is then used to call `resume(..)` (line 11).

Cancellation. With the given mode of the CQS, we show that `onCancellation()` and `completeRefusedResume(..)` are correct. The `onCancellation()` operation performs a `Fetch-And-Add`, attempting to perform a cancellation registration, at line 29. If the CQS was empty (and the value in `size` is negative), then a new insertion permit is generated, and the cancellation procedure obtains an X using the $U(v)$ provided to it, where v is the value of the resumer of this cell (possibly undecided at the moment). If the CQS did contain other Futures, the size of the CQS is decremented, successfully registering the cancellation.

The `completeRefusedResume(..)` implementation is valid since $U(v)$ and the insertion permit, both of which are needed to perform `tryInsert(..)`, are present as part of X. If `tryInsert(..)` fails, $U(v)$ is taken back and can be used in a call to `put(..)`.