

Verifying an Effect-Based Cooperative Concurrency Scheduler in Iris

Adrian Dapprich

Feb 1, 2024

Table of Contents

1. Introduction
2. Verifying the Eio Scheduler
3. Verifying Eio's Customized CQS
4. Extending the Scheduler with Thread-Local variables
5. Evaluation
6. Conclusion
7. Appendix

1. Introduction (WIP)

- As a motivation for the work: program verification, **safety** and why we care about it.
- Iris is a new separation logic which allows proving safety for programs using mutable shared state.
- Many programs nowadays use user-level concurrency to handle a big number of tasks. As an example for OCaml 5 there exists the Eio library which provides concurrency primitives using effect handlers.
- Effect handlers are a versatile concept which allow a modular treatment of effects, the implementation in form of a handler is separated from the code using the effect, and it's more lightweight than monads. Give a simple example of state.
 - The biggest upside is that they are more composable than monads which often require rewriting of parts of the program into monadic style
 - In theory effect can be tracked by the type system, although OCaml 5 does not do that yet.
 - Explain the concept of **effect safety** here.
 - Mention that continuations can only be invoked once? (not really necessary info)
- We want to verify some parts of the Eio library but the standard Heaplange language for Iris does not support effect handlers.
 - Hazel is an Iris language formalizing effect handlers using protocols.
 - Since OCaml 5 allows both effect handlers and mutable shared state we had to add a multi-threaded semantics to Hazel.
- Inherent part of a scheduler is liveness, because it is responsible for running all fibers to completion. Unfortunately it is hard to prove liveness properties in Iris so we just focus on safety and effect safety.

1.1. The Eio Library (WIP)

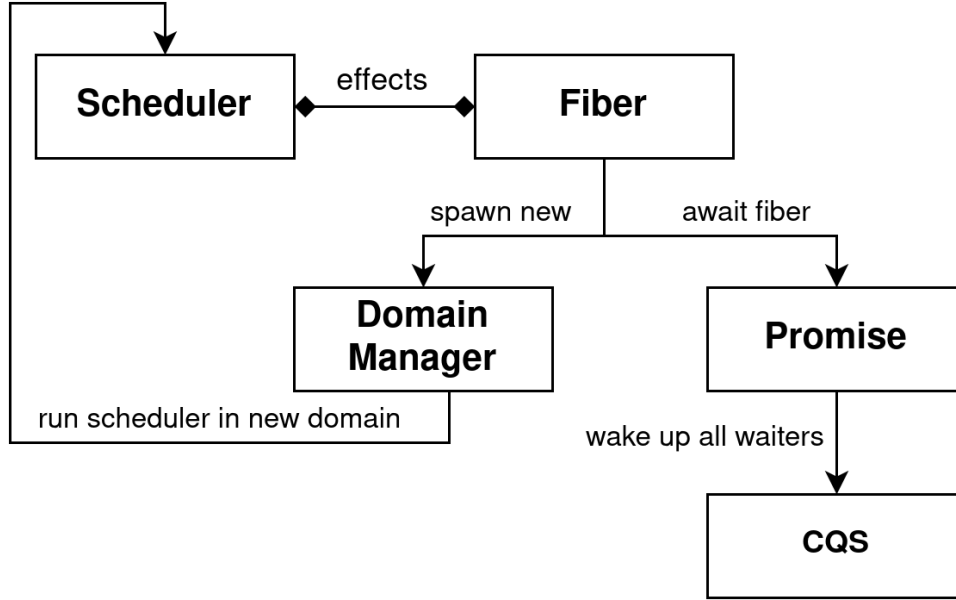
- Library for cooperative concurrency in OCaml 5.
- Implements switching between tasks using effect handlers.
- A fiber is a normal OCaml function which may perform effects that are handled by a scheduler.
- Each scheduler is only responsible for a single thread, more can be spawned.
- It offers abstractions to operating system resources to fibers, e.g. network, filesystem, timers etc.
- It also offers synchronization and message passing constructs like mutexes & channels which are specialized to handle fibers, i.e. a mutex does not suspend the system-level thread, but the fiber.

1.2. Focus and Structure of the Thesis

Eio aims to be the standard cooperative concurrency library for OCaml 5 so it includes many functions for structured concurrency of fibers (e.g. `Fiber.{first, any, both, all}` which run two or more fibers and combine their results), support for cancelling fibers, abstractions for operating system resources, a different scheduler implementation per OS, and

synchronization constructs like promises and mutexes. But for this work we restrict ourselves to verifying the safety and effect safety of Eio’s core functionalities: 1. Running fibers in a “common denominator” scheduler that does not interact with any OS resources, 2. awaiting the result of other fibers using the *promise* synchronization construct, 3. and spawning new schedulers to run fibers in another thread.

Below we show the simplified module hierarchy of the concepts we focus on.



Fibers can fork off new fibers using the **Fork** effect and suspend execution using the **Suspend** effect, which are both handled by the scheduler. This is all discussed in [ref, section 2.2]. Promises are built on top of a *CQS* datastructure, which is a kind of lock-free condition variable that is used by fibers to suspend execution until a promise is fulfilled. The specification of promises is discussed in [ref, section 2.3]. The CQS specification is already verified using Iris, but Eio uses a custom implementation for which we had to adapt the proof and we discuss this process in [ref, section 3]. Fibers in Eio also have access to *thread-local variables* by performing an effect, which is discussed in [ref, section 4]. They are thread-local in the sense that they are shared between all fibers of one scheduler. Finally, we discuss our addition of multi-threading to the Hazel operational semantics in order to model running schedulers in different threads. This turned out to be technically trivial so we only discuss it in the appendix and take a multi-threaded semantics and support for Iris’ *shared invariants* as a given.

1.3. Contributions

To summarize our contributions, in this thesis we verify the **safety** and **effect safety** of a simplified model of Eio which serves as an extended case study on the viability of Hazel for verifying programs with effect handlers. This includes:

- The verification of the basic Eio fiber abstraction running on a common denominator scheduler.
- An adaptation of the existing verification of CQS to the customized version used by Eio.
- Adding multi-threading to Hazel’s operational semantics, which shows we can reason about programs with multi-threading & effect handlers.

2. Verifying a Simplified Eio Scheduler

[TODO mention that all code examples are an OCaml rendering of the verified Hazel code, based on but not equal to the Eio code]

Cooperative concurrency schedulers are (often/commonly?) treated in the literature on effect handlers [ref, Paolos dissertation, Dolan paper, Dan Leijen paper] because they are a lucid example for the usefulness of handling delimited continuations in this way. Generally, the scheduler contains an effect handler and a fiber is just a normal function. The fiber can yield execution by performing an effect, jumping to the effect handler (i.e. the scheduler) and providing it with the rest of the fiber’s computation in the form of a continuation. The scheduler has a collection of continuations and by invoking one of them it schedules the next fiber. This approach is also used in Eio.

We can therefore use the simple cooperative concurrency scheduler case study from the dissertation of de Villhena [ref] as a starting point for our verification work. In the following section we discuss the implementation of the simplified Eio model in more detail to give an intuition about what their specification and the logical state to prove them should look like. From this intuition we will derive a formalization in [ref, section 2.2 & section 2.3].

2.1. Implementation

Let us first get an idea of how different components of the core Eio fiber abstraction interact by looking at their types. `Scheduler.run` is the main entrypoint to Eio and it is provided a function which represents the first fiber to be executed. `Fiber.fork_promise` is also provided a function which represents a fiber, but this one will be forked so that it runs concurrently. It returns a promise holding the eventual return value of the new fiber. The promise is thread-safe so any other fiber can use the `Promise.await` function to block until the value is available. Common problems like deadlocks are not prevented in any way and are the responsibility of the programmer.

```
Scheduler.run : (() -> 'a) -> ()
Fiber.fork_promise : (() -> 'a) -> 'a Promise.t
Promise.await : 'a Promise.t -> 'a
```

We present code examples in a pseudo-OCaml 5 syntax because the concrete syntax of effect handlers is verbose. Instead, we use an overloading of the match syntax that is common in the literature which includes cases for handled effects.

```
(* declares an effect E that carries a value of type int and has a bool return value. *)
effect E : int -> bool
```

(Matches on the expression e and evaluates the second branch if it performs the effect E.
The continuation k captures the rest of the computation of e. *)*

```
match e with
| v -> ...
| effect (E v) k -> ...
```

Scheduler.run As mentioned above this is the main entry point to the Eio library. It receives the main fiber as an argument and sets up the scheduler environment. The `run_queue` is a queue of ready fibers and the `next` function pops one fiber (i.e. function) and evaluates it. The inner `fork` function is called once on each fiber to execute it and handle any performed effects.

Handling a `Fork` effect is simple because it just carries a new fiber to be executed so the handler recursively calls the `fork` function.

Handling a `Suspend` effect may look complicated at first due to the higher-order `register` function. This effect is used by fibers to suspend execution and through the `register` function they can control under what conditions they should resume execution again. The key point is that as long as the continuation `k` is not invoked, the fiber will not continue to execute. So the `waker` callback is defined to “wake up” a fiber by placing its continuation into the run queue and the `register` function can install it as a callback somewhere (or even call it directly). Waking up should be possible across thread boundaries, which is why the run queue in Eio needs to be thread-safe.

```
effect Fork : (() -> 'a) -> ()
type 'a waker : 'a -> ()
effect Suspend : ('a waker -> ()) -> 'a
```

```
let run (main : () -> 'a) : () =
  let run_queue := queue_create () in
  let rec fork fiber =
    match fiber () with
    | _ -> next run_queue
    | effect (Fork fiber) k ->
      queue_push run_queue (fun _ -> invoke k ());
      fork fiber
    | effect (Suspend register) =>
      let waker = fun v -> queue_push run_queue (fun _ -> invoke k v) in
      register waker;
      next run_queue
  in
  fork main
```

Fiber.fork_promise This is the basic way to create a new fiber in Eio and the only one we model in our case study. It will create a promise and spawn the provided function as a new fiber using the `Fork` effect. When `f` is reduced to a value `result`, it will fulfill the promise with that value and signal all fibers waiting for that result to wake up. The major difference to the implementation of de Vilhena is that promises in Eio are entirely

handled by the fiber, and not in the effect handler code of the scheduler. This achieves a better separation of concerns and simplifies the logical state needed for the proof.

```
let fork_promise (f : () -> 'a) : 'a Promise.t =
  let p = Promise.create () in
  let fiber = fun () ->
    let result = f () in
    match Atomic.get p with
    | Done _ -> error "unreachable"
    | Waiting cqs ->
      Atomic.set p (Done result);
      CQS.signal_all cqs
  in
  perform (Fork fiber)
  p
```

Promise.await This function is arguably the most complicated in our case study which is partly due to the **Suspend** effect and also due to the use of CQS functions. **Promise.await p** suspends execution of the calling fiber until **p** is fulfilled and then returns its value. The “suspend execution” part is handled by performing the **Suspend** effect because the scheduler will switch to the next fiber in its run queue. Then, the “until **p** is fulfilled” part is handled by CQS [ref, paper].

CQS is an implementation of the observer pattern and functionally similar to condition variables¹ in languages like C++ (only lock-free), allowing fibers to register callbacks that will be called when a condition is signalled. In our case, we want to wait until **p** is fulfilled, so we register the **waker** function of the **Suspend** effect with CQS. The **Fiber.fork_promise** function then uses **CQS.signal_all** to call all registered **wakers** after it fulfills the promise.

Therefore, after the **Suspend** effect returns we know that the **Done** branch is reached.

Because CQS is lock-free, the register function must take care of possible interleavings where the promise is fulfilled in another thread. First, if **CQS.register** notices that there is a concurrent **CQS.signal_all** it will directly call the **waker**. Otherwise, the **waker** is registered but in fact the **CQS.signal_all** might have already finished before **CQS.register** even started. In this case the callback would be “lost” in the CQS, never to be called. To avoid this, **register** must check the state of the promise again, try to cancel the callback registration and call it directly.

```
let await (p: 'a Promise.t) : 'a =
  match Atomic.get p with
  | Done result -> result
  | Waiting cqs ->
    let register = fun waker ->
      let register_result = CQS.register cqs waker in
      match register_result with
      | None -> ()
      | Some register_handle ->
```

¹https://en.cppreference.com/w/cpp/thread/condition_variable

```

match Atomic.get p with
| Done result ->
  if CQS.try_cancel register_handle
  then waker ()
  else ()
| Waiting _ -> ()
in
perform (Suspend register);
match Atomic.get p with
| Done result -> result
| Waiting _ -> error "unreachable"

```

The only **safety** concerns in the above implementations are `Fiber.fork_promise` expecting the promise to be unfulfilled and `Promise.await` expecting the promise to be fulfilled in the last match. In the next section we show how the former is addressed by defining a unique resource that is needed to fulfill a promise, and the latter is a consequence of the protocol of the `Suspend` effect.

2.2. Specifications

[TODO say that logical state is based on de Vilhena's case study but Eio both allows some simplifications & necessitates less powerful specifications since fibers are more free.]

Protocols First we look at the protocols for the `Fork` and `Suspend` effect.

$$\begin{aligned}
\text{isSuspend} \text{ reg} &::= \forall \text{waker}. \text{promiseInv} \multimap \\
&\quad (\forall v. P \ v \multimap \text{ewp} (\text{waker } v) \langle \perp \rangle \{ \top \}) \multimap \\
&\quad \triangleright \text{ewp} (\text{reg waker}) \langle \perp \rangle \{ \top \} \\
\text{SUSPEND} &::= ! \text{reg } P (\text{reg}) \{ \text{isRegister reg} \}. \\
&\quad ? \ v \ (v) \{ P \ v \} \\
\text{FORK} &::= ! \ e \ (e) \{ \triangleright \text{ewp} (e \ ()) \langle \text{Coop} \rangle \{ \top \} \}. \\
&\quad ? \ () \{ \top \}
\end{aligned}$$

Compared to de Vilhena's case study, `Fork` is almost the same but since promise handling is done entirely in the fiber, the scheduler does not interact with the return value so the `ewp` degenerates to a trivial postcondition.

The protocol for `Suspend` is entirely new. From the type of the `Suspend` effect we already know that some value can be transmitted from the party that calls the `waker` callback to the fiber that performed the effect. The protocol now expresses the same idea on the level of resources. By appropriately instantiating `P` the fiber can enforce that some condition holds before it can be signalled to resume execution. For example, in the `Promise.await`

specification below, we ensure that the promise must be fulfilled before the effect returns by instantiating P with `promise_done` γ .

Logical State The most basic ghist state we track is whether a promise is fulfilled or not. The `promise_waiting` γ resource exists as two halves, one owned by the fiber and one by the invariant that tracks all promises. They can be combined when fulfilling the promise, yielding the persistent `promise_done` γ resource.

$$\begin{aligned}
\text{Ready } q \ \phi \ k &\triangleq \forall v. \ \Box \phi(v) \multimap \triangleright \text{PromiseInv } q \multimap \triangleright \text{isQueue } q \ (\text{Ready } q \ (\lambda w. w = ())) \multimap \\
&\quad \text{ewp } (k \ v) \ \langle \perp \rangle \{ _ . \text{True} \} \\
\text{PromiseInv } q &\triangleq \exists M. \left[\begin{array}{c} \bullet \overline{M} \end{array} \right]^\delta * \\
&\quad \forall \{ (p, \gamma) \mapsto \phi \} \in M. \exists b. \left[\begin{array}{c} \bullet \overline{b} \end{array} \right]^\gamma * \\
&\quad \text{if } b \text{ then } \exists ks. p \mapsto \text{Waiting } ks * \forall k \in ks. \text{Ready } q \ \phi \ k \\
&\quad \text{else } \exists v. p \mapsto \text{Done } v * \Box \phi(v) \\
\text{isPromise } p \ \phi &\triangleq \exists \gamma. \left[\begin{array}{c} \circ \{ (p, \gamma) \mapsto \phi \} \end{array} \right]^\delta
\end{aligned}$$

Fig. 12. Internal definitions for the verification of the concurrency library

PromiseInv tracks the state of all existing promises, which are either *Done* or *Waiting*. The *Ready* predicate expresses that k is safe to be called on a value satisfying Φ under the condition that all promises and the scheduler's run queue are well-formed. Note that *Ready* is used both recursively to ensure that all runnable fibers are safe to execute, and mutually recursive with *PromiseInv* to ensure that all fibers waiting for a promise are safe to execute. If a promise is waiting it contains a list of continuations that will be invoked by the scheduler when a fiber finishes execution.

Because of this, the *Ready* predicate

- The logical state of the scheduler tracks the state of all currently existing promises (across all threads) in a shareable invariant using an auth. Since promises are not handled by the scheduler anymore we can completely remove it from the scheduler's spec.
- The *ready* predicate is a lot simplified because the queue parameter is now an invariant and can be dropped so it's not mutually recursive with *is_promise* anymore
- Explain the proof of the spec of `Sched.run`
 - First explain the protocol of the `Fork` effect since it's the simplest, here we also need the spec of the next function.
 - Then explain the protocol of the `Suspend` effect and how the `waker` callback precondition is returned from the protocol, which means that doing a `suspend` effects waits until some user-specified condition is met.
- Explain the proof of the spec of `Fiber.fork_promise`
 - It creates a new promise and uses the `Fork` effect.
 - In the end it changes the logical state of the promise to take out the `resume_all_permit` and fulfill the promise to get a `promise_state_done`. Using both it can then call `Broadcast.resume_all`.
- Explain the proof of the spec of `Promise.await`.

2.3. Spec of Promises (WIP)

- We instantiate the `P` parameter of the protocol of `Suspend` by `promise_state_done` so that the last match succeeds.
- For the inner match we just optionally get the `wp` back and can then call the callback by ourselves.

[TODO this text was copied from an earlier version and not adapted to the current state of the CQS chapter]

Now that we have outlined how the operations work we want to explain the code of `Eio`'s promise in more detail. Since the code is lock-free there are a couple of interleavings the implementation must take care of.

We recall that when awaiting a promise, a fiber first checks if the promise is already fulfilled by atomically loading its state. If it is not fulfilled, the fiber then performs a `Suspend` effect and starts a suspend operation, providing the `waker` of the `Suspend` effect as the handle. The suspend operation might fail because the promise could have been fulfilled concurrently. Since the promise could have been fulfilled in the meantime, the fiber must then again atomically load the state of the promise.

- If it has not been fulfilled the fiber does not need to do anything because it will eventually be woken up by a resume all operation invoking the `waker`.
- But if the promise has been fulfilled the fiber must attempt to cancel the suspend operation. That is because in this situation the suspend operation races with a concurrent resume all operation, which might already have invoked all `wakers` **before** this fiber was able to save its `waker` in the broadcast. In this case the `waker` would be lost and the fiber never resumes execution. If the `waker` has not been invoked yet (either because resume all has not arrived at this waker or it arrived before the waker was saved in the broadcast) the cancellation attempt succeeds and the fiber invokes its own `waker`. Otherwise we know that the `waker` has already been invoked, so the fiber does not need to do anything.

This complicated interplay between two fibers is due to CQS being lock-free but it ensures that fibers only resume execution when the promise is fulfilled and that all `wakers` will be eventually called.

Aside: All `wakers` are eventually called.

This statement is purely based on a reading of the code. It might be possible to formally prove this with an approach like Iron [ref] or Transfinite Iris [ref] because it is a liveness property.

But for the Iron approach it is unclear to us how to formulate the linearity property.

- Since thread-local variables are orthogonal we don't treat them here but in a later section.
- First we look at how we had to adapt the existing CQS verification to the version that is used by `Eio`.

3. Verifying Eio’s Customized CQS (WIP)

CQS [ref, paper] (for CancellableQueueSynchronizer) is an implementation of a synchronization primitive that allows execution contexts to wait until signalled. Its specification is already formally verified in Iris, which we adapted to use in our case study. The nature of a CQS execution context is kept abstract but it is assumed that they support stopping execution and resuming with some value. This is because CQS is designed to be used in the implementation of other synchronization constructs (e.g. mutex, barrier, promise, etc.) which take care of actually suspending and resuming execution contexts as required by their semantics.

In the case of Eio an “execution context” is an Eio fiber but nevertheless CQS works across multiple threads, so fibers can use CQS to synchronize with fibers running in another thread. Eio implements a custom version of CQS adapted from the paper [ref, paper] in the **Broadcast** module, which in turn is used in the implementation of the *promise* synchronization construct. In this chapter we describe the behavior of Eio’s *customized CQS*, highlight differences to the *original CQS*, and explain how we adapted the verification of the original CQS for our case study. If something applies to both the customized and original version we just use the term *CQS*. After having presented the adapted specification for the **Broadcast** module we can then explain the implementation of the **Promise** module which we kept abstract in section 1.

Operations of CQS

The original CQS supports three operations that are interesting to us. In a *suspend operation* the requesting execution context wants to wait until signalled. It places a handle to itself in the datastructure and is expected to stop execution afterwards. But before it actually stops execution it can use the *cancel operation* to try to cancel the *suspend operation*. Finally, a *resume operation* can be initiated from a different execution context. It takes one handle out of the datastructure and uses it to signal the original execution context that it can resume execution. This fails if the *suspend operation* (and thereby the handle) had already been cancelled.

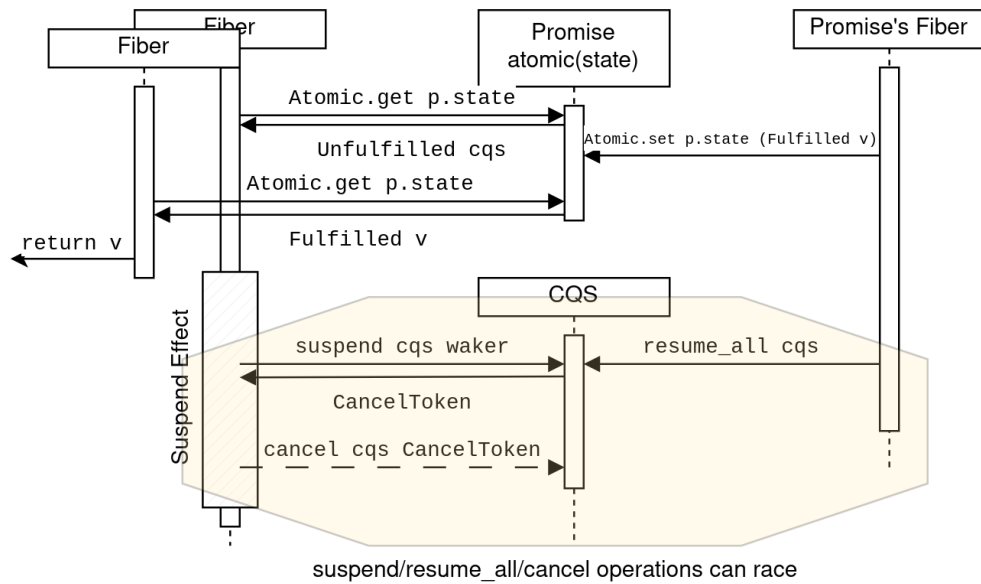
These operations enable a single execution context to wait until it is signalled by another. Eio’s customized CQS supports an additional operation called the *resume-all operation*. As the name implies, it is a *resume operation* that applies to all currently saved handles. This operation was added so that **all** fibers waiting on a promise can be signalled when the promise is fulfilled.

To understand the operations it is helpful to view them in the context of their Eio implementation. Here, what we called the “handle” to an execution context is the **waker** callback resulting from a fiber performing a **Suspend** operation. We recall that if the **waker** callback is invoked, its fiber is placed into the scheduler’s run queue and will therefore resume execution. We show the operations’ OCaml types and also how the operations are used in the outer synchronization construct (i.e. an Eio *promise*).

An interaction with CQS as described in [ref, paper] is always guarded by first accessing some atomic variable. In the case of Eio, the atomic variable holds the state of the promise, which can either be **Unfulfilled cqs** – holding a customized CQS instance – or

Fulfilled v – holding the final value v of the associated fiber.

- If the promise is already fulfilled with a value, a requesting fiber immediately returns that value.
- If the promise is not yet fulfilled, a requesting fiber will perform a **Suspend** effect in order to stop execution and use the *suspend operation* to wait until the promise is fulfilled.
- Optionally, it can also use the *cancel operation* afterwards.
- The fiber that is associated with the promise will fulfill it with a value and then use the *resume-all operation* to signal all waiting fibers that they can now retrieve the value.



It is important to note that since CQS is lock-free and fibers can run on different threads there can be a race between concurrent *suspend*, *cancel* and *resume-all operations*. Possible interleavings and the necessity of the *cancel operation* are explained in section [ref, Promise Implementation]. This example illustrates that a CQS instance always acts as a thread-safe store for cancellable callbacks. More precisely, it is a FIFO queue but a *resume-all operation* dequeues all elements at once.

That CQS is “just” a store for cancellable callbacks is also reflected in the rather barebones types of the operations as implemented in OCaml. A CQS instance can be **created** and shared between different threads. New callbacks are inserted using the **suspend** function, yielding an optional **request** value. If **suspend** returns **None** the callback has already been invoked due to a concurrent **resume_all**. A **request** value can then be used to cancel the insertion, signifying that a fiber can only cancel its own callback. The **resume_all** function (logically) consumes the CQS, which will become more clear when we present the specifications in [ref, Verification of the Broadcast module]

type t

type request

val create : unit -> t

val suspend : t -> (unit -> unit) -> request option

val cancel : request -> bool

val resume_all : t -> unit

Implementation and Logical Interface of CQS

CQS is implemented as a queue of *cells* with two pointers pointing to the beginning and end of the active cell range, the *suspend pointer* and the *resume pointer*. Cells not reachable from either pointer are garbage collected but their logical state is still tracked. There is a stack of operations for manipulating these pointers to implement the higher-level functionality but they are not part of the public API so we do not focus on them. Each cell is a container for one handle and the logical state of the queue tracks the logical state of all existing cells shown in figure [ref, below].

The number of active cells n (i.e. the length of the queue) is tracked by the logical resource `cqs_state n`. In normal usage of CQS, the atomic variable of the outer synchronization construct would encode the length of the queue in its value and keep this resource in an associated invariant. Logically changing the length of the queue is done using *enqueue* and *dequeue registration* operations when opening this invariant.

As we saw before, however, for promises the exact length of the queue is irrelevant because the *resume-all operation* will always set the length to 0. So in the adapted proof we keep the `cqs_state n` resource in the invariant of CQS itself. As a consequence we also move the *enqueue* and *dequeue registration* out of the public API because they are now done internally.

Verification of the Broadcast Module

In the following we describe the specifications we proved for the three operations `suspend`, `cancel` and `resume_all` of Eio's Broadcast module, in which points they differ from the specifications of the original CQS operations, and what changes we did to the internal logical state of CQS to carry out the proofs.

The first major change was replacing the future-based interface of the suspend operation with a callback-based interface. In the original CQS, performing a suspend operation returns a new future, which is also inserted as the handle into the queue. The execution context can then use the future to stop execution because it is assumed there is a runtime that allows suspending execution until the completion of a future. But Eio cannot use this interface because it uses the customized CQS to *build* the runtime that allows fibers to suspend until the completion of a promise. As explained above, Eio implements CQS with a callback-based interface where the fiber performing the suspend operation passes in a callback as the handle and afterwards implicitly stops execution. Performing a resume operation analogously invokes the callback, instead of completing the future.

This changes the logical state of CQS only slightly. The original CQS tracked the state of the future for each cell and managed *futureCancellation* and *futureCompletion* tokens.

In the customized CQS we analogously track the state of the callback for each cell and manage *callbackInvocation* and *callbackCancellation* tokens.

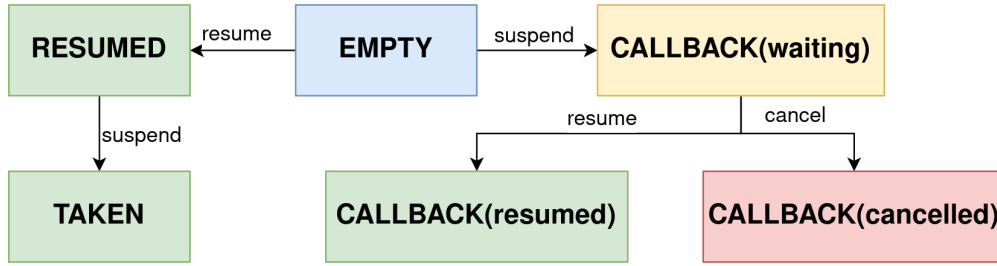
For all three operations, the Eio implementation differs from the implementation already verified in the original CQS (i.e. some reordered instructions or a slightly different control flow) and they have different specifications as discussed below. However, the specifications of the underlying operations for manipulating the cell pointers are modular enough to allow us to prove the new specifications for *suspend* and *cancel*. Note that the presented specifications are cleaned up for readability.

Aside: Implementation of *resume_all*

Eio implements *resume_all* by atomically increasing the **resume pointer** by some number *n*, instead of just 1 like in the original re. Because of technical differences between the infinite array implementation in the CQS mechanization & the infinite array impleme. Instead, I actually defined *resume_all* simply as a loop over a *resume* operation.

Since *resume_all* is only called once I posit that this verification is still valid but I still want to verify Eio's *resume_all* and remove this

The logical state of an individual cell is changed by the functions according to the following diagram.



create Creating a CQS instance requires *inv_heap_inv* which is an Iris propositions that we are in a garbage-collected setting. It creates an *is_cqs* γ q which is a persistent resource that shows the value q is a CQS queue, along with a collection of ghost names we summarize with γ . The resource *cqs_state* n mentioned above is now kept inside *is_cqs* γ q . It also returns the unique resource *resume_all_permit* γ , which is held by the enclosing promise and allows calling the *resume_all* function once.

Theorem *create_spec*:

```

{{{ inv_heap_inv }}}
  newThreadQueue #()
  {{{  $\gamma$   $q$ , RET  $q$ ; is_cqs  $\gamma$   $q$  * resume_all_permit  $\gamma$  }}}.

```

suspend For a *suspend* operation the *suspend permit* from the original CQS is not needed anymore since we do the *enqueue registration* internally. The *is_waker* resource is defined as $V' \multimap \text{EWP } k () \{ \{ T \} \}$ and represents the permission to invoke the callback k . We instantiate V' with *promise_state_done* γp so that the callback transports the knowledge that the promise has been fulfilled. *is_waker* is not persistent because the callback must be invoked only once and it might be accessed from a different thread.

The *suspend* function will advance the *suspend pointer* to allocate a new cell in the

EMPTY logical state. If there is a concurrent call to `resume_all` which changed the cell to the **RESUMED** logical state before this function can CAS the callback into the cell, the callback is invoked immediately and **NONEV** is returned. In this case, the state of the cell will be set to **TAKEN**. Otherwise the callback is saved in the cell, which is advanced to the **CALLBACK(waiting)** logical state and a `is_suspend_result` resource is returned as the cancellation permit.

Theorem `suspend_spec` γ q k :

```

{{{ is_cqs  $\gamma$   $q$  *
  is_waker  $V'$   $k$  }}}
suspend  $q$   $k$ 
{{{  $v$ , RET  $v$ ;  $\ulcorner v = \text{NONEV} \urcorner$   $v$ 
   $\exists \gamma k r, \ulcorner v = \text{SOMEV } r \urcorner$  *
  is_suspend_result  $\gamma \gamma k r k$  }}}.

```

cancel The specification of the *cancel operation* is a lot simplified compared to the original due to removed features. The `is_suspend_result` resource is used as a permission token and the r value is used to find the callback that should be cancelled.

If the callback had already been invoked by a concurrent call to `resume_all` (i.e. the logical state is **CALLBACK(resumed)**) the function returns `false` and no resources are returned to the caller. Otherwise, the permission to invoke the callback is returned and the cell is advanced to the **CALLBACK(cancelled)** logical state.

Theorem `try_cancel_spec` γ q γk r k :

```

{{{ is_cqs  $\gamma$   $q$  *
  is_suspend_result  $\gamma \gamma k r k$  }}}
cancel  $r$ 
{{{ ( $b$ : bool), RET  $\#b$ ; if  $b$  then is_waker  $V'$   $k$ 
  else True }}}.

```

resume_all The specification of the *resume-all operation* is also a lot simplified compared to the specification of the original *resume operation* because we removed multiple unused features. The `resume_all_permit` is a unique resource used to ensure the function can only be called once. The V' resource must be duplicable because it will be used to invoke multiple callbacks, which have V' as their precondition. It does not return any resources because its only effect is making an unknown number of fibers resume execution, which is not something we can easily formalize in Iris.

Theorem `resume_all_spec` γ q :

```

{{{ is_thread_queue  $\gamma$   $q$  *
   $\Box V'$  *
  resume_all_permit  $\gamma$  }}}
resume_all  $q$ 
{{{ RET  $\#()$ ; True }}}.

```

Features Removed from Original CQS

The original CQS supports multiple additional features like a synchronous mode for suspend and resume, and also a smart cancellation mode. These features enlarge the state space of CQS and complicate the verification but are not used in Eio so when we ported the verification of CQS to our Eio case study we removed support for these features. This reduced the state space of a cell shown below (taken from the original paper) to something more manageable for us when adapting the proofs.

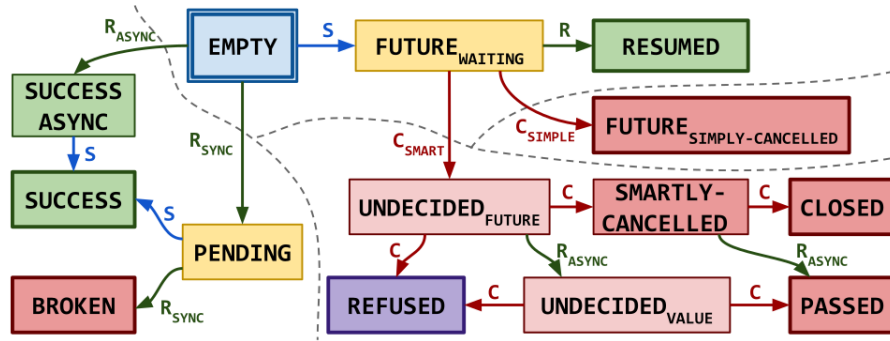


Figure 14: The state transition system for a single cell from the logical perspective.

Due to this, the part of the verification of the original CQS that we had to customize for Eio shrunk by approximately 1300 lines of Coq code from the original 3600 lines of Coq code, while there is an additional ~4000 lines of Coq code that we did not need to adapt.

4. Extending the Scheduler with Thread-Local Variables (WIP)

- How thread-local variables can be used.
- Explain the GetContext effect in Eio and how we model it in our scheduler.
- How we adapt our logical state to include GetContext. And explain that we need to parameterize the protocol to solve the issue of shared knowledge between the scheduler and fiber.

6. Evaluation (WIP)

7. Conclusion (WIP)

Bibliography

- A Formally-Verified Framework For Fair Synchronization in Kotlin Coroutines
- A Separation Logic for Effect Handlers
- A Type System for Effect Handlers and Dynamic Labels
- Concurrent System Programming with Effect Handlers

- Structured Asynchrony with Algebraic Effects
- Retrofitting Effect Handlers onto OCaml
- Retrofitting Parallelism onto OCaml

Appendix

A. Translation Table

Eio	Thesis	Mechanization
enqueue	waker callback	waker
f	register function	register
Fiber.fork_promise	Fiber.fork_promise	fork_promise
Promise.await	Promise.await	await
Broadcast & Cells	CQS	CQS

B. Towards A Multi-Threaded Scheduler

OCaml 5 added not only effect handlers but also the ability to use multiple threads of execution, which are called *domains* (in the following we use the terms interchangeably). Each domain in OCaml 5 corresponds to one system-level thread and the usual rules of multi-threaded execution apply, i.e. domains are preemptively scheduled and can share memory. Eio defines an operation to make use of multi-threading by forking off a new thread and running a separate scheduler in it. So while each Eio scheduler is only responsible for fibers in a single thread, fibers can await and communicate with fibers running in other threads.

In order for a fiber to be able to await fibers in another thread, the `wakers_queue` [note it will be in the Simple Scheduler section] from above is actually a thread-safe queue based on something called CQS, which we will discuss in detail in a later section.

Heaplang supports reasoning about multi-threaded programs by implementing fork and join operations for threads and defining atomic steps in the operational semantics, which enables the use of Iris *invariants*. In contrast, Hazel did not define any multi-threaded operational semantics but it contained most of the building blocks for using invariants. In the following we explain how we added a multi-threaded operational semantics and enabled the use of invariants.

Aside: Memory Model in OCaml 5

In the OCaml 5 memory model, `*atomic variables*` are needed in order to access shared memory without introducing data races. Instead of modelling atomic variables in Hazel, we continue to use normal references because the multi-threaded operational semantics by definition defines all memory operations to be sequentially consistent. This seems to be the sta

Adding Invariants to Hazel

Invariants in Iris are used to share resources between threads. They encapsulate a resource to be shared and can be opened for a single atomic step of execution. During this step

the resource can be taken out of the invariant and used in the proof but at the end of the step the invariant must be restored.

Hazel did already have the basic elements necessary to support using invariants. It defined a ghost cell to hold invariants and proved an invariant access lemma which allows opening an invariant if the current expression is atomic. In order to use invariant we only had to provide proofs for which evaluation steps are atomic. We provided proofs for all primitive evaluation steps. The proofs are the same for all steps so we just explain the one for `Load`.

```
Lemma ectx_language_atomic a e :
  head_atomic a e → sub_exprs_are_values e → Atomic a e.
```

```
Instance load_atomic v : Atomic StronglyAtomic (Load (Val v)).
Instance store_atomic v1 v2 : Atomic StronglyAtomic (Store (Val v1) (Val v2)).
...
```

An expression is atomic if it takes one step to a value, and if all subexpressions are already values. The first condition follows by definition of the step relation and the second follows by case analysis of the expression.

Since performing an effect starts a chain of evaluation steps to capture the current continuation, it is not atomic. For the same reason an effect handler and invoking a continuation are not atomic except in degenerate cases. Therefore, invariants and effects do not interact in any interesting way.

[TODO How we add support for the `iInv` tactic to use invariants more easily.]

Adding Multi-Threading to Hazel

To allow reasoning in Hazel about multi-threaded programs we need a multi-threaded operational semantics as well as specifications for the new primitive operations `Fork`, `Cmpxcgh` and `FAA`.

The language interface of Iris provides a multi-threaded operational semantics that is based on a thread-pool. The thread-pool is a list of expressions that represents threads running in parallel. At each step, one expressions is picked out of the pool at random and executed for one thread-local step. Each thread-local step additionally returns a list of forked-off threads, which are then added to the pool. This is only relevant for the `Fork` operation as all other operations naturally don't fork off threads.

```
(e, \sigma) ->_t (e', \sigma', es')
-----
(es_1 ++ e ++ es_2, \sigma) ->_mt (es_1 ++ e' ++ es_2 + es', \sigma')
```

Heaplang implements multi-threading like this and for Hazel we do the same thing. We adapt Hazel's thread-local operational semantics to include `Fork`, `Cmpxcgh` and `FAA` operations and to track forked-off threads and get a multi-threaded operational semantics “for free” from Iris' language interface.

[TODO one of the proofs that the language interface requires was a bit tricky so include that.]

Additionally, we need to prove specifications for these three operations. `Cmpxchg` and `FAA` are standard so we will not discuss them here. The only interesting design decision in the case of Hazel is how effects and `Fork` interact. This decision is guided by the fact that in OCaml 5 effects never cross thread-boundaries. An unhandled effect just terminates the current thread. As such we must impose the empty protocol on the argument of `Fork`.

$\text{EWP } e <| \text{bot} |> \{ \text{top} \}$

$\text{EWP } (\text{Fork } e) <| \text{Phi} |> \{ x, x = () \}$

[TODO explain how that proof works.]

Using these primitive operations we can then build the standard `CAS`, `Spawn`, and `Join` operations on top and prove their specifications. For `Spawn` & `Join` we already need invariants as the point-to assertion for the done flag must be shared between the two threads.

Lemma `spawn_spec` ($Q : \text{val} \rightarrow \text{iProp } \Sigma$) ($f : \text{val}$) :

$\text{EWP } (f \#()) <| \text{bot} |> \{ \{ Q \} \} \text{-* EWP } (\text{spawn } f) \{ \{ v, \exists (l : \text{loc}), \ulcorner v = \#l \urcorner \text{* join_handle } l \ Q \} \}$.

Lemma `join_spec` ($Q : \text{val} \rightarrow \text{iProp } \Sigma$) l :

$\text{join_handle } l \ Q \text{-* EWP } \text{join } \#l \{ \{ v, Q \ v \} \}$.

Definition `spawn_inv` ($\gamma : \text{gname}$) ($l : \text{loc}$) ($Q : \text{val} \rightarrow \text{iProp } \Sigma$) : $\text{iProp } \Sigma :=$

$\exists lv, l \mapsto lv \text{* } (\ulcorner lv = \text{NONEV} \urcorner \vee$
 $\exists w, \ulcorner lv = \text{SOMEV } w \urcorner \text{* } (Q \ w \ v \text{ own } \gamma \text{ (Excl } ()))).$

Definition `join_handle` ($l : \text{loc}$) ($Q : \text{val} \rightarrow \text{iProp } \Sigma$) : $\text{iProp } \Sigma :=$

$\exists \gamma, \text{own } \gamma \text{ (Excl } ()) \text{* inv } N \text{ (spawn_inv } \gamma \ l \ Q).$

Note that for `Spawn` we must also impose the empty protocol on `f` as this expression will be forked-off.

This allows us to implement standard multi-threaded programs which also use effect handlers. For example, we can prove the specification of the function below that is based on an analogous function in `Eio` which forks a thread and runs a new scheduler inside it. Note that same as in `Eio` the function blocks until the thread has finished executing, so it should be called in separate fiber.

Definition `spawn_scheduler` : $\text{val} :=$

(λ : "f",
 let: "new_scheduler" := (λ : <>, run "f") in
 let: "c" := spawn "new_scheduler" in
 join "c")%V.

Lemma `spawn_scheduler_spec` ($Q : \text{val} \rightarrow \text{iProp } \Sigma$) ($f : \text{val}$) :

$\text{promiselnv -* EWP } (f \#()) <| \text{Coop} |> \{ \{ _, \text{True} \} \} \text{-*}$
 $\text{EWP } (\text{spawn_scheduler } f) \{ \{ _, \text{True} \} \}$.

The scheduler run and therefore also the `spawn_scheduler` function don't have interesting return values, so this part of the specification is uninteresting. What is more interesting

is that they encapsulate the possible effects the given function f performs.

C. A Note on Cancellation

- That we tried to model cancellation but the feature is too permissive to give it a specification.
- There is still an interesting question of safety (fibers cannot be added to a cancelled Switch).
But including switches & cancellation in our model would entail too much work so we leave it for future work.