Verifying an Effect-Based Cooperative Concurrency Scheduler in Iris

Adrian Dapprich
Department of Computer Science
Saarland University

Advisors: Prof. Derek Dreyer & Prof. François Pottier

February 29, 2024

Contents

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 user 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.
  - Explain how performing and handling effects is implemented using delimited continuations.
  - 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 Heaplang language for Iris does not support effect handlers.
  - Hazel is an Iris language formalizing effect handlers using protocols.
  - Syntax and semantics of 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 pasing 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.

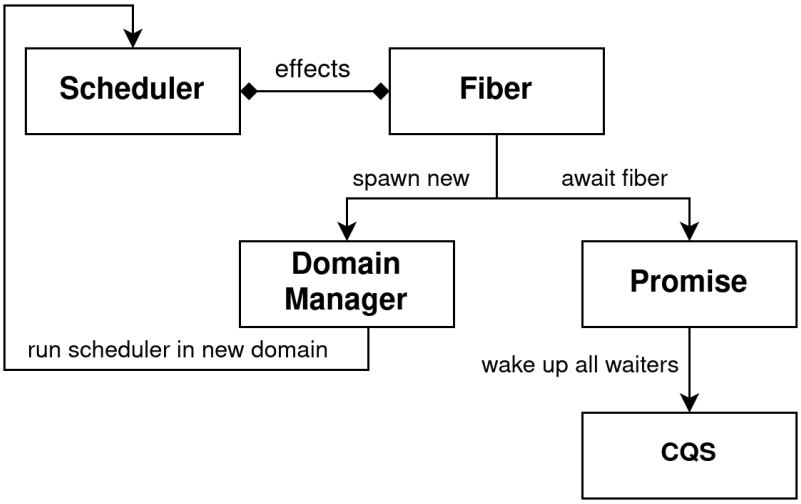

Figure 1: Eio Module Hierarchy

Figure 1 shows the simplified module hierarchy of the concepts we focus on. A standard arrow stands for a direct source code dependency from one module to another. The diamond arrow between `Scheduler` and `Fiber` stands for the implicit dependency that code in the fiber module performs effects that are handled by code in the scheduler module.

Fibers can fork off new fibers using the *Fork* effect and suspend execution using the *Suspend* effect, which are both handled by the scheduler. The implementation of the fiber and scheduler functions are discussed in section 2.1. *Promises* are built on top of the *CQS* datastructure, which is a lock-free condition variable that is used by fibers to suspend execution until a promise is fulfilled. The specification of promises is discussed in section 2.2. 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 section 3. Fibers in Eio also have access to *thread-local variables* by performing a *GetContext* effect, which is discussed in 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 in the reminder of the main text.

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 that use both **multi-threading** and **effect handlers**.

2 Verifying a Simplified Eio Scheduler

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 for user-level threads (i.e. *fibers*) are commonly treated in the literature on effect handlers [2, 4, 5] because they are a lucid example for their usefulness. Generally, the architecture contains an effect handler as the scheduler and fibers are normal functions which perform effects to yield execution. This is because performing an effect causes execution to jump to the enclosing effect handler (i.e. the scheduler), providing it with the rest of the fiber's computation in the form of a delimited continuation. The scheduler keeps track of a collection of these continuations and by invoking one of them the next fiber is scheduled. This approach is also used in Eio.

We can therefore use the simple cooperative concurrency scheduler case study from the dissertation of de Vilhena [1] as a starting point for our verification work. In the following section we first discuss the implementation of our simplified model of Eio in more detail. Using this implementation we give an intuition about what specifications the functions should satisfy and what kind of logical state is needed to prove these specifications. Based on this intuition we will then build a formalization in section 2.2.

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`[1] is the main entry point to Eio. It runs a scheduler and is provided a function which represents the first fiber to be executed. A scheduler runs the main fiber and all forked-off fibers in a single thread. However, a fiber can also spawn new schedulers in separate threads to run other fibers in parallel as detailed in appendix B. The `Fiber.fork_promise` function is used to spawn fibers in the current scheduler. The function returns a promise holding the eventual return value of the new fiber. The promise is thread-safe so that it can be shared with fibers running in different threads. The `Promise.await` function can be used by any fiber to wait until the value of a promise is available. Common problems like deadlocks are not prevented in any way and are the responsibility of the programmer.

```
(* Basic interface of the Eio library. *)
Scheduler.run : (() -> 'a) -> 'a option
Fiber.fork_promise : (() -> 'a) -> 'a Promise.t
Promise.await : 'a Promise.t -> 'a
```

We present code examples in a simplified syntax[2] because the concrete syntax of effect handlers in OCaml 5 is verbose. We use an overloading of the `match` expression, which includes cases for handled effects, that is common in the literature.

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

(* Evaluates the expression e and if the effect E is performed
 * control is transferred to the second branch.
 * The continuation k captures the rest of the computation of e.
 * The match acts as a deep handler, i.e. even if during the
 * evaluation of e the effect E is performed multiple times, the
 * second branch will be evaluated every time.
 * When e is reduced to a value, the non-effect branches are
 * used for pattern matching as usual. *)
match e with
| v -> ...
| effect (E v) k -> ...
```

2.1.1 `Scheduler.run`

As mentioned above this is the main entry point to the Eio library and its code is shown in figure 2. It sets up the scheduler environment and runs the main fiber that is passed as an argument.

The `run_queue` contains closures that will immediately invoke the continuation of an effect. This represents ready fibers which can continue execution from the point where they performed an effect. The `next` function pops one fiber (i.e. function) from the `run_queue` and executes it. If no more ready fibers

[1] The scheduler's result is optional because the main fiber might deadlock.

[2] This syntax is planned to be implemented in OCaml 5 in the future: https://github.com/ocaml/ocaml/pull/12309

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

let run (main : () -> 'a) : option 'a =
  let run_queue = Queue.create () in
  let next () =
    match Queue.pop run_queue with
    | None -> ()
    | Some cont -> cont ()
  let rec execute fiber =
    match fiber () with
    | () -> next ()
    | effect (Fork fiber) k ->
      Queue.push run_queue (fun () -> invoke k ());
      execute fiber
    | effect (Suspend register) k =>
      let waker = fun v -> Queue.push run_queue (fun () -> invoke k v) in
      register waker;
      next ()
  in
  let result = ref None in
  execute (fun () -> result := main ());
  !result
```

Figure 2: Implementation of `Scheduler.run`

remain – either because all fibers terminated or there is a deadlock – the next function just returns and the scheduler exits.

The inner `execute` function is called once on each fiber to evaluate it and handle any performed effects.

Value Case

The only non-effect case of the `match` just runs the next fiber because there are two types of fibers and their return value is always `()`.

- The main fiber is wrapped in a saves its return value in a reference and returns `()`.
- All other fibers are forked using `Fiber.fork_promise`, which wraps them in a closure that saves their return value in a promise and returns `()`.

This emphasizes the fact that an Eio scheduler is only used for running fibers. The interaction between fibers waiting for values of other fibers is handled separately by promises.

Fork Case

Handling a *Fork* effect is simple because it carries a new `fiber` to be executed, so the handler recursively calls the `execute` function to execute it immediately. The execution of the original fiber is paused due to performing an effect and its continuation `k` is placed in the run queue so that it can be scheduled again. This prioritizes the execution of a new fiber and is a design decision by Eio. It would be equally valid to place the `fiber` argument in the run queue instead.

Suspend Case

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 until a condition is met. The fiber defines this condition by constructing a `register` function which in turn receives a wake-up capability by the scheduler in form of the `waker` function. The key point is that as long as the continuation `k` is not invoked, the fiber will not continue execution. The `waker` function places `k` into the run queue so that the fiber can continue execution by a call to the scheduler's `next` function. The `register` function is called by the scheduler right after the fiber suspends execution and is responsible for installing `waker` as a callback at a suitable

place (or even call it directly). For example, to implement promises, the `waker` function is installed in a data structure that will call `waker` after the promise is fulfilled.

Note that the `waker` function's argument `v` has a *locally abstract type*, which is a typical pattern in effect handlers. From the point of view of the fiber, the polymorphic type `'a` of the *Suspend* effect is instantiated depending on how the effect's return value is used. But the scheduler does not get any information about this so the argument type of the continuation `k` and the `waker` function is abstract.

Waking up must be possible across thread boundaries, which is why the `run_queue` in the scheduler is thread-safe queue. For the verification we assume the specification of a suitable `Queue` module that supports thread-safe push and pop operations.

2.1.2 `Fiber.fork_promise`

```
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 "impossible"
    | Waiting cqs ->
        Atomic.set p (Done result);
        CQS.signal_all cqs
  in
  perform (Fork fiber)
  p
```

Figure 3: Implementation of `Fiber.fork_promise`

This function is the basic way to create a new fiber in Eio and the only one we model in our case study. The code is presented in figure 3. 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.

2.1.3 `Promise.await`

This is the most complicated looking function in our case study which is partly due to the *Suspend* effect and also due to the use of CQS functions. The code is presented in figure 4. The purpose of `Promise.await p` is to suspend execution of the calling fiber until `p` is fulfilled with a value and then return this value. The "suspend execution" part is handled by performing the *Suspend* effect. Then, the "until `p` is fulfilled" part is implemented by using CQS [3] functions as described in the following.

CQS is an implementation of the observer pattern and functionally similar to condition variables[3] in languages like C++ (as defined by the POSIX standard), allowing fibers to register callbacks that will be called when a condition is signalled. The difference is that traditional condition variables are always used together with a mutex to enable synchronization between different threads, while CQS is a lock-free data structure implementing a similar API.

In figure 5 show the public API of the CQS module. The implementation and specification will be expanded upon in section 3.

In the `Promise.await` function if the promise is not fulfilled initially the fiber should wait until that is the case, so it performs a *Suspend* effect. It defines a `register` function that registers the `waker` function with CQS by using `CQS.register`. When at some point the `CQS.signal_all` function is called – this happens in `Fiber.fork_promise` – all registered `wakers` will be called in turn. Recall that calling a `waker` function will enqueue the fiber that performed the *Suspend* effect in the scheduler's run queue so that it can continue execution.

In the default case the following simplified chain of events happens:

[3] https://en.cppreference.com/w/cpp/thread/condition_variable

```
1  type 'a t = Done of 'a | Waiting of CQS.t
2
3  let create () : 'a t =
4    let cqs = CQS.create () in
5    Atomic.create (Waiting cqs)
6
7  let make_register (p: 'a t) (cqs: CQS.t) : (() waker -> ()) =
8    fun waker ->
9      let register_result = CQS.register cqs waker in
10     match register_result with
11     | None -> ()
12     | Some register_handle ->
13       match Atomic.get p with
14       | Done result ->
15         if CQS.try_cancel register_handle
16         then waker ()
17         else ()
18       | Waiting _ -> ()
19
20 let await (p: 'a t) : 'a =
21   match Atomic.get p with
22   | Done result -> result
23   | Waiting cqs ->
24     let register = make_register p cqs
25     perform (Suspend register);
26     match Atomic.get p with
27     | Done result -> result
28     | Waiting _ -> error "impossible"
```

Figure 4: Implementation of `Promise.await`

```
1  type callback = () -> ()
2  type register_handle
3
4  val create : () -> t
5  val register : t -> callback -> register_handle option
6  val try_cancel : register_handle -> bool
7  val signal_all : t -> ()
```

Figure 5: Interface of the `CQS` module.

1. The fiber suspends execution at the point of evaluating `perform (Suspend register)`.
2. The `waker` function is registered with CQS.
3. The promise is fulfilled.
4. The `waker` function is called.
5. The fiber resumes execution at the point of evaluating `perform (Suspend register)`.

Therefore, after the *Suspend* effect returns we know the state of the promise is `Done` and the final value can be returned.

But because CQS is lock-free and promises can be shared between different threads there are a number of possible interleavings that the `register` function must take care of as well. The definition of the register function is interesting enough that we split it out into `make_register` and give a separate specification, even though it is not part of the public API of the module. First, there could be a race on the state of the promise itself. Right after the state is read in line 21 another thread might change the state to `Done` and go on to call `CQS.signal_all`. If that happens there is another race between the `CQS.register` in line 9 and the `CQS.signal_all` in the other thread. If `CQS.register` notices that there is a racing `CQS.signal_all` it will directly call the `waker`[4]. 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 `waker` would be ”lost” in the CQS,

[4]TODO mention that this is just an optimization.

never to be called. To avoid this, `register` must check the state of the promise again in line 13, and if it is fulfilled try to cancel the `waker` registration. The cancel will fail if the `waker` function was already called. If it succeeds the `register` function has the responsibility of calling `waker` itself, which is done in line 16.

The only **safety** concerns in the above implementation are `Fiber.fork_promise` expecting the promise to be unfulfilled after the fiber has finished execution, and `Promise.await` expecting the promise to be fulfilled in the last match. In both cases, the program would crash (signified by the `error` expression) if the expectation is violated. So to establish the safety of Eio we wish to prove that the expectations always hold, and the two `error` expressions are never reached. In the next section we show how the first situation 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 Specification

To prove specifications for an effectful program in Hazel we have to define not only ghost state constructs to track program state as usual but also protocols which describe the behavior of the program's effects. For our Eio case study we adapt both the ghost state and the effect protocols from the cooperative concurrency scheduler case study from chapter 4 of de Vilhena's dissertation [1].

2.2.1 Protocols

First we look at the protocols for the *Fork* and *Suspend* effect that are shown in figure 6 In Hazels' protocol syntax they are formalized in the following way, where the precondition of *Suspend* is given the name *isRegister* to describe the behavior of the fiber-defined `register` function.

$$
\begin{aligned}
\mathit{Coop} \triangleq \quad & \mathit{Fork}\ \#\ !\ e\ (e)\ \{\triangleright\ \mathit{ewp}\ (e)\ \langle \mathit{Coop}\rangle\ \{\top\}\}.?\ ()\ \{\top\} \\
& \mathit{Suspend}\ \#\ !\ \mathit{reg}\ P\ (\mathit{reg})\ \{\mathit{isRegister}\ \mathit{reg}\ P\}.?\ y\ (y)\ \{P\ y\} \\
\mathit{isRegister}\ \mathit{reg}\ P \triangleq \quad & \forall \mathit{waker}.\ \left(\forall v.\ P\ v \mathrel{-\!\!*} \mathit{ewp}\ (\mathit{waker}\ v)\ \langle\bot\rangle\ \{\top\}\right) \mathrel{-\!\!*} \triangleright\ \mathit{ewp}\ (\mathit{reg}\ \mathit{waker})\ \langle\bot\rangle\ \{\top\}
\end{aligned}
$$

Figure 6: Definition of *Coop* Protocol with *Fork* & *Suspend* Effects.

The *Fork* effect accepts an arbitrary expression *e* which represents the computation that a new fiber executes. To perform the effect one must prove that *e* acts as a function that can be called on unit and obeys the *Coop* protocol itself. This means spawned off fibers can again perform *Fork* and *Suspend* effects. The $\mathit{ewp}\ (e)\ \langle \mathit{Coop}\rangle\ \{\top\}$ is guarded behind a later modality because of the recursive occurrence of the *Coop* protocol. Since promise handling is done entirely in the fibers and the *Fork* effect just hands off the fiber to the scheduler, the protocol is simplified in two ways compared to the original from the case study of de Vilhena. First, the scheduler does not interact with the return value of the fiber, so the *ewp* has a trivial postcondition. Second, because the scheduler does not create and return the promise, the protocol itself also has a trivial postcondition.

From the type of the *Suspend* effect we already know that some value can be transmitted from the party that calls the `waker` function to the fiber that performed the effect. The *Suspend* protocol now expresses the same idea on the level of resources. To suspend, a fiber must supply a function `register` that satisfies the *isRegister* predicate. This predicate says that `register` must be callable on a `waker` function and in turn gets to assume that the `waker` function is callable on an arbitrary value `v`, which satisfies the predicate `P`. Both `register` and `waker` must not perform effects. The predicate `P` appears twice in the definition of the protocol, once in the precondition of `waker` and then in the postcondition of the whole protocol. It signifies the resources that are transmitted from the party that calls the `waker` function to the fiber that performed the effect.

By appropriately instantiating `P`, we can enforce that some condition holds before the fiber can be signalled to continue execution, and we get to assume the resources `P v` for the rest of the 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 a resource that says the promise is fulfilled.

$$\text{PS-Create}\quad \frac{}{\dot{\models}\exists\gamma.\ \mathit{promiseWaiting}\ \gamma * \mathit{promiseWaiting}\ \gamma} \qquad \text{PS-Combine}\quad \frac{\mathit{promiseWaiting}\ \gamma * \mathit{promiseWaiting}\ \gamma}{\mathit{promiseDone}\ \gamma}$$

$$\text{PS-Contra}\quad \frac{\mathit{promiseWaiting}\ \gamma * \mathit{promiseDone}\ \gamma}{\bot}$$

Figure 7: Rules for the *PromiseState* Resource

.

2.2.2 Logical State

The most basic ghost state we track is whether a promise is fulfilled or not. If a promise `p` is unfulfilled, two copies of *promiseWaiting* γ exist, one owned by the fiber and one by the invariant that tracks the state of all promises. When fulfilling the promise, both copies can be combined and converted to a persistent *promiseDone* γ resource. The *promiseWaiting* γ and *promiseDone* γ resources cannot exist at the same time. This design allows us to deduce the current state of the promise depending on if we own a *promiseWaiting* γ or a *promiseDone* γ. This is formalized in the rules in figure 7.

Maybe use meta_tokens to hide gamma

Other pieces of ghost state are *PromiseInv*, *isPromise*, and *Ready* described in figure 8. *PromiseInv* tracks additional resources for all existing promises by using an authoritative map which contains for each promise: a location p holding its current program value, a ghost name γ that is used for the *promiseWaiting* γ and *promiseDone* γ resources, and a predicate Φ that describes the value the promise will eventually hold.

Additionally, for each promise in the map we own some resources as part of *PromiseInv* that depend on the current state of the promise. As long as the promise is not fulfilled we own a broadcast, one copy of *promiseWaiting* γ, and a *signalAllPermit*. The *signalAllPermit* is used to call the `CQS.signal_all` function which must only be called once. When the promise has been fulfilled, we instead own a *promiseDone* γ and the knowledge that the final value satisfies the given postcondition Φ.

Establish *broadcast* as a name for `CQS`

isPromise is a persistent resource that denotes that p exists as a promise in *PromiseInv*. The *pn* ghost name is globally unique and included in the resource algebra we use for the proofs.

The *Ready* predicate describes fibers in a scheduler's `run_queue`. It expresses that f is safe to be executed and is used as the invariant for a scheduler's run queue, i.e. it should hold for all fibers in the run queue that they can be executed.

$$\begin{aligned}
\mathit{PromiseInv} \triangleq \exists M.\ & \boxed{\bullet\ M}^{pn} * \\
& \forall\,(p,\gamma) \mapsto \Phi \in M. \\
& \quad (\exists\, v.\ p \mapsto \mathit{Done}\ v * \mathit{promiseDone}\ \gamma * \square\,\Phi\ v) \\
& \quad \vee\,(\exists\, \mathit{cqs}.\ p \mapsto \mathit{Waiting}\ \mathit{cqs} * \mathit{isCQS}\ \mathit{cqs} * \mathit{promiseWaiting}\ \gamma * \mathit{signalAllPermit}) \\
\mathit{isPromise}\ p\ \Phi \triangleq \exists\gamma.\ & \boxed{\circ\ \{[(p,\gamma) \mapsto \Phi]\}}^{pn} \\
\mathit{Ready}\ f \triangleq\ & \mathit{ewp}\ (f\ ())\ \langle\bot\rangle\ \{\top\}
\end{aligned}$$

Figure 8: Logical State Definitions for the Verification of Scheduler & Promise Modules

In the next sections we discuss the specifications we proved for the three functions. We show a detailed proof of the specification only for `Promise.await` because it is the most involved.

2.2.3 `Scheduler.run`

The interesting part about the scheduler specification is that it proves **effect safety** of the runtime, i.e. no matter what a fiber does it will not crash the scheduler due to an unhandled effect. This is expressed by allowing the fiber *main* to perform effects according to the *Coop* protocol, but running the scheduler on the main fiber (*run main*) obeys the empty protocol, so no effects escape.

$$\textsc{Spec-Run}\ \frac{\mathit{ewp}\ (\mathit{main}\ ())\ \langle \mathit{Coop} \rangle\ \{\top\}}{\mathit{ewp}\ (\mathit{run}\ \mathit{main})\ \langle \bot \rangle\ \{\top\}}$$

However, the specification only talks about effect safety and not about handling fibers correctly in any other way, e.g. regarding fairness of scheduling or just not dropping fibers. For example, a trivial *run* function which ignores the *main* argument and immediately returns satisfies the same specification. For a scheduler it would be desirable to prove these properties, too, but since they are liveness properties it is hard to do in Iris and not a focus of this thesis.

Explain why it's hard

2.2.4 `Fiber.fork_promise`

For this specification, the *PromiseInv* argument is needed to interact with promises and the *ewp* proves that the new fiber is safe to execute and obeys the `Coop` protocol. In return, the caller gets a promise that will eventually hold a value satisfying the predicate Φ.

```
Lemma ewp_fork_promise (f: val) Φ :
  promiseInv ∗ EWP f #() <| Coop |> {{v, □ Φ v}}
⊢
  EWP (fork_promise f) <| Coop |> {{ y,
    ∃ (p: loc), ⌜ y = #p ⌝ ∗ isPromise p Φ}}.
```

The proof proceeds as follows: - First, a new promise is created, which updates the *PromiseInv* invariant and yields one half of the *promiseWaiting* γ resource for that new promise. - We define the actual fiber and prove its *ewp*. - Evaluating `f` yields a value satisfying Φ as given by the *ewp*. - Because we own *promiseWaiting* γ the second branch of the match can be ruled out. Now the *PromiseInv* invariant is accessed to update the promise state to `Done`. This consumes both halves of the *promiseWaiting* γ resource and yields a *promiseDone* γ. We also take out the *signalAllPermit*. - We use this permit along with *promiseDone* γ to call `CQS.signal_all`. *promiseDone* γ is persistent so it can be used to call all wakers. - Using the *ewp* for the wrapped `f` we can perform a *Fork* effect. - Since the promise will be fulfilled with a value satisfying Φ we have the `isPromise p Φ` that we must return.

2.2.5 `Promise.await`

The implementation of `Promise.await` is very different from the original but still satisfies the same specification. *PromiseInv* and *IsPromise* are both needed to interact with the promise's state.

```
Lemma ewp_await_make_register (p: loc) (cqs: val) Φ:
  isPromise p Φ ∗ is_cqs cqs
⊢
  EWP (make_register #p cqs) <| ⊥ |> {{register,
    ∀ (waker: val), (promise_done γ -∗ EWP waker #() <| ⊥ |> {{ ⊤ }}) -∗
    promiseInv -∗
      ▷ EWP register waker <| ⊥ |> {{ ⊤ }} }}.

Lemma ewp_await (p: loc) Φ :
  promiseInv ∗ isPromise p Φ ⊢
    EWP await #p <| Coop |> {{v, □ Φ v}}.
```

The proof of the `Promise.await` specification proceeds as follows:

- For the first match on the promise state we don't have any resources to constrain the possible results.
- If the promise is already fulfilled we can take the `Φ v` and return that.
- If it is not fulfilled, then we get access to a CQS instance and can make the `register` function using the *IsPromise* and `is_cqs` resources.
- Using the *ewp* for the `register` function we can invoke the *Suspend* effect and set `P _ := promise_done γ`.
- As a result we now have the *promiseDone* γ resource and when we match on the promise again, the unfulfilled case can be ruled out.

- So now we can take the `Φ v` and return it.

The proof of the `make_register` specification follows directly from the specifications of the CQS functions, which are explained in further detail in the next chapter.

2.2.6 Comparison of Logical State

Insert logical state from the dissertation

In de Vilhena's case study, the *Ready* predicate fulfills two roles.

1. It expresses that all continuations in the scheduler's run-queue are safe to execute.
2. It expresses that all continuations in a promise's waiting-queue are safe to execute.

PromiseInv and *isQueue* were both necessary as preconditions because they describe global state, so they had to be passed around.

In our case study *PromiseInv* was dropped from the definition of Ready because it is now put into an Iris shareable invariant, so we don't need to pass it around explicitly. Similarly, the *isQueue* precondition was dropped from the definition of *Ready* because in Eio the run queue must be thread-safe, so the *isQueue* resource is persistent and can be passed to a fiber once when it is spawned. Therefore, our *Ready* is neither recursive nor mutually recursive with *PromiseInv* anymore, which simplifies its usage in Iris. We note that the (mutual) recursion was only necessary because *PromiseInv* was used to track global state but was not put into an Iris shareable invariant, so it had to be passed around explicitly in many places.

We also split up the two uses of *Ready* and only use it under this name for the first role. In the case of a scheduler's run-queue the `Φ v` degenerates just to `v = ()`, so we can drop both from the definition and use `()` directly. This is why in our definition of *Ready* it is only an *ewp* without preconditions.

For the second use case of describing the continuations in a promise's waiting-queue we now have another specialized version of *Ready*. A broadcast has the following invariant for all stored callbacks: `P v -∗ ewp (callback ()) <| ⊥ |> {{ ⊤ }}`. This is just *Ready* where `P v` replaces `Φ v`, which is the same `P` as in the definition of the *Suspend* effect since the callbacks in a broadcast are `waker` functions.

3 Verifying Eio's Customized CQS (WIP)

CQS [3] (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 [3] 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.

3.1 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 *signal-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 [3] 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 *signal-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 *signal-all operations*. Possible interleavings and the necessity of the *cancel operation* are explained in section 2.1.3. 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 *signal-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

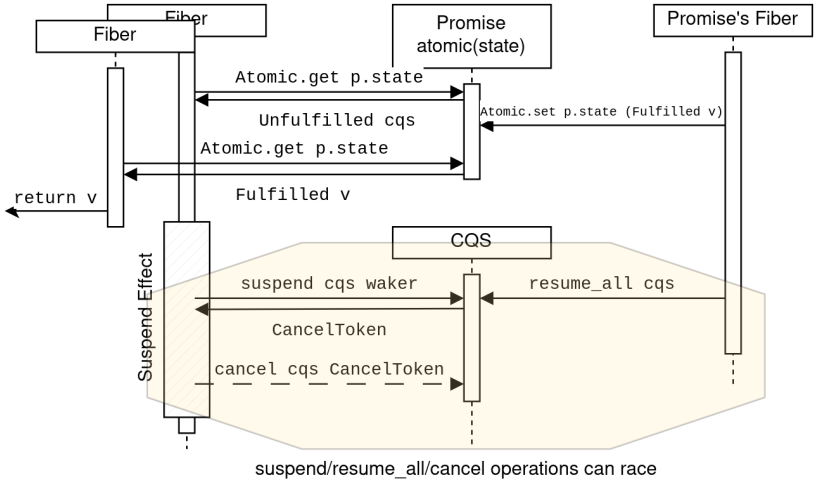

Figure 9: Usage of CQS with an Outer Atomic Variable

`suspend` returns `None` the callback has already been invoked due to a concurrent `signal_all`. A `request` value can then be used to cancel the insertion, signifying that a fiber can only cancel its own callback. The `signal_all` function (logically) consumes the CQS, which will become more clear when we present the specifications in section 3.3.

```
type t
type request

val create : unit -> t
val suspend : t -> (unit -> unit) -> request option
val cancel : request -> bool
val signal_all : t -> unit
```

Figure 10: Interface of the CQS Module

3.2 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 11.

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 *signal-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.

3.3 Verification of the `Broadcast` Module

In the following we describe the specifications we proved for the three operations `suspend`, `cancel` and `signal_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 *callbackInvokation* 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.

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

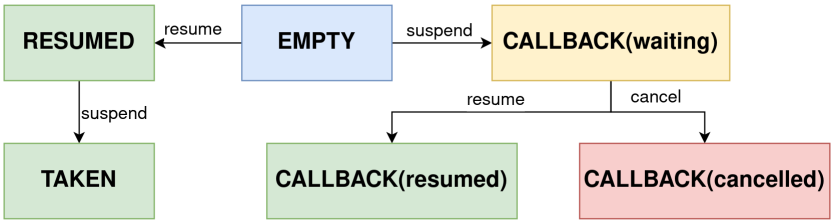

Figure 11: State Transition Diagram for a Single Cell.

3.3.1 `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 *signalAllPermit*, which is held by the enclosing promise and allows calling the `signal_all` function once.

```
Theorem create_spec:
{{{ inv_heap_inv }}}
newThreadQueue #()
{{{ γ q, RET q; is_cqs γ q ∗ signal_all_permit γ }}}.
```

3.3.2 `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' -∗ EWP k () {{ ⊤ }}` 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 `signal_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 γ q *
                is_waker V' k }}}
suspend q k
    {{{ v, RET v; ⌜v = NONEV⌝ ∨
                        ∃ γk r, ⌜v = SOMEV r⌝ *
                        is_suspend_result γ γk r k }}}.
```

3.3.3 `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 `signal_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 γ q *
                is_suspend_result γ γk r k }}}
cancel r
    {{{ (b: bool), RET #b; if b then is_waker V' k
                        else True }}}.
```

3.3.4 `signal_all`

The specification of the *signal-all operation* is also a lot simplified compared to the specification of the original *resume operation* because we removed multiple unused features. The *signalAllPermit* 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 signal_all_spec γ q:
{{{ is_thread_queue γ q *
                □ V' *
                signal_all_permit γ }}}
signal_all q
    {{{ RET #(); True }}}.
```

3.4 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.

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.

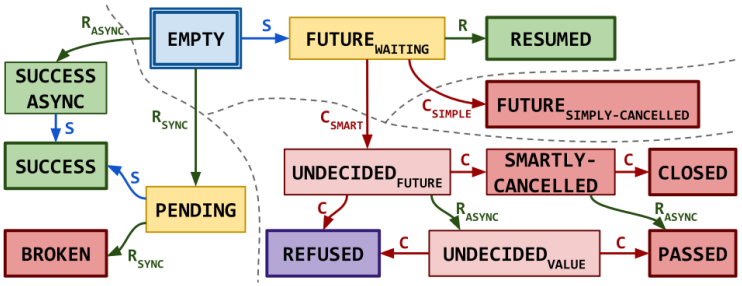

Figure 14: The state transition system for a single cell from the logical perspective.

Figure 12: Cell States in the Original CQS

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.

$$
\begin{aligned}
\mathit{Coop}\,\delta \coloneqq \quad & \mathit{Fork}\ \#\ !\ e\ (e)\ \{\triangleright\ \mathit{ewp}\ (e)\ \langle \mathit{Coop} \rangle\ \{\top\}\}.?\ ()\ \{\top\} \\
& \mathit{Suspend}\ \#\ !\ \mathit{reg}\ P\ (\mathit{reg})\ \{\mathit{isRegister}\ \mathit{reg}\ P\}.?\ y\ (y)\ \{P\ y\} \\
& \mathit{GetContext}\ \#\ !\ ()\ \{\top\}.?\ \mathit{ctx}\ (\mathit{ctx})\ \{\mathit{isFiberContext}\ \delta\ \mathit{ctx}\}
\end{aligned}
$$

Figure 13: Definition of *Coop* Protocol with *Fork* & *Suspend* Effects.

5 Evaluation

6 Conclusion

Appendix

A Translation Table

Eio	Thesis	Mechanization
`enqueue`	`waker` function	`waker`
`f`	`register` function	`register`
`Fiber.fork_promise`	`Fiber.fork_promise`	`fork_promise`
`Promise.await`	`Promise.await`	`await`
`Sched.run`	`Scheduler.run`	`run`

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 preemtively 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.

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.

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.

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*, `Cmpxchg` and `FAA` operations and to track forked-off threads and get a multi-threaded operational semantics "for free" from Iris' language interface.

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*.

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.

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.

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.

References

[1] Paulo De Vilhena. “Proof of Programs with Effect Handlers”. PhD thesis. Université Paris Cité, 2022.

[2] Stephen Dolan et al. “Concurrent system programming with effect handlers”. In: *Trends in Functional Programming: 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers 18*. Springer. 2018, pp. 98–117.

[3] Nikita Koval, Dmitry Khalanskiy, and Dan Alistarh. “CQS: A Formally-Verified Framework for Fair and Abortable Synchronization”. In: *Proceedings of the ACM on Programming Languages* 7.PLDI (2023), pp. 244–266.

[4] Daan Leijen. “Structured asynchrony with algebraic effects”. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*. 2017, pp. 16–29.

[5] Paulo Emílio de Vilhena and François Pottier. “A separation logic for effect handlers”. In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–28.