Project Report: Cross-Platform Gears

Maximilian Müller

Jan 05, 2024

1. Introduction

Gears is a Scala library for direct-style concurrency and asynchronous programming. Starting from a solid foundation [1], [2], this library was restructured to run on multiple platforms (JVM and Native, so far). Additionally, interfaces of basic components (like listeners) were updated to reconcile core principles like generality of sources and atomicity of race.

This report first presents the current status of the library as a whole in Section 2. Then it focusses on the main areas of this semester's work. The platform abstraction is described in Section 3. Then, Section 4 contains a detailed explanation of the new `Listener` interface that combines the requirements of channels and `race`. Finally, Section 5 covers some benchmarks that were used to assess implementations and identify the best option.

2. Library Overview

This section presents the current status of Gears. Specifically it focuses on all Gears artifacts that played a significant role for the semester project. We start with some core concepts and definitions which are used throughout the report. Following a short introduction of the new `AsyncSupport` layer, the `Source` abstraction and the most prominent example, the `Future`, are described in Section 2.1. Then, Section 2.2 introduces channels and their exposed `Sources`. After those elementary `Sources`, Section 2.3 presents some operations to create derived `Sources`. Finally, the cross-cutting aspect of cancellation is briefly highlighted in Section 2.4 with a strong focus on new usages of the concepts.

First and foremost, the following concepts are at the heart of Gears [1]:

Source an asynchronous object whose result can be awaited

Listener a handler to retrieve a single object from a `Source`

Future a specific type of `Source` that, once completed, yields its result forever whenever asked

Async the capability to spawn concurrent computations (=: *AsyncSpawn*) and await asynchronous results (=: *AsyncAwait*)

Asynchronous Operation a function that requires the `Async` capability, expressed by taking an `Async` context parameter

Around those core principles, Gears is composed of multiple components which make up the functionality of the library. The very foundation of concurrency and asynchronicity is the `AsyncSupport` layer (see Section 3). It must be provided to Gears at the entry point where a user has the choice to use a default, configure it, or provide an entirely different implementation. This layer consists of a scheduler providing concurrency and a delimited continuation implementation providing direct-style asynchronous capabilities, both of which are platform-dependent. While the scheduler exposes methods to execute and schedule (now or later), the continuation support closely follows the Suspension API described in [3].

2.1. Sources, Futures, and Listeners

Those two APIs are used to implement the `RunnableFuture`. It is a specific `Future` that describes a concurrent computation which is started as a top-level suspension boundary in a task submitted to the scheduler. The `Async` capability for its body is composed of the parent *AsyncSpawn* capability (necessary to spawn the `RunnableFuture`) and its own implementation of *AsyncAwait*. For

AsyncSpawn, it combines the parent's scheduler with a new `CompletionGroup` which defines the scope of the `Future`'s body. The *AsyncAwait* wraps the parent's `AsyncSupport` layer and uses it to suspend to the top-level boundary.

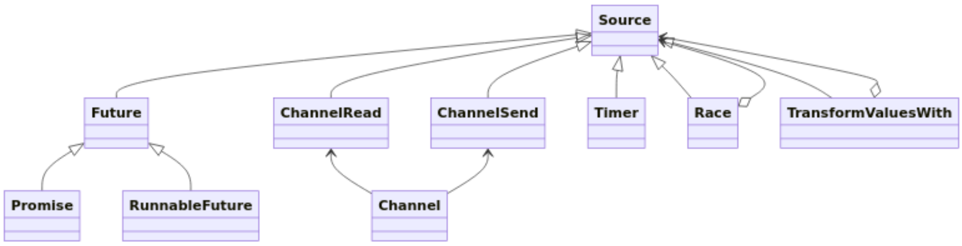

FIGURE 1. Hierarchy of Sources

Besides `RunnableFuture`s, there are many other `Source`s which are depicted in Figure 1. The result of a `Source` is exposed through a synchronous (`poll`) and an asynchronous (`onComplete`) interface. Both methods take a `Listener` parameter and try to call it with an element if available. If none is available, `poll` indicates this by returning `false`, while `onComplete` keeps the `Listener` in an internal queue to complete it asynchronously later, once data is available. A `Listener` is only completed once and dropped afterwards. It can also be dropped explicitly using `dropListener`, when it is no longer interested in the data. The *AsyncAwait* capability builds on this by, before suspending, registering a `Listener` that schedules the continuation with the obtained data. The full interface of `Source` is shown in Listing 1. The only difference to [1] is the `awaitResult` naming, and the return value `true` from `poll` if an element is available that is rejected by the `Listener`.

```
trait Source[+T]:
  def poll(k: Listener[T]): Boolean
  def onComplete(k: Listener[T]): Unit
  def dropListener(k: Listener[T]): Unit

  /** Utility method for direct polling. */
  def poll(): Option[T] = ...

  /** Utility method for direct waiting with `Async`. */
  final def awaitResult(using ac: Async) = ac.await(this)
```

LISTING 1. Trait Async.Source

A second `Future` implementation, the `Promise`, is available. It is not completed automatically using the result of a code block (as is the `RunnableFuture`), but manually by the user. There are two variants: `Promise.apply` creates a detached `Promise` instance that can be stored and completed anywhere for maximum flexibility. Meanwhile, the new `Future.withResolver` provides a utility for the common case of an external asynchronous computation. To start the computation, a handler is passed to `withResolver` which calls that handler with a `Resolver` handle. The handle allows to `resolve` or `reject` similarly to Javascript's `Promise` constructor [4].

2.2. Channels

`Channel` operations are `Source`s as well. A `Channel` is a data flow and synchronization primitive that allows message passing with a `send`/`read` (*receive*) interface. There exist three variants: First, a `SyncChannel` only permits communication by *rendezvous* between a sender and a receiver. Second, a `BufferedChannel` has an internal buffer of limited size, so that a `read` can take an element from a non-empty buffer and `send` append to a non-full buffer without delay. Third, the `UnboundedChannel` is a special case of the `BufferedChannel` without an upper bound on buffered elements. In particular, the `send` operation always succeeds immediately.

In general, this is not the case, as `send` and `read` might have to wait for a `rendezvous` or buffer space availability. Therefore, those operations are *asynchronous operations* which are exposed as such (e.g., `def send(x: T)(using Async): Unit`). But in addition, they are also available as `Sources` to allow for composition. In contrast to `Futures`, those are passive `Sources` that only act on behalf of an attached `Listener`. On the reading end, each `Channel` consequently has one `readSource`. For each `Listener` that is attached (through `poll` or `onComplete`), the `Channel` attempts to read an element (from the buffer or a parallel `send`) to pass it to the `Listener`. To send through the `Source`-interface, a `Source` is created for a given element. For each attached `Listener`, the `Channel` attempts to send that same element and passes `Unit` to the `Listener` as success indicator.

A `Channel` may also be closed to indicate to readers that no more elements will be sent. This possible state is represented by a result type of `Either[Closed, *]` for both types of `Channel` `Sources` and the suspending `read` operation. As the sender is usually the one to close the `Channel`, the suspending `send` does not expect the `Channel` to be closed. In case it is closed, however, send throws. The full interface is shown in Listing 2. The changes to [2] are, besides naming and code organization, the `sendSource`, moving from `Try` to `Either` for closed channels, and the new `UnboundedChannel`.

```
object Channel:
  case object Closed

trait SendableChannel[-T]:
  def sendSource(x: T): Async.Source[Either[Channel.Closed, Unit]]
  def send(x: T)(using Async): Unit = ...

trait ReadableChannel[+T]:
  val readSource: Async.Source[Either[Channel.Closed, T]]
  def read()(using Async): Either[Channel.Closed, T] = ...

trait Channel[T] extends SendableChannel[T], ReadableChannel[T],
java.io.Closeable
```

Listing 2. Trait Channel

Another elementary `Source`, which exists primarily as an example, is the `Timer`. It is a `Source` that emits ticks in a given interval of time. To do so, it must be started synchronously on a thread or fiber where it loops until external cancellation, alternating between sleeping and sending a tick to all subscribed `Listeners`.

2.3. Derived Sources

In addition to those elementary `Sources`, there also exist *derived* `Sources` that compose one or multiple others. The simplest derived `Source[T]` is a `transformValuesWith` (a monad's *map*) that transforms elements of one upstream `Source[U]` one-by-one using a simple function `U => T`. It forwards requests (poll/onComplete/dropListener) to its upstream by wrapping any incoming `Listener[T]` in a *derived* `Listener[U]`. An extract of this is shown in Listing 3.

The method `transform` is used to create a derived `Listener` from a `Listener` `k` that is given to the derived `Source`. This derived `Listener` is then handed over to the upstream `Source` as proxy for completion: Once the upstream `Source` completes that derived `Listener`, it forwards the `complete` call to the downstream `Listener` `k`.

```
// wrap the listener k attached to derivedSource in a derived listener
def transform(k: Listener[U]) =
  new Listener.ForwardingListener[T](derivedSource, k):
    def complete(data: T, source: Async.Source[T]) =
      k.complete(func(data), derivedSource)

// the source interface is implemented in terms of [[transform]],
// onComplete and dropListener similar
def poll(k: Listener[U]) = upstream.poll(transform(k))
```

Listing 3. Implementation transformValuesWith

This two-way derivation is a general principle, that becomes more powerful as multiple `Sources` are aggregated to one. It is depicted in Figure 2. Each derived `Source` creates a derived `Listener` for each `Listener` that is attached to it (denoted by the curly arrow). Each derived `Source` holds a reference (simple arrow) to its upstream of which it is composed (in image below), whereas each derived `Listener` holds a reference to its downstream to forward completion as well as to the derived `Source` that created it (same color). This is called the *lineage* of a `Listener` [1]. The `Source` reference is used for mainly two things: First, to tell the downstream `Listener` from which `Source` it was completed (which, from that `Listener`'s perspective is the derived `Source`). Second, the `derivedSource` is passed to the `ForwardingListener` constructor. This ensures that equal `Listeners` on derived `Sources` always result in equal derived `Listeners`, which is fundamental to `Listener` dropping.

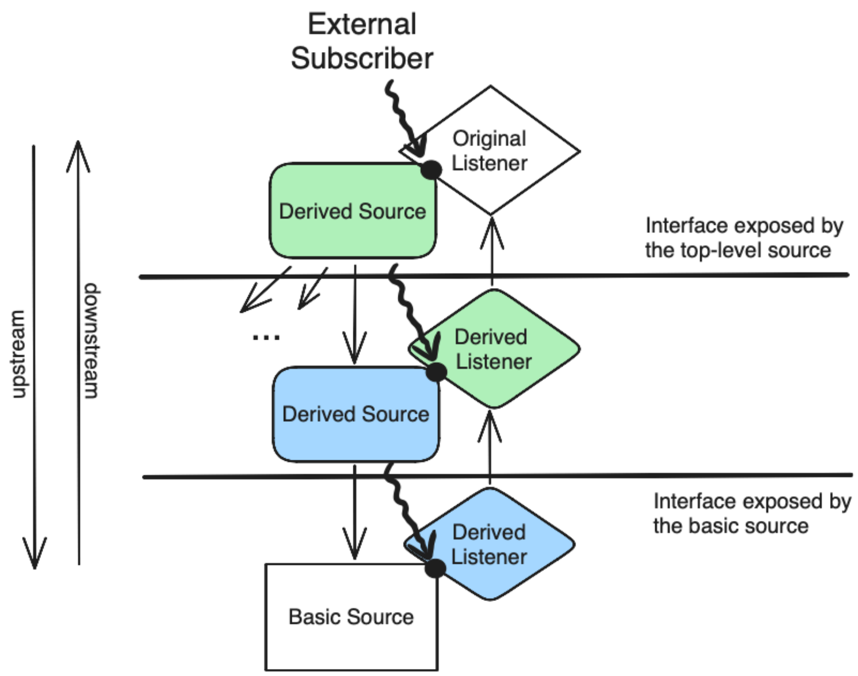

Figure 2. Multi-Level Derived Sources

The most important aggregating `Source` is `race`. It takes a sequence of upstream `Sources` and forwards each `Listener` that is attached to it to all of those `Sources`. As soon as the first upstream completes the `Listener`, the `race` unsubscribes automatically from the other upstreams. Beyond guaranteeing that the `Listener` is always only invoked once (`Listener` contract), it also tells a `Source` in advance whether its item is taken or not. The underlying mechanism is described in Section 4.

In addition to the simple `race` that combines `Source[T]` instances to a single `Source[T]`, there also exists a new `raceWithOrigin` of type `Source[(T, Source[T])]` that annotates the item itself with the source which it was provided from. This is used to implement the higher-level `select` construct. It implements the common use case where multiple `Sources` (possibly of different types) are awaited, but the code to run afterwards depends on which `Source` yielded the result. The `select` therefore operates on a sequence of `Source`/handler tuples which are represented as `opaque type` alias (`SelectCase[T] = (Source[?], Nothing => T)`) to guarantee type correctness in the absence of full dependent object types. A special constructor for those values of the form `[U, T] => (Source[U], U => T) => SelectCase[T]` discards the parameter `U`.

2.4. Cancellation

Gears adheres to the principle of structured concurrency by default. This means that all asynchronous operations which are started in a scope terminate before the scope ends. This is ensured by tracking every asynchronously started operation in a `CompletionGroup`, so that it can be cancelled and awaited at the end of a scope. Further, every long-running operation that can be performed asynchronously is tracked so that it can be cancelled (a `Cancellable`). Every `Async` capability is bound to a `CompletionGroup`. A `CompletionGroup` is `Cancellable` itself, forwarding the cancellation request to its members, and it can be awaited, suspending until all members have `unlinked` themselves [1]. The await is not exposed publicly as it could easily be used for a scope to await itself. Instead the scope logic is available as `Async.group` which can be wrapped by user functions.

There are two long-running primitives as of now, `await` and `sleep` (on Native, only `await` is primitive), and one asynchronous primitive, `RunnableFuture`. In particular, the top-level `Async.blocking` is not cancellable, nor is an `await` performed with the top-level capability. To support cancellation in `RunnableFutures`, the `await` method of the `FutureAsync` implementation registers a `Cancellable` to the internal group of the `RunnableFuture` (the one of its body scope, not the group the `Future` is linked to). When this `Cancellable` is cancelled, the `Listener` is dropped from the awaited `Source` and a `CancellationException` is thrown.

The JVM `sleep` implementation works similarly. It registers a `Cancellable` to the group of the passed `Async` capability before sleeping. This `Cancellable` captures the (virtual) thread instance to interrupt it on cancel request. The `InterruptedException` is catched and a `CancellationException` thrown instead.

This requires advanced `CompletionGroup` support. Whereas the `RunnableFuture.await` can check the `Future`'s internal cancellation status before starting the operation, `sleep` cannot. Therefore, `CompletionGroups` were adapted to persist cancellation. With this change, every `Cancellable` that is `linked` to the group, after it had been cancelled, is cancelled immediately as well.

Finally, `Promises` created using `withResolver` support cancellation as well. The body can register a cancellation handler with the resolver using `onCancel`. The default behavior is to complete the `Promise` immediately with a `CancellationException`. Any `Future`, as soon as it is `completed` (implicitly at the end of the `RunnableFuture` body or explicitly), `unlinks` itself from its `CompletionGroup`.

3. Platform abstraction: The 'Support' Layer

While the previous version of Gears was implemented on top of JVM threads and object monitors [5], the library now supports Scala Native as well. To allow for this compatibility, a new abstraction is introduced. It encapsulates the scheduling and suspending capabilities which must be provided by the platform to allow for `RunnableFutures` and `awaiting`.

The structure of this section is as follows: It starts with the theory of `await` and the interface of the generic 'Support' layer. Section 3.1 introduces the implementation of this interface on top of JVM virtual threads. Afterwards, the optimization for an efficient implementation of `await`, the key point of this layer, is presented in Section 3.2. Section 3.3 concludes with remaining cross-platform issues, such as `Scheduler` and `sleep`, and how this layer is exposed.

As noted in [1] ("Implementing Await"), `await` can be done in two ways. A straightforward solution is possible using *delimited continuations*, previously introduced for Scala in [3]. In this approach, an `await` is translated into a `suspend` that yields to a boundary (in this case, the scheduler). In the user handler that is run after the `suspend` set up a `Suspension` instance, a `Listener` wrapping that `Suspension` is created and attached to the awaited `Source`. The other option is to implement `await` using *fibers*, as provided by Project Loom [6]. Here, `await` simply uses some locking/waiting mechanism to yield to the fiber runtime.

This abstraction sticks to the former approach, as both `await` is easier to implement in terms of continuations that on fibers, and continuations are easier to implement in terms of fibers than vice-versa. The resulting interface is given in Listing 4.

```
trait Suspension[-T, +R]:
  def resume(arg: T): R

trait SuspendSupport:
  type Label[R]
  type Suspension[-T, +R] <: gears.async.Suspension[T, R]

  def boundary[R](body: Label[R] ?=> R): R
  def suspend[T, R](body: Suspension[T, R] => R)(using Label[R]): T
```

Listing 4. Support Layer: Suspend Support

3.1. Support Implementation

The Scala Native implementation wraps the new platform-provided API with the same structure [7]. On JVM, fibers, i.e., virtual threads, are employed to emulate suspensions. To create a `boundary`, a new virtual thread is started to run the body. The `Label` instance is used to communicate the final or intermediate result `R` from the virtual thread to the code outside the `boundary`. It contains an `Option[R]` field to hold the value and a lock with a condition variable to do synchronization and waiting. It is awaited for the initial result after the `boundary` thread was started. When the `boundary` body `suspends`, the fresh `Suspension` instance is used equivalently to store the next input given from outside to be returned by `suspend`. An extract is given in Listing 5.

```
override def suspend[T, R](body: Suspension[T, R] => R)
                          (using label: Label[R]): T =
  val suspension = new VThreadSuspension[T, R]()
  val result = body(suspension)
  label.setResult(result)
  suspension.waitInput()

class VThreadSuspension[-T, +R](using val label: Label[R]): ...
  override def resume(input: T): R =
    label.clearResult()
    this.setInput(input)
    label.waitResult()
```

Listing 5. Virtual Thread-based Suspensions

Note that it is important to use condition variables instead of object monitors (`synchronized`, `wait`/`notify`) because the latter operations make the virtual thread "pinned to its carrier" [8].

In addition to the delimited continuation support, a `Scheduler` is necessary, not only to spawn `RunnableFuture`s, but also to continue asynchronously after `await`. Our proposed `Scheduler` contains a method to submit a `Runnable` for immediate execution (`execute`) and a method to submit a `Runnable` to be run after a given delay (`schedule`). The latter returns a handle which can be used to optimistically cancel the operation before it starts.

3.2. Async Implementation and Joint Operations

Given those two things, a `Suspension` implementation and a `Scheduler`, it is possible to construct an `Async` capability, i.e., to implement `RunnableFuture`s that can await asynchronous `Source`s. The two common operations can then be implemented as follows [1]: To spawn, we compose `execute` with `boundary` to submit a task to the `Scheduler`. This task will finish once the body first `suspends` (or completes). To await a `Source`, the awaiting body `suspends` and wraps the `Suspension` in a `Listener` that is registered to the `Source`. This `Listener` will, on completion, `resume` the `Suspension` but it should not run on the `Listener`-completing thread. Instead it submits the continuation of the `Suspension` to the `Scheduler`, composing `execute` with `resume`.

While this is how it works (and should work) on Scala Native, it introduces some inefficient overhead on the JVM as scheduling and delimited continuations both use virtual threads. To account for that, two joint methods, `scheduleBoundary` (replacing `execute(boundary(...))`) and `resumeAsync` (replacing `execute(resume(...))`), are introduced. Both have exactly the same effect and even implemented as a simple forward on Native.

On the JVM, `boundary` spawns a virtual thread to run the body and awaits some intermediate or final result of that body. In combination with `execute`, this spawns a thread (`execute`) that spawns another thread (`boundary`) and waits for an intermediate/final result. Similarly, `resume` tells the `Suspension`, that sleeps on another virtual thread, to go on (`setInput` in Listing 5) and then waits for the next result (`waitResult`). Again, the thread that would be spawned by `execute` would only send a message and wait. In both cases, this virtual thread for `execute` can be saved by removing the waiting part from `boundary` and `resume`, respectively.

```scala
// AsyncSupport.scala
trait AsyncSupport extends SuspendSupport:
  type Scheduler <: gears.async.Scheduler

  private[async] def resumeAsync[T, R](suspension: Suspension[T, R])
                          (input: T)(using sched: Scheduler): Unit =
    sched.execute(() => suspension.resume(input))

  private[async] def scheduleBoundary(body: Label[Unit] ?=> Unit)
                                   (using sched: Scheduler): Unit =
    sched.execute(() => boundary(body))

// VThreadSupport.scala
object VThreadSupport extends AsyncSupport: ...
  override private[async] def resumeAsync[T, R]
          (suspension: Suspension[T, R])(input: T)(using Scheduler): Unit =
    suspension.label.clearResult()
    suspension.setInput(input)
```

Listing 6. AsyncSupport Trait and JVM Implementation

These two joint operations are exposed in a trait `AsyncSupport` extending the `SuspendSupport`. It is linked to a `Scheduler` type by an abstract type member in `AsyncSupport`. As can be seen in Listing 6, the default implementation consists of combining the two basic operations. The implementation with virtual threads is able to refine this implementation because the definitions of `resumeAsync` and `scheduleBoundary` accept instances of the `Suspension` and `Label` type members from SuspendSupport (see Listing 4).

3.3. Scheduler, `sleep` and the Default Package

The `Scheduler` implementation for JVM is a singleton that forwards `execute` to spawning a virtual thread. The `schedule` implementation also spawns and waits the delay before running the body. On Native, the default `Scheduler` implementation takes a `scala.concurrent.ExecutionContext` which is used to spawn computations. For `schedule`ing, a single platform thread is started that keeps a priority queue of scheduled tasks. This thread is defined as a daemon thread to simplify cleanup. It loops forever, sleeping and spawning.

For both Scala Native and JVM, the default support implementation and `Scheduler` are exposed as top-level `given`s in the `gears.async.default` package (the default Native `Scheduler` is created with an unconfigured `java.util.concurrent.ForkJoinPool`). A user of the library must import them (or provide a custom instance) as context parameters when entering `Async.blocking`. There, they are captured in the `Async` capability and passed to its derived capabilities (e.g., spawned `RunnableFuture`s).

In addition to these abstractions, one basic operation is currently provided in the support layer, which is `sleep`. In general, this can be implemented with a `Source` (e.g., a `Promise`) and the `Scheduler`'s `schedule`, which is done on Native. On the other hand, the JVM allows a more efficient solution, as `Thread.sleep(...)` is compatible with virtual threads. This is encapsulated independently of the previously mentioned parts of the support layer in another trait. Again, the default implementations are given in `gears.async.default`.

4. Atomicity in Listeners

The major challenge of the design that was outlined in Section 2 is the combination of atomic `race` and channels. After an explanation of the problem, the new solution is presented in multiple steps. First, the approach to use locks is introduced and incorporated into the `Listener` interface in Section 4.1. Then, Section 4.2 describes how derived and composed Sources/Listeners are dealt with. Afterwards, we go back to channels and present the algorithm and encapsulation for locking two Listeners in parallel in Section 4.3. Finally, the complications and proposed solutions for Listener dropping in `race` are shown in Section 4.4.

While a `Listener` does not affect a `Future`'s behavior (neither when attached nor when completed) by any means, this is not the case for channel `Source`s. `Future`s are active `Source`s already running in the background. Termination depends only on that background task. When a `Future` is completed, all `Listener`s are completed with the same final value. Channel `Source`s, on the other hand, are passive and only act on behalf of a `Listener`. Attaching a `Listener` is a (receive or send) request and completion is the action itself.

At the same time, `race` wraps multiple upstream `Source`s and completes when one of these completes. It therefore only accepts a single item from one `Source` and cannot handle any other. As a `Source` may change its internal state due to a completed `Listener` (as channels do), the `Listener` must tell the `Source` whether it can handle the item. The previous solution was for a `Listener` to return a `Boolean` in its completion method, indicating whether the item was handled [1]. It was therefore defined as `trait Listener[-T] extends (T => Boolean)`. The `race` implementation could keep a flag to remember and atomically check in its derived `Listener` whether it was already completed and short-circuit every upcoming call.

4.1. Introduce Locks to Listeners[1]

The problem with this approach is that it only works with a single `Listener`, as completion and acceptance check are done atomically within a single `Listener` method. When the `Source` receives the return value, the item was already handled. But in a rendezvous channel where both receiver and sender are `Sources`, there are two ends to complete in a single message passing cycle. Both ends must signal their availability before the opposite end may actually handle the item. Locking is necessary to guarantee that.

Consequently, the `Listener` is extended with an optional (i.e., nullable) `lock` field that contains the `Listener`'s locking facility (`ListenerLock`) if it employs locking. If it is absent, the `complete` method can be called at any time. If it is non-null, the lock must be acquired before calling `complete`, which is responsible for releasing the lock internally. If the lock was acquired but the `Source` is unable to `complete`, it can `release` the lock again through the facility. Both `complete` in the `Listener` and `lockSelf` in the `ListenerLock` take a `Source` argument to know which `Source` attempts to lock/complete.

With locking, the risk of deadlocks is introduced as well. This can happen in channels where two `Listeners` (one sender, one receiver) are locked at the same time. As an example, consider `race(read(C1), send(C2))` in parallel with `race(send(C1), read(C2))` and read-before-send locking; when both channels have acquired their read end, they cannot proceed. To cope with that, we employ lock numbering. A lock number is obtained by extending the `NumberedLock` trait, which uses a global `AtomicLong` to generate ascending numbers. The obtained number is then exposed as `selfNumber` in the `ListenerLock`, so that a consistent ordering of locks can be achieved.

4.2. Locked Listener Composition

Further, `Listeners` may have a *lineage*, as `Sources` can be composed at an arbitrary depth (see Section 2.3). There are two types of derivation that needs special care, that is, when a lock-employing `Listener` is wrapped. The derivation can be lock-free, in which case the derived `Listener` still employs locking (indirectly). The `ListenerLock` facility of the derived `Listener` is a thin wrapper around the original facility with the same `selfNumber`. It forwards lock and release requests without further ado, except for replacing the `Source` argument to `lockSelf` with the derived `Source` that is known to the base `Listener`.

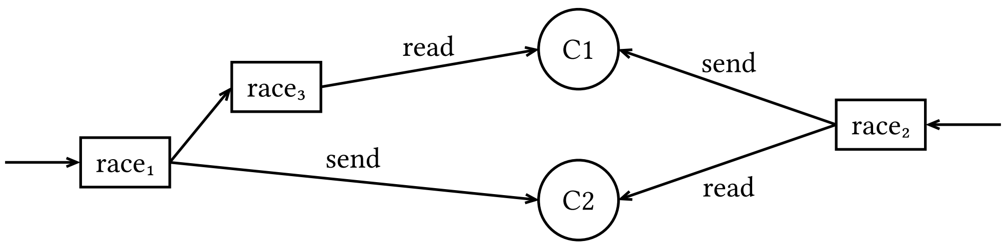

Figure 3. Nested Locked Listener

Wrapping a lock-employing `Listener` in a lock-employing derivation (e.g., a nested `race`) introduces more complexity. The `selfNumber` is the number of the derivation's lock, which is larger than the base lock because the wrapping object is instantiated after the wrapped object and the number-providing `AtomicLong` is increased on every request. As the derivation of a `Listener` may happen in

[1]The full interface is shown in Listing 7

multiple steps unatomically, lock numbers in one lineage are possibly not contiguous. To lock a `Listener`, its entire lineage of locks must be acquired.

A possible result is the scenario depicted in Figure 3 that illustrates another risk of deadlock. The arrows indicate the direction of attachment and derivation of `Listeners`, the subscripts of `race` are the lock numbers (note that $race_3$ is derived from $race_1$, thus the higher number), and the circles denote the channels. Again, both channels could decide to lock the read end first (C1 locking locks $race_1$ and $race_3$, C2 locking $race_2$), resulting in a deadlock (similarly for send first). But even looking at the lock numbers of the attached `Listener` is not enough: Locking the higher number first, this has the same outcome as read first (C1 prefers $race_3$ to $race_2$, unknowingly locking $race_1$ too; C2 prefers $race_2$ to $race_1$).

4.3. Parallel Locking

To handle this, the two `Listeners` have to be locked in parallel. Our first approach was based on delimited continuations, where a channel would set up a boundary to lock a `Listener` which is required to suspend with the lock number before any locking. With a nested-lock-first approach, implying low-to-high locking, any locking derivation is required to first lock its base `Listener` before locking itself. The channel can start the boundaries in any order. As long as both parts have locks left (yielded with `suspend`), the locking continues with the one that provides a lower lock number. As soon as one boundary terminates (successfully or not), the other one can be run to completion or be cancelled.

This approach has two main benefits: First, encapsulating the `suspend` requirement in a `LockContext`, the traditional single-`Listener` locking is barely impacted by this feature in terms of performance, as the locking operation stays a simple recursing call. This is important, as every `Source` action, including `await` involves those `Listeners`. Second, the `Listener` does not have to present the lock facility or lock numbers at all. Everything stays encapsulated inside the `Listener`, which makes the interface and `Listener`-facing code much less complex. The lock numbers only appear when passing a custom `LockContext`. This implementation can be found in the repository[2].

On the other hand, the continuations have heavy performance downsides which are discussed in Section 5.1. Therefore we adopt an interface with explicit continuation. In this approach, a locking derivation returns an intermediate result from the `lockSelf` method, as long as nested locks remain. This enables the lock requestor to switch between two locking processes. The result type of locking is thus threefold, as, in addition to the success (`Locked`) and rejection case (`Gone`), the intermediate result (`PartialLock`) can occur. This `PartialLock` is a representation of an already acquired lock and it exposes the interface of the next lock, i.e., its lock number and lock operation.

By locking in derivation-lock-first order (and thus from high to low lock numbers), this process can be done without allocation, that would otherwise be incurred by recursing and wrapping the intermediates. A locking derivation acquires its internal lock and checks its state. If it is gone (e.g., already completed), it releases the lock and returns `Gone`. If it is available, the return value depends on the downstream `Listener`. If it is lock-free, the locking is complete and `Locked` is returned. If it employs locking (directly or indirectly), a static `PartialLock` (same lifetime as `Listener` itself), that wraps the downstream's locking facility, is returned.

4.4. Cleaning up: Release and Drop

Avoiding allocation entails difficulties for release. There are four reasons to release a lock, apart from the immediate release if the lock owner rejects due to internal state: First, after the `Listener` has been locked, if an item is available - this is moved to `complete`. Second, after the `Listener` has been

[2] https://github.com/lampepfl/gears/blob/25bb9328e4c95641032ac8458d59/src/main/scala/async/Listener.scala

locked, if no item is available any more. In this case, the entire `Listener` with its lineage must be released. Third, during locking, if a nested lock facility rejects, and fourth, during a locking on hold, if another `Listener`, which is locked in parallel, rejects and the operation is cancelled.

The interesting cases are those where only a subset is acquired. Since a `PartialLock` returned from a downstream (nested) `Listener` is not wrapped by its derived `Listener`, there is no way from the `PartialLock` to reach the acquired locks. The release operation must therefore be rooted at the upstream `Listener` where the locking initially started. The information that only a subset of the locks should be released is transmitted by passing the current `PartialLock` as argument to `release` (or `Locked` if the `Listener` has been locked completely). A `Listener` (derivation) implementation can compare this `PartialLock` to its own instance and stop if they are the same. If there are acquired locks remaining, the next one is returned (instead of recursed) to be able to employ optimized tail-recursion, null otherwise.

This leaves the issue of `Listener` dropping. A `Listener` that accepted or rejected an element once is considered permanently gone. This can be respected by the elementary `Source` that starts locking by removing the `Listener`, once any lock stage returned `Gone` or after completion. But in aggregating derived `Sources`, e.g. `race`, there may be other elementary `Sources` that keep a `Listener` that rejected or was completed by one `Source`. Those others would only drop it when they have data available and try to lock it.

To clean this up earlier, the `race` implementation drops `Listeners` in two places: In the `complete` method of the derived `Listener` and, in case of downstream rejection, in `lockNext` of its `PartialLock` wrapper for the downstream `ListenerLock`. It does not perform any explicit dropping when it itself rejects, because it only rejects after a successful `complete`, which already drops. If `lockSelf` of the downstream `Listener` (called in `lockNext`) rejects, it expects that it is considered permanently gone. Therefore, the `race` implementation of `lockNext` drops that downstream `Listener` from the `race Source`, i.e., it drops the `race Listener` that was derived from the downstream `Listener` from all raced `Sources`.

```
trait Listener[-T]:
  def complete(data: T, source: Async.Source[T]): Unit
  val lock: Listener.ListenerLock | Null

object Listener:
  sealed trait LockResult
  case object Locked extends LockResult
  case object Gone   extends LockResult
  trait PartialLock  extends LockResult:
    val nextNumber: Long
    def lockNext(): LockResult

  type LockMarker = PartialLock | Locked.type

  trait ListenerLock:
    val selfNumber: Long

    def lockSelf(source: Async.Source[?]): LockResult
    protected def release(to: Listener.LockMarker): ListenerLock | Null
```

Listing 7. Final Listener Interface

The cleanup in `complete` has a subtle detail. If a derived `race Listener` is completed, the completing `Source` automatically drops that `Listener` but the other raced `Sources` do not. As dropping can be an expensive operation, especially if the completing `Source` is itself a `race`, the `Listener` should only be dropped from all other `Sources` (see Section 5.2). For this reason, a `Source` parameter is passed to `continue` (and `lockSelf`, in case another `Source` may need it later).

5. Performance

In the following sections, some microbenchmarks of implemented operations are presented. They are written in Scala using the Java Microbenchmark Harness (JMH, [9]) and the SBT plugin [10]. They were run on a 2015 MacBook Air and serve a purely comparative purpose.

5.1. Listener Deadlock Prevention

This benchmark was conducted to compare two implementations of parallel `Listener` locking (cf. Section 4.3). One implementation employs delimited continuations to suspend before locking, while the other implementation ("explicit") returns intermediate results after locking. This benchmark measures throughput of two operations (*complete* and *lockBoth*) with both implementations and two `Listener` setups.

The *complete* measures locking and completion of a single complex `Listener`, whereas *lockBoth* locks the complex `Listener` in parallel with a dummy `Listener` (lock-free, accept and ignore). The complex `Listener` is created by attaching a dummy `Listener` to a complex `Source` and extracting the derived `Listener` from the base `Source`. The complex `Source` is either a `race` of 20 dummy `Sources` ("Broad Race") or a nested `race` of a single `Source` in 20 levels ("Deep Race").

Throughput was measured (operations per millisecond) in 3 forks, each running 2 seconds of warmup, followed by 2 iterations of 2 seconds. Each of the 6 iterations was used as a data point and their average and standard error (assuming normal distribution) are presented in Figure 4.

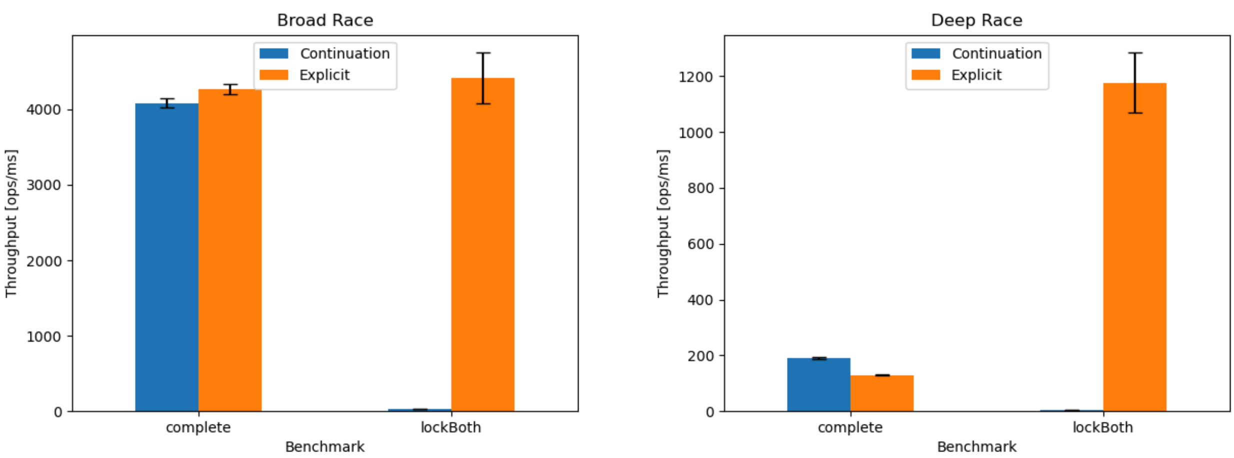

FIGURE 4. Benchmark: Continuation vs Explicit

There is a clear difference in *lockBoth* performance (factor >150 in broad race and >50 in deep race). The difference in *complete* in the broad case is too small to be valuable in face of implementation optimizations applied later. In the deep race, the continuation approach has roughly 45% higher throughput (30% faster). This is expectable as every level in the explicit approach involves a `PartialLock` proxy and thus more checks and invocations. It is, however, more important to provide fast channels (which require perform `lockBoth`) than deeply nested `race`.

The drastic performance difference between lockBoth and complete in "Deep Race" was the cause for the next benchmark.

5.2. Listener Dropping on Completion

These benchmarks were conducted to analyze the cost of `Listener` dropping in `race`. When a `race` `Source` is completed by a `Listener` from one `Source`, the `Listener` is gone from now on and should thus be dropped from the remaining `Sources`. The same holds when the dowstream `Listener` of the uncompleted `race Listener` rejects (cf. Section 4.4).

In the initial approach, the `Listener` was dropped from all `raced Sources`, including the currently locking/completing one (case "all"). This turned out to be much slower in the "Broad Race" scenario (Section 5.1) than a "release" (which does not drop), which is almost as fast as only locking without releasing nor completing afterwards ("lockSingle").

To check whether this is caused by dropping, we ran completion without dropping ("noDrop") or with partial dropping, skipping the supplying `Source` via `Seq.filter` ("filter") or manually using an `if` inside a `for` loop ("loop"). Throughput was measured (ops/ms) in 5 forks, each running 5 seconds of warmup followed by 5 iterations of 1 second. The resulting 25 datapoints per operation are plotted in Figure 5 (left).

A second benchmark with the same parameters was conducted with the final solution, comparing only locking ("lockCompletely"), lock followed by complete ("complete"), and lock followed by release ("releaseCompletely"). The results are shown in Figure 5 (right).

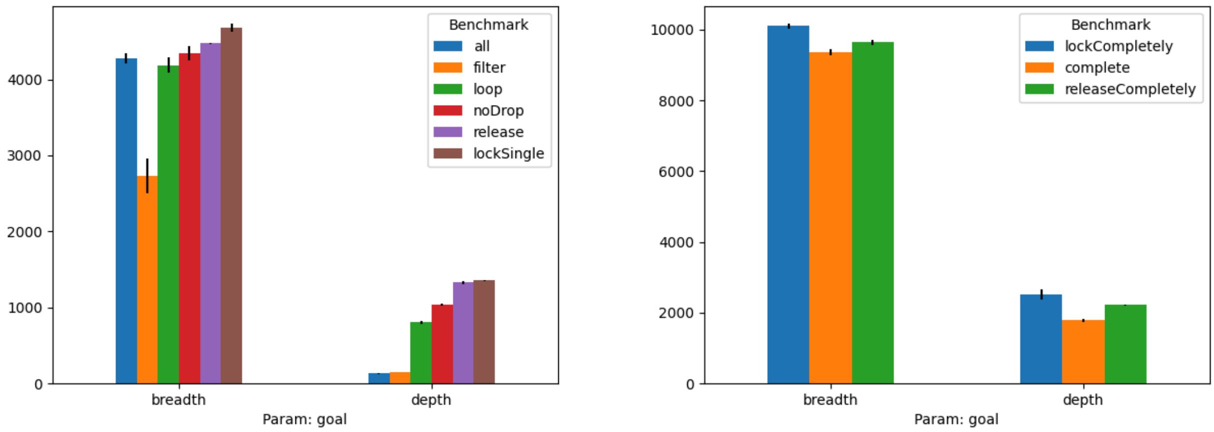

FIGURE 5. Benchmark: Listener Dropping

In the first benchmark, we see that the performance difference appears only in the deep race ("depth") and is largely, but not completely explained by `Listener` dropping (all vs noDrop vs release). For some reason that was not investigated further, looping proved to be much faster than a combination of `filter` and `foreach`.

In the final implementation, this difference between completing and releasing is still present. This seems unavoidable, as completion includes data passing and state update in a recursive manner, compared to releasing locks in compiler-optimized tail-recursion.

5.3. Race Under Contention

This benchmark measures performance degradation when many `Sources` try to lock and complete `race Listener` in parallel. The complex `Source` is created as a two-level `race` where the first level groups the dummy `Sources` in groups of 5. All those derived `Sources` are then `raced` again. Again, a dummy `Listener` is attached to the complex `Source` and extracted from the dummy `Sources`.

In the uncontended scenario, one `Source` is active. It locks and releases the `Listener` 99 times (yielding in between) before locking and completing it. In the contended scenario, all `Sources` are active and do the same. Completing the `Listener` releases a binary semaphore that is blockingly awaited at the end of the benchmark body. The yield is necessary because the `Sources` run as `RunnableFutures` on virtual threads and the JVM does not employ preemption.

Throughput was measured (ops/sec) in 10 forks, each running 5 seconds of warmup followed by 8 iterations of 2 seconds. The resulting 80 datapoints per operation are plotted with their 99,9% confidence interval (assuming normal distribution) in Figure 6.

The uncontended performance decreases due to `Listener` registration and dropping. When all `Sources` are active, they suffer from contention heavily. This seems to be unavoidable given the exclusive, lock-based nature of `race`.

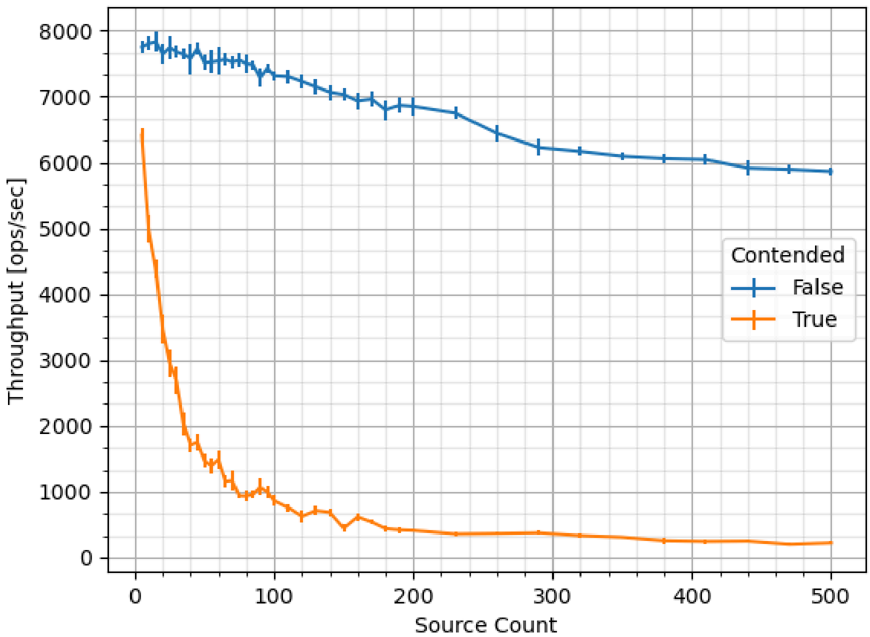

Figure 6. Benchmark: Contended Race

5.4. Future Overhead

This benchmark measures the overhead of the `RunnableFuture` abstraction, compared to plain JVM virtual threads and the Ox library (see Section 6). In the plain JVM benchmark ("VThread"), a no-op virtual thread is started and joined. In Ox, a `scope` is created and a `fork` spawned and joined. In Gears, a `Async.blocking` scope is created and inside, a `RunnableFuture` is spawned and awaited.

Throughput was measured (ops/ms) in 5 forks, each running 5 seconds of warmup followed by 5 iterations of 2 seconds. The resulting 25 datapoints per operation are plotted with their 99,9% confidence interval in Figure 7.

The Gears `RunnableFuture` is slower than the two other benchmarks. In comparison to plain virtual threads it achieves 12.6% lower throughput (14.4% slower). To see whether this can be improved, benchmarks including profiling would be needed.

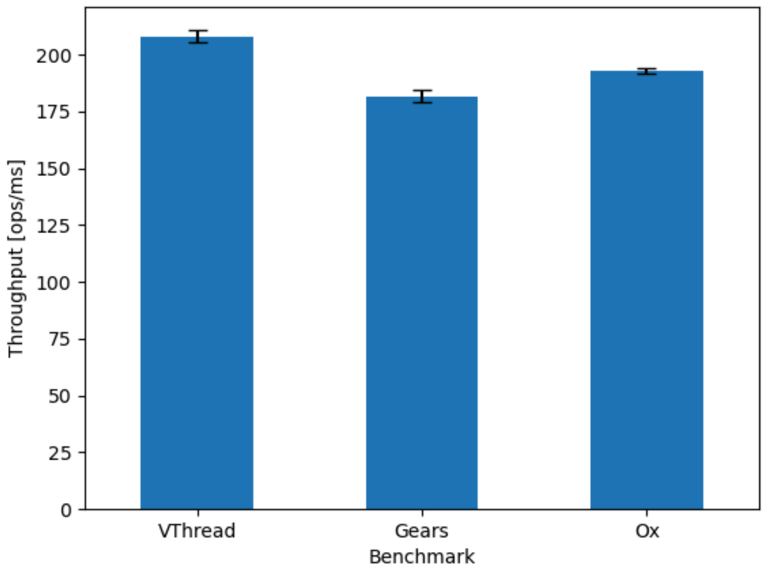

FIGURE 7. Benchmark: Future Overhead

6. Related Work

There are various projects that also work on structured and/or direct-style concurrency for Scala. The library Ox has very similar goals in providing a library for structured concurrency, including "high-level concurrency operators, safe low-level primitives", and communication primitives [11]. However, its scope also includes more advanced error, resilience, and resource handling, which Gears currently does not. It also focuses solely on the JVM, wrapping the new concurrency APIs [12], [13] in addition Loom [6].

The project dotty-cps-async is a tool to allow direct-style programming of a monad [14]. The implementation is a compiler plugin that transforms the direct-style code to continuation-passing style (CPS). This can be used in place of delimited continuations for providing async/await. While this could be valuable for ScalaJS that lacks both delimited continuations and fibers, it also has important limitations: high-order functions need manual support to be applicable to functions that await.

Finally, there is also a feature request for the Scala language to incorporate a *generator* or suspendable functions [15]. This is an alternative to exposing delimited continuations using context parameters [3]. It still faces the same implementation difficulties.

7. Conclusion

In this report, an overview of the current status of Gears and the important changes from this project's scope were presented. The library runs on an abstraction of the asynchronous capabilities that can be implemented using both delimited continuations or fibers. The `Listener` trait was refined to allow atomic `race` in face of channel read and send operations.

Future work will be necessary to incorporate IO in the library. This will affect not only the `Async` capability, but also possibly the `Scheduler` interface. Further, concurrency primitives like locks, semaphores, etc. are required. Those implementations can wrap the virtual-thread-enabled primitives from the standard library on the JVM. On Scala Native, this will require more manual work, possibly involving the `Scheduler` as well.

The results presented in this report were achieved together with Cao Nguyen Pham. My work focussed on the aspects described in Section 3, Section 4 (including Section 2.3), and Section 5, as well as the new Promises (Section 2.1) and the cancellation details from Section 2.4. The final `Listener` implementation (Section 4) was developed in close collaboration. For the entire implementation of channels and `Timer` (Section 2.2) and for the Scala Native implementation of the Support layer (Section 3), my contribution was limited to reading, understanding, and fixing (review).

References

[1] Martin Odersky, "Towards A New Base Library for Asynchronous Computing". Feb. 16, 2023.

[2] Bartosz Białas, "Summary of the project", Jun. 2023.

[3] Martin Odersky, "Direct Style Scala". Mar. 24, 2023.

[4] MDN Web Docs, "Promise() constructor". [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/Promise

[5] "Gears futures.scala - branch 'channels-working'". Jun. 27, 2023. [Online]. Available: https://github.com/lampepfl/gears/blob/channels-working/src/main/scala/async/futures.scala

[6] OpenJDK Contributors, "Project Loom". [Online]. Available: https://wiki.openjdk.org/display/loom

[7] Cao Nguyen Pham, "Project Report: Native delimited continuations as asynchronous programming primitives on Scala Native", Jun. 2023.

[8] Ron Pressler and Alan Bateman, "JEP 444: Virtual Threads". Oct. 26, 2023. [Online]. Available: https://openjdk.org/jeps/444

[9] OpenJDK Contributors, "JMH". [Online]. Available: https://openjdk.org/projects/code-tools/jmh/

[10] Scala SBT Contributors, "sbt-jmh". [Online]. Available: https://github.com/sbt/sbt-jmh

[11] SoftwareMill, "Ox". [Online]. Available: https://github.com/softwaremill/ox

[12] Alan Bateman and Ron Pressler, "JEP 428: Structured Concurrency (Incubator)". Jun. 08, 2023. [Online]. Available: https://openjdk.org/jeps/428

[13] Andrew Haley and Andrew Dinn, "JEP 429: Scoped Values (Incubator)". Nov. 29, 2023. [Online]. Available: https://openjdk.org/jeps/429

[14] rssh, "dotty-cps-async". [Online]. Available: https://github.com/rssh/dotty-cps-async

[15] joshlemer, "Generators via Continuations / Suspendable Functions". [Online]. Available: https://contributors.scala-lang.org/t/generators-via-continuations-suspendable-functions/3933