Saarland University
Department of Computer Science

Master's Thesis

Verifying an Effect-Based Cooperative Concurrency Scheduler in Iris

Author
Adrian Dapprich

Advisors
Prof. Derek Dreyer
Prof. François Pottier

Submitted: 19th April 2024

Eidesstattliche Erklärung
Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung
Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent
I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

Abstract

In this thesis we work on the formal verification of the OCaml library *Eio* which provides user-level concurrency using the new effect handlers feature of OCaml 5. As part of formal verification, the goal of program verification is to show that a program obeys a specification and is safe to execute, meaning that its execution will not run into any undefined behavior or crash. Program verification for languages with mutable state is commonly done using separation logics. For reasoning about effect handlers there exists the ML-like language *HH* and an associated program logic called Hazel, built on top of the Iris separation logic framework.

We tackle the question of safety for the central elements of the Eio library, which includes *spawning fibers* that are *run by a scheduler* and can wait for the completion of other fibers by *awaiting promises*. Therefore, our work serves as an extended case study on the usefulness of modelling and verifying programs with effect handlers in Hazel. The formal verification is carried out in the Hazel logic and our results are mechanized in the Coq proof assistant.

We were able to verify the safety of the central elements of the Eio library, and prove specifications for its public API and for the declared effects. We also extended the *HH* language to include multithreading in order to adapt previous verification work on a data structure that Eio uses.

Contents

1 Introduction

With the spread of the Internet and computers transmitting ever more data there has been a trend in programming languages to support *user-level concurrency* constructs where an application is responsible to schedule the execution of multiple *tasks* (i.e. some unit of work), analogous to how an operating system traditionally schedules multiple processes. User-level concurrency is especially beneficial when there are many small tasks that are often blocked until an I/O resource like a network socket becomes available (i.e. they are *I/O bound*). In this case the application can quickly switch to another task that is able to do work, avoiding costly jumps to kernel code and doing a context switch. Another advantage is that user-level concurrency has a lighter memory footprint. This way an application can organize many more tasks (possibly on the order of millions) than if it uses a traditional thread-per-task approach.

There is not one standardized implementation for user-level concurrency. It is generally said to use *lightweight threads* as opposed to system-level threads (provided by the operating system), but in different languages or language libraries the concept is known under terms like *async/await* (Rust, Python, JavaScript), *goroutines* (Go), and *fibers* (Java's Project Loom [16], OCaml 5's Eio).

We will look at the Eio library of OCaml 5 and formally verify the safety of its core elements for user-level concurrency. The library uses the new effect handler feature from OCaml 5 to implement fibers in an efficient way without stack copying [18]. In order to formally verify the code that uses effect handlers we use the Hazel program logic by de Vilhena [3].

Formal Verification There is a growing need for the formal verification of programs or computer systems to provide a high assurance that they are *safe to use*. Formal verification means mathematically modelling programs to enable rigorous proofs about their properties. As such, it also entails mathematically defining when a program is *safe to use*. Two important concepts behind this intuition are **safety** and **functional correctness**. By safety, we mean that when evaluating a given program according to the rules of the language it will never get into a state where there are no rules of how to evaluate it further. In some languages this is called *undefined behavior*, but we often model it as crashing the program. Safety is the baseline for the type of program verification that we do and as a next step we can show that programs are functionally correct by proving that they obey a **specification**. Specifications further restrict the possible program executions to a defined set of *good behaviors*, such as "for a given input n, this program computes the nth Fibonacci number".

Separation Logic We express specifications as logical propositions and do all reasoning in a separation logic based on *Iris* [7]. Separation logics [15] are based on Hoare logic, which defines the *Hoare triple* construct $\{P\}\, s\, \{Q\}$ to encode the specification of a program. It means that given preconditions P, execution of the program s either diverges or terminates so that the postcondition Q holds[1]. Further, separation logics are a type of affine logic that have a *separating conjunction* connective $P * Q$ in addition to the standard logical connectives.

The separating conjunction allows an interpretation of propositions as *resources* that can be split up into disjoint parts P and Q. The most prominent example of a resource is the proposition $l \mapsto v$ (also called *points-to connective*), representing a heap fragment where the location l holds the value v. This also implies *ownership* over the location l, i.e. no one else can access the location as long as we have that resource. A separating conjunction of heap fragments $l \mapsto v * l' \mapsto v'$ additionally implies that $l \neq l'$, because the heap fragments are necessarily disjoint. The dual connective of a separating conjunction is the *magic wand* $P \mathrel{-\!\!*} Q$, which states that in order to have the resources Q the resources P are still missing. It follows the elimination rule $P * (P \mathrel{-\!\!*} Q) \vdash Q$.

Another type of proposition is duplicable *knowledge*, which is also called *persistent*. For example, Hoare triples $\{P\}\, s\, \{Q\}$ are defined as persistent because under the given assumptions P the evaluation of s should always be valid.

Iris Iris [7] is a framework for building separation logics that can be instantiated with different programming languages and is implemented in the Coq proof assistant. In recent years Iris has been used in many program verification developments [5, 8, 14] as it is useful for modular reasoning about stateful and

[1] We only look at an expression-based language where Q is allowed to mention the final value of s.

multithreaded programs. Instantiating Iris with different programming languages allows layering different *program logics* on top of the base separation logic, which contain additional reasoning rules about the evaluation of programs in a concrete language. Verifying a program in Iris follows the schema of first deciding which specifications are necessary for each of the program's components. These are expressed using predicates in the logic, which we call the *logical state definitions*. If necessary, one can also use so-called *ghost state*, which is a versatile feature of Iris that allows keeping track of program state and mutating it during a proof. For complicated programs we often define ghost state and derive additional rules that modify it, in order to model the complex state space and state transitions of the program execution. Ghost state updates in Iris are restricted to happen under an *update modality* $\Rightarrow P$.

The last step in proving the program specification consists of deriving a (partial) *weakest precondition* $\mathrm{wp}\, e\, \{v.\, Q\, v\}$ for the program expression e. The weakest precondition is defined such that, if evaluation of e eventually terminates (i.e. divergence is permitted) in a final value v, it must satisfy $Q\ v$. The name is derived from the fact that it is by definition the weakest precondition P that makes the Hoare triple $\{P\}\, e\, \{v.\, Q\, v\}$ true. Hoare triples are even defined this way in Iris:

$$\{P\}\, e\, \{v.\, Q\, v\} \coloneqq \square(P \mathrel{-\!\!*} \mathrm{wp}\, e\, \{v.\, Q\, v\})$$

Since propositions are affine by default, the *persistence modality* $\square\, P$ is used to define Hoare triples as persistent. Therefore, deriving a weakest precondition for an expression e proves a specification for it in terms of the assumptions P and conclusion Q. This also establishes the safety of the expression due to a soundness lemma of the logic.

One other powerful feature of Iris are *shared invariants* $\boxed{I}^{\mathcal{N}}$, which represent knowledge that a resource does not change over time, so they are also persistent. They are used to encapsulate a resource I in order to share it under the restriction that the invariant can only be opened for one atomic step of execution at a time. If the invariant is opened, the contained resource I can be accessed but must be restored at the end of the execution step. This ensures that even in the presence of multiple threads executing in parallel, the invariant is never observably violated.

The standard language for Iris is called heaplang and is an ML-like language with mutable state and multithreading. However, it does not support effect handlers as present in OCaml 5. So for reasoning about programs with effect handlers we use the *HH* language for Iris which is based on heaplang but includes effect handlers.

Effect Handlers *Effect handlers* [13] (and the related concept of *algebraic effects*) are a versatile concept explored in some research languages [1, 10] and now also implemented in OCaml 5 [18]. They are often called *resumable exceptions* because analogous to exception handlers, one installs an effect handler around an expression `e` to handle its effects, but the effect handler also receives a delimited continuation `k`, representing the rest of the computation of `e` from the point where the effect was performed. The OCaml 5 implementation brings with it an extensible variant type `eff`, meaning one can add new constructors to the type to define effects, and a keyword `perform` to perform an effect, which transfers control to an appropriate effect handler.

We present code examples in a simplified OCaml 5 syntax[2] as shown in figure 1, because the concrete syntax of effect handlers is verbose. We use an overloading of the `match` expression, which includes cases for handled effects, that is common in the literature.

The biggest advantage of effect handlers for treating effects in a language over using monads is that they are more composable. For one, using non-monadic functions together with monadic functions often requires rewriting parts of the code into monadic style. Also, composing multiple monads results in monad transformer stacks which are notoriously confusing. Instead, effect handlers can be layered just like normal exception handlers and code written without the use of effect handlers can be used as-is.

Languages like Koka additionally track the possible effects of an expression in their type. This might be implemented for OCaml 5 in the future, but for now effects are not tracked by the type system. It is the responsibility of the programmer to install effect handlers that handle all possible effects of their program. This raises the question of **effect safety** for OCaml programs using effect handlers, which means that a program does not perform any unhandled effects. The OCaml 5 runtime treats unhandled effects as an error and crashes the program when an effect reaches the top level without being handled by an effect handler. So to prove the safety of OCaml 5 programs we must additionally establish their effect safety.

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

```
(* Declares a new constructor for the effect type
 * E : int -> bool eff *)
type _ eff += E : int -> bool eff

(* Evaluating a perform expression with a value of type 'a eff
 * transfers control to the enclosing handler and (possibly)
 * terminates in a value of type 'a. *)
let e () =
  let (b : bool) = perform (E 1) in
  b

(* Evaluates the expression e () and if the effect E is performed,
 * control is transferred to the second branch.
 * 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 is 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 -> v
(* This handler just checks if the passed value is 1.
 * The continuation must be explcitly invoked using the
 * continue expression and a value to add k to the stack.
 * Then, control is transferred back to where the perform expression
 * was evaluated. *)
| effect (E v) k -> continue k (v = 1)
```

Figure 1: Example for the effect handler syntax.

Hazel & Protocols In our development we use the Iris language *HH* and the associated program logic Hazel by de Vilhena [3, 19] which formalizes an ML-like language with effect handlers. We restate the most important concepts but for a deeper understanding we refer to [19].

Hazel defines an *extended weakest precondition ewp* $(e)\ \langle \Psi \rangle\ \{v, Q\, v\}$ which – in addition to what is implied by a normal weakest precondition – shows we can observe that the expression e performs effects according to the protocol Ψ. A protocol Ψ acts as a specification for effects in terms of their *input* and *output*, casting them in a similar light to function calls. The main way to specify a protocol is by the following constructor.

$$!\,\vec{x}\,(v)\,\{P\}.\ ?\,\vec{y}\,(w)\,\{Q\}$$

The input (!) and output (?) syntax is inspired by session types [6] which are used to describe the behavior of communicating parties. Intuitively, the part after the exclamation mark gets *sent* to the effect handler and the part after the question mark is *received* as an answer. $\vec{x}$ and $\vec{y}$ are binders whose scope extends from their position all the way to the right. The client who performs the effect transmits the value v to the effect handler and must prove the proposition P. In return, the client receives from the effect handler a value w and gets to assume Q. In total, this can be thought of as an analogue to a Hoare triple like $\{P\}\, \mathit{handler}\ v\, \{w.\ Q\ w\}$, where we explicitly name the handler that handles the effect. But the client only indirectly invokes the effect handler by evaluating a *perform* expression, so in practice we can prove Hoare triples of the following form.

$$\{P\}\, \mathit{perform}\ v\, \{w.\ Q\ w\}$$

Apart from the above there are three additional ways to define protocols. There is the sum constructor $\Psi_1 + \Psi_2$ to combine two protocols, allowing e to perform effects according to both, and its neutral element, the empty protocol $\bot$, which allows no effects. Finally, there is a tag constructor $f \,\#\, \Psi$ to give a name to protocols. Our example effect `E` from figure 1 could therefore be formalized using the following protocol $\Psi_{\mathtt{E}}$.

$$\Psi_{\mathtt{E}} \coloneqq \mathtt{E} \,\#\, !\,i\,(i)\,\{i \in \mathtt{int}\}.\ ?\,b\,(b)\,\{\mathit{if}\ i = 1\ \mathit{then}\ b = \mathit{true}\ \mathit{else}\ b = \mathit{false}\}$$

Using the extended weakest precondition with the $\bot$ protocol then enables us to prove that a program is **effect safe**, as it shows that we cannot observe any effects from the top level. Note that internally the program can of course perform effects, but an effect handler hides the effects of its discriminant expression which leads to an empty protocol at the top.

1.1 The Eio Library

We first give a general overview of the functionality provided by the Eio library before discussing what we focus on in our verification work in the next section. Eio is a library for cooperative user-level concurrency where individual tasks are represented by *fibers*[3]. Fibers are just OCaml functions that are allowed to perform a defined set of effects to interact with the cooperative scheduler. A scheduler is responsible for running an arbitrary amount of fibers in a single thread. However, if multithreading is required it is possible to spawn additional schedulers in new threads, providing some initial fiber.

In a cooperative user-level concurrency setting, many existing APIs for operating system resources in OCaml are not suitable anymore because they are blocking. Therefore, Eio also provides concurrency-aware abstractions to these resources, such as network sockets, the file system, and timers, i.e. they suspend the running fiber instead of blocking the system-level thread. Since these schedulers must interact with the operating system, there are specialized schedulers for multiple platforms such as Windows, Linux, and a generic POSIX scheduler. Eio also offers synchronization and message passing constructs like mutexes and channels which are also concurrency-aware.

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 implementing 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 platform, 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 operating system resources but just schedules fibers.
2. Awaiting the result of other fibers using the *promise* synchronization construct.
3. And spawning new schedulers to run fibers in another thread.

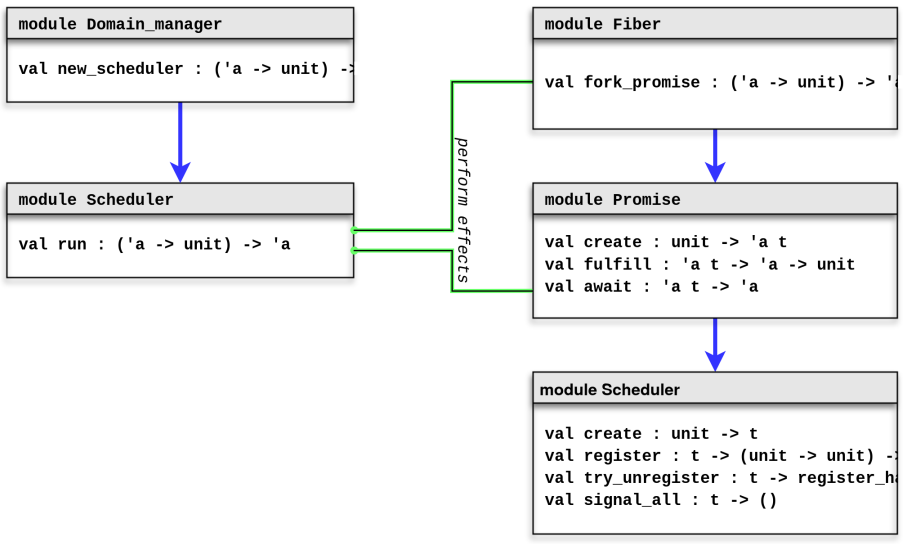

Figure 2: Eio module hierarchy.

Figure 2 shows the simplified module hierarchy of the concepts we focus on. A standard blue arrow stands for a direct source code dependency from one module to another. The highlighted green arrows

[3]Note that these are technically different from the existing fiber concept in OCaml 5, where a fiber denotes a stack frame under an effect handler, and the runtime stack is a linked list of those fibers. But since Eio fibers are evaluated under an effect handler, they all have an associated OCaml fiber. See also: https://v2.ocaml.org/manual/effects.html#s:effects-fibers

from the `Fiber` and `Promise` modules to the `Scheduler` module stand for the implicit dependency of fiber code performing effects which are handled by the scheduler.

Fibers can fork off new fibers using the *Fork* effect, suspend execution using the *Suspend* effect, and get access to some context data using the *GetContext* effect, all of which are handled by the scheduler they are running in. The implementation of the fiber and scheduler functions are discussed in section 2.1. *Promises* are built on top of the *broadcast* data structure, which is a lock-free signalling construct that is used by fibers to signal other fibers when they are done. The specification of promises is discussed in section 2.2. Broadcast is based on the *CQS* data structure, whose specification is already verified using Iris [9], but Eio customizes the implementation so we had to adapt the proof. We discuss this process in section 3. Then in section 4 we discuss the design of the Eio component that allows running schedulers in multiple threads. Fibers in Eio also have access to *thread-local variables* by performing a *GetContext* effect, which is discussed in section 5. They are thread-local in the sense that they are shared between all fibers of one scheduler. Finally, we discuss our addition of multithreading to the *HH* operational semantics in order to model running schedulers in different threads. This turned out to be technically trivial, so we only discuss it in appendix B and take a multithreaded 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.
- Proving reusable specifications for the main three effects of Eio: *Fork*, *Suspend*, and *GetContext*.
- An adaptation of the existing verification of CQS to the customized version used by Eio.
- Adding multithreading to the operational semantics of *HH*, which shows we can reason about programs that use both **multithreading** and **effect handlers**.

Our results are mechanized in the Coq proof assistant and are publicly available at https://gitlab.mpi-sws.org/addapp/master-thesis/.

2 Verifying a Basic Eio Scheduler

Cooperative concurrency schedulers for user-level threads (i.e. *fibers*) are commonly treated in the literature on effect handlers [4, 11, 19] because they are a good example for the usefulness of manipulating delimited continuations with effect handlers. Generally, the scheduler contains an effect handler and fibers are normal functions which perform effects to yield execution. Performing an effect causes execution to jump to the enclosing effect handler, 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 it can schedule the next fiber. This approach is also used in Eio.

In the following we define a basic model of the Eio scheduler and related data structures such as promises. Throughout the thesis we then extend this model with more features. We first discuss the implementation of our model and give an intuition about the behavior of each component in section 2.1. Based on this intuition we then build a formalization in section 2.2.

2.1 Implementation

Let us first get an idea of how the different core elements of Eio interact by looking at their types. The code we present throughout the thesis is OCaml 5 code that represents the *HH* code we verify.

```
1 (* Basic interface of the Eio library. *)
2 Scheduler.run : (unit -> 'a) -> 'a
3 Fiber.fork_promise : (unit -> 'a) -> 'a Promise.t
4 Promise.await : 'a Promise.t -> 'a
```

`Scheduler.run` is the main entry point to Eio. It runs a scheduler and is provided a function which represents main fiber. A scheduler runs the main fiber and all forked off fibers in a single thread, but we already assume that fibers can run in different threads to build key data structures in a thread-safe way.

The `Fiber.fork_promise` function is used to fork off 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 suspend execution 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.

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 3. It sets up the scheduler environment and then runs the main fiber (and every subsequent fiber) under an effect handler.

The `result` reference eventually holds the final value of the main fiber. The `run_queue` (line 8) contains closures that invoke the continuation of an effect. The closures represent ready fibers which can continue execution from the point where they performed an effect. The `next` function (line 9) pops one fiber from the `run_queue` and executes it. If no more ready fibers remain, the function will check if the main fiber has already finished execution 12 and if so it will also exit, which causes the scheduler to return the main fiber's final value (line 31). Otherwise, the main fiber's continuation exists *somewhere* – it could be deadlocked or just awaiting a promise from a different thread – so the `next` function busy loops until a fiber becomes available again. Busy looping makes sense in this case because other threads can push values into the `run_queue`. For the verification we assume the specification of a suitable `Queue` module that supports thread-safe push and pop operations and given a predicate I, maintains that all elements v in the queue satisfy $I\ v$. The inner `execute` function (line 18) is called once on each fiber to evaluate it and handle any performed effects.

Value Case

The non-effect case of the `match` (line 20) only runs the next fiber because Eio adopts the convention that all fibers return a unit value and their real return value is handled out of band.

- The main fiber is wrapped in a closure that saves its return value in a reference (line 30).

```
1  module Scheduler = struct
2    type _ eff += Fork : (unit -> unit) -> unit eff
3    type 'a waker : 'a -> unit
4    type _ eff += Suspend : ('a waker -> unit) -> 'a eff
5
6    let run init (main: unit -> 'a) : 'a =
7      let result = ref None in
8      let run_queue = Queue.create () in
9      let rec next () =
10       match Queue.pop run_queue with
11       | None -> begin
12           match !result with
13           | None -> next ()
14           | Some _ -> ()
15         end
16       | Some fiber -> fiber ()
17     in
18     let rec execute fiber =
19       match fiber () with
20       | () -> next ()
21       | effect (Fork fiber), k ->
22           Queue.push run_queue (fun () -> continue k ());
23           execute fiber
24       | effect (Suspend register), k ->
25           let waker = fun v -> Queue.push run_queue (fun () -> continue k v) in
26           register waker;
27           next ()
28     in
29     let tlv = ref init in
30     execute (fun () -> result := Some (main ()));
31     match !result with
32     | None -> error "impossible"
33     | Some result -> result
34 end
```

Figure 3: Implementation of `Scheduler.run`.

- All other fibers are forked using `Fiber.fork_promise`, which wraps them in a closure that saves their return value in a promise.

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 (line 21) is simple because it only carries a new `fiber` to be executed, so the handler recursively calls `execute` (line 23) on it. 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 (line 22). This prioritizes the execution of a new fiber and is a design decision by Eio. It would be equally valid to push the closure `(fun () -> execute fiber)` into the run queue instead, to give priority to the already running fiber.

Suspend Case

Handling a *Suspend* effect (line 24) 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 the form of a `waker` function. The key point is that as long as the continuation `k` is not invoked, the fiber does not continue execution. So the `waker` function "wakes up" a fiber by placing its continuation `k` into the run queue (line 25). The `register` function is called by the scheduler right after the fiber suspends execution (line 26) and is responsible for installing `waker` as a callback at a suitable place (or even call it directly). For example, to implement awaiting promises, the `waker` function is saved in a data structure that calls the function 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.

2.1.2 `Fiber.fork_promise`

```
1  module Promise = struct
2    type 'a state = Done of 'a | Waiting of Broadcast.t
3    type 'a t = 'a state Atomic.t
4
5    let create () : 'a t =
6      let bcst = Broadcast.create () in
7      Atomic.create (Waiting bcst)
8
9    let fulfill (p: 'a t) (result: 'a) =
10     match Atomic.get p with
11     | Done _ -> error "impossible"
12     | Waiting bcst ->
13         Atomic.set p (Done result);
14         Broadcast.signal_all bcst
15
16   (* ... *)
17 end
18
19 module Fiber = struct
20   let fork_promise (f: unit -> 'a) : 'a Promise.t =
21     let p = Promise.create () in
22     let fiber = fun () ->
23       let result = f () in
24       Promise.fulfill p result
25     in
26     perform (Fork fiber);
27     p
28 end
```

Figure 4: Excerpt of the `Promise` module & implementation of `Fiber.fork_promise`.

This function is the basic way to fork a new fiber in Eio and the only one we model in our development. The code is presented in figure 4. It creates a promise (line 21) and spawns the provided function as a new fiber using the *Fork* effect (line 26). Promises are always created in a `Waiting` state (we also say *unfulfilled*) and calling `Promise.fulfill` sets it to the `Done` state, at which point the final value can be retrieved. When `f ()` is reduced to a value `result`, the promise is fulfilled with that value (line 24), which signals all fibers waiting for that result to wake up (line 14). The meaning of the `Broadcast.t` contained in a promise is explained in the next section.

2.1.3 `Promise.await`

This is the most complex looking function in our development which is partly due to the *Suspend* effect and also due to the use of *broadcast* functions. Its code is presented in figure 5. 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 a *Suspend* effect. Then, the "until `p` is fulfilled" part is implemented by using the *broadcast* data structure.

In Eio, a broadcast is an implementation of a signalling mechanism used for similar purposes as condition variables in various languages. The major differences are that a broadcast does not use a mutex (it is a *lock-free* data structure) and that callers do not directly suspend execution if the condition is not met, but supply a callback that will be called when the condition is signalled.

In figure 6 we show the public API of Eio's `Broadcast` module. The `Broadcast.register` function attempts to register a given `callback` with the data structure while `Broadcast.signal_all` calls all registered callbacks. For `Broadcast.register`, a return value of `Invoked` means that it already called the supplied callback because the function detected the signal while it was running. Otherwise, a return

```
1  module Promise = struct
2    let make_register (p: 'a t) (bcst: Broadcast.t) : (unit waker -> unit) =
3      fun waker ->
4        let register_result = Broadcast.register bcst waker in
5        match register_result with
6        | Invoked -> ()
7        | Registered register_handle ->
8          match Atomic.get p with
9          | Done result ->
10             if Broadcast.try_unregister register_handle
11             then waker ()
12             else ()
13         | Waiting _ -> ()
14
15   let await (p: 'a t) : 'a =
16     match Atomic.get p with
17     | Done result -> result
18     | Waiting bcst -> begin
19         let register = make_register p bcst in
20         perform (Suspend register);
21         match Atomic.get p with
22         | Done result -> result
23         | Waiting _ -> error "impossible"
24       end
25
26   (* ... *)
27 end
```

Figure 5: Implementation of `Promise.await`.

value of `Registered` means that the callback was registered. A registered callback can be unregistered by calling `Broadcast.try_unregister`, which returns a boolean indicating the cancellation status. If the cancellation was successful, the previously registered callback is not called when `Broadcast.signal_all` is executed. The specifications of the functions is explained in more detail in section 3.3, for now we just explain their usage in the context of `Promise.await`.

```
1  type t
2  type callback = unit -> unit
3  type register_handle
4  type register_result = Invoked | Registered of register_handle
5
6  val create : unit -> t
7  val register : t -> callback -> register_result
8  val try_unregister : register_handle -> bool
9  val signal_all : t -> unit
```

Figure 6: Interface of the `Broadcast` module.

In the `Promise.await` function if the promise is not fulfilled initially (figure 5 line 18) then the fiber should wait until that is the case, so it performs a *Suspend* effect (line 20). The `register` function passed to the effect registers the `waker` function using `Broadcast.register` (line 4). When at some point the `Broadcast.signal_all` function is called – this happens in `Fiber.fork_promise` – all registered `wakers` are called in turn. Recall that calling a `waker` function enqueues 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:

1. The fiber suspends execution at the point of evaluating `perform (Suspend register)`.
2. The `waker` function is registered with a broadcast.
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, when matching on the promise state again after the *Suspend* effect returns (line 21) we know the state of the promise is `Done` and the final value can be returned.

But because broadcast is a lock-free data structure 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, which 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 (figure 5 line 16) another thread might change the state to `Done` and go on to call `Broadcast.signal_all`. If that happens there is another possible race between the call to `Broadcast.register` (line 4) and the call to `Broadcast.signal_all` in the other thread[4]. If `Broadcast.register` detects that it lost the race, it directly calls the `waker` function and returns `Invoked`. Otherwise, the `waker` function is registered but in fact the `Broadcast.signal_all` might have already finished before `Broadcast.register` even started, so it failed to detect the race. In this case the `waker` would be "lost" in the broadcast, never to be called. To avoid this, `register` must check the state of the promise again (line 8), and – if it is fulfilled – try to cancel the `waker` registration. The cancellation fails if the `waker` function was already called. Otherwise, the cancellation succeeds and the `register` function has the responsibility of calling `waker` itself (line 11).

2.1.4 Safety of the Implementation

The **safety** concerns in the above implementation are

1. `Scheduler.run` expecting the `result` reference to hold `Some` value after `execute` returns (figure 3 line 32)
2. `Fiber.fork_promise` expecting the promise to be unfulfilled after the fiber has finished execution (figure 4 line 11),
3. and `Promise.await` expecting the promise to be fulfilled in the last match (figure 5 line 23).

In all 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 `error` expressions are never reached. In the next section we show how the first two situations are addressed by defining a resource describing a one-shot assignment to a reference, and the last is a consequence of the protocol of the *Suspend* effect.

2.2 Specification

To prove specifications for an effectful program using Hazel, in addition defining to ghost state constructs for describing the program state space, we also need to define protocols that describe the behavior of the program's effects. For our Eio development we modify the ghost state and the effect protocols from the cooperative concurrency scheduler development from chapter 4 of de Vilhena's dissertation [3].

2.2.1 Protocols

The protocols for the *Fork* and *Suspend* effect are shown in figure 7. The subscripts on definitions indicate that we will change them later when extending the model.

Fork The *Fork* effect accepts a value 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 $Coop_1$ protocol itself. This means all forked off fibers can again perform *Fork* and *Suspend* effects. The weakest precondition argument is guarded behind a later modality because of the recursive occurrence of $Coop_1$.

[4]They both race to set an atomic reference holding the state of the callback registration. For more details see the implementation linked in [9].

$$
\begin{aligned}
\mathit{isWaker}\ \mathit{wkr}\ W &\triangleq \forall v.\ W\ v \mathrel{-\!\!*} \mathit{ewp}\ (\mathit{wkr}\ v)\ \langle\bot\rangle\ \{\top\} \\
\mathit{isRegister}_1\ \mathit{reg}\ W &\triangleq \forall \mathit{wkr}.\ \mathit{isWaker}\ \mathit{wkr}\ W \mathrel{-\!\!*} \mathit{ewp}\ (\mathit{reg}\ \mathit{wkr})\ \langle\bot\rangle\ \{\top\} \\
\mathit{Coop}_1 \triangleq\ & \mathit{Fork}\ \#\ !\ e\ (e)\ \left\{\triangleright\, \mathit{ewp}\ (e\ ())\ \left\langle \mathit{Coop}_1 \right\rangle\ \{\top\}\right\}.?\ ()\ \{\top\} \\
& \mathit{Suspend}\ \#\ !\ \mathit{reg}\ W\ (\mathit{reg})\ \left\{\mathit{isRegister}_1\ \mathit{reg}\ W\right\}.?\ y\ (y)\ \{W\ y\}
\end{aligned}
$$

Figure 7: Definition of Coop_1 protocol with *Fork* & *Suspend* effects.

Suspend From the type of the *Suspend* effect in figure 3 we already know that a value (of type `'a`) 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 $\mathit{isRegister}_1$ predicate. This predicate expresses that `register` can be called on a `waker` function for which we get to assume that it is callable on any value v that satisfies $W\ v$.

Both `register` and `waker` must not perform effects and are callable only once (since the *ewp* is an affine resource itself). The predicate W 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 W, we can enforce that some condition holds before the fiber can be signalled to continue execution, and we get to assume the resources $W\ v$ for the rest of the execution.

2.2.2 Logical State

The most basic ghost state we define is a variation of a *one-shot*, which we use in several places to track whether a reference l holding an optional value has been assigned to. Its rules are described in figure 8. Initially, we create two copies of $\mathit{osWaiting}\ \gamma$, which expresses that the reference holds a *None* value. One copy can be placed into an invariant that either holds an $\mathit{osWaiting}\ \gamma$ or an $\mathit{osAssigned}\ \gamma\ v$ along with the points-to connective of the reference l. Using the second copy, we can then differentiate the two cases of the invariant because the $\mathit{osWaiting}\ \gamma$ and $\mathit{osAssigned}\ \gamma$ resources cannot exist at the same time. When assigning a value v to the reference, both copies are combined and converted to a persistent $\mathit{osAssigned}\ \gamma\ v$. If the value does not matter we just write $\mathit{osAssigned}\ \gamma$.

Other pieces of ghost state are $\mathit{promiseInv}'$, *isPromise*, $\mathit{mainResult}'$, and Ready_1 described in figure 9.

$\mathit{promiseInv}'$ 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 $\mathit{osWaiting}\ \gamma$ and $\mathit{osAssigned}\ \gamma$ resources, and a predicate Φ that describes the value the promise will eventually hold. Additionally, for each promise in the map we own resources as part of $\mathit{promiseInv}'$ that depend on the current state of the promise. As long as the promise is not fulfilled we know that *bcst* is a broadcast instance, and we own one copy of $\mathit{osWaiting}\ \gamma$ and a *signalAllPermit*. The *signalAllPermit* is used to call the `Broadcast.signal_all` function which must only be called once. When the promise is fulfilled, we instead own an $\mathit{osAssigned}\ \gamma$, and we know that the final value satisfies the given postcondition Φ.

We define *promiseInv* as an invariant that contains the promise map so that we can globally share it. *isPromise* represents the knowledge that a certain promise is contained in the map of $\mathit{promiseInv}'$ and can be used to temporarily access the resources of this promise. The γ_p ghost name is globally unique to identify the global map of promises.

We take a similar approach for the result of the main fiber but this resource exists for each scheduler instead of being globally unique. $\mathit{mainResult}'\ \gamma\ l_{res}\ \Phi$ tracks the state of the location l_{res} (`result` in figure 3). The location either contains *None* or a value that satisfies the postcondition of the main fiber Φ.

The Ready_1 predicate is used as the invariant for each scheduler's `run_queue`. It is parameterized by the ghost name γ of the scheduler's $\mathit{mainResult}'$ resource. $\mathit{Ready}_1\ \gamma$ expresses that all fibers are safe to execute and will only return when the result of the main fiber has been assigned (hence the $\mathit{osAssigned}\ \gamma$). This formulation is due to the continuation passing style construction of the scheduler, which invokes a continuation at the end of the `execute` function, so the function only returns when all fibers have finished.

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.

$$\textsc{OneShotV} \triangleq \textsc{Frac} +_{\mathcal{U}} \textsc{Ag}(val) \qquad \text{Persistent}(osAssigned\ \gamma\ v)$$

$$osWaiting\ \gamma \triangleq \boxed{\tfrac{1}{2}}^{\gamma} \qquad osAssigned\ \gamma\ v \triangleq \boxed{\mathsf{ag}(v)}^{\gamma}$$

$$\textsc{OS-Create}\ \frac{}{\dot{\Rightarrow} \exists \gamma.\ osWaiting\ \gamma * osWaiting\ \gamma} \qquad \textsc{OS-Combine}\ \frac{osWaiting\ \gamma * osWaiting\ \gamma}{\Box\ osAssigned\ \gamma\ v} \qquad \textsc{OS-Contra}\ \frac{osWaiting\ \gamma * osAssigned\ \gamma\ v}{\bot}$$

Figure 8: Rules for the one-shot assignment resource.

$$\begin{aligned}
promiseState\ p\ \gamma\ \Phi \triangleq\ & (\exists bcst.\ p \mapsto Waiting\ bcst * osWaiting\ \gamma * isBroadcast\ bcst * signalAllPermit) \\
& \vee (\exists v.\ p \mapsto Done\ v * osAssigned\ \gamma * \Box\ \Phi\ v) \\
promiseInv' \triangleq\ & \exists M.\ \boxed{\bullet\ M}^{\gamma_p} * \forall (p, \gamma) \mapsto \Phi \in M.\ promiseState\ p\ \gamma\ \Phi \\
isPromise\ \gamma\ p\ \Phi \triangleq\ & \boxed{\circ\ \{[(p, \gamma) \mapsto \Phi]\}}^{\gamma_p} \\
promiseInv \triangleq\ & \boxed{promiseInv'}^{\mathcal{N}_p} \\
mainResult'\ \gamma\ l_{res}\ \Phi \triangleq\ & (l_{res} \mapsto None * osWaiting\ \gamma) \\
& \vee (\exists v.\ l_{res} \mapsto Some\ v * osAssigned\ \gamma * \Box\ \Phi\ v) \\
mainResult\ \gamma\ l_{res}\ \Phi \triangleq\ & \boxed{mainResult\ \gamma\ l_{res}\ \Phi}^{\mathcal{N}_r} \\
Ready_1\ \gamma\ f \triangleq\ & ewp\ (f\ ())\ \langle \bot \rangle\ \{osAssigned\ \gamma\}
\end{aligned}$$

Figure 9: Logical state definitions for the verification of our Eio model.

2.2.3 `Scheduler.run`

The interesting part about the scheduler specification Spec-Run is that it proves **effect safety** of the fiber 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 $Coop_1$ protocol, but running the scheduler on the main fiber (*run main*) obeys the empty protocol, so no effects escape. Of course, the *ewp* itself also implies **safety** of running both the main fiber and the scheduler.

$$\textsc{Spec-Run}\ \frac{ewp\ (main\ ())\ \langle Coop_1 \rangle\ \{v.\ \Box\ \Phi\ v\}}{ewp\ (run\ main)\ \langle \bot \rangle\ \{v.\ \Box\ \Phi\ v\}}$$

Regarding the safety of matching on the `result` reference: Because the `execute` function only returns when the main fiber has finished (so it has also assigned a value to `result`), we show that the postcondition of `execute` includes $osAssigned\ \gamma$, which allows us to refute the error branch of the final match expression since according to *mainResult'* the reference is assigned some value.

2.2.4 `Fiber.fork_promise`

The specification Spec-ForkPromise expresses that we receive from $fork\_promise$ a promise p that will eventually hold a value satisfying Φ. It has two preconditions, for one we must give it an arbitrary expression f representing the new fiber. When called on unit, f obeys the $Coop_1$ protocol and returns some value v satisfying Φ. Also, $fork\_promise$ needs the *promiseInv* invariant to interact with the global collection of promises, because it creates a new promise and fulfills it after f has finished execution.

$$\textsc{Spec-ForkPromise}\ \frac{promiseInv * ewp\ (f\ ())\ \langle Coop_1 \rangle\ \{v.\ \Box\ \Phi\ v\}}{ewp\ (fork\_promise\ f)\ \langle Coop_1 \rangle\ \{p.\ \exists \gamma.\ isPromise\ \gamma\ p\ \Phi\}}$$

2.2.5 `Promise.await`

The specification Spec-Await is the direct counterpart to Spec-ForkPromise. It shows that *await* consumes a promise p and eventually returns its value v satisfying the predicate Φ. The precondition *promiseInv* is again necessary to interact with the global collection of promises and *isPromise* is used to identify the promise p in that collection.

If p is still unfulfilled the first time *await* checks the promise state, it calls *make_register* to create a `register` function which it passes to the *Suspend* effect. As the Spec-MakeRegister specification shows, *make_register* returns a suitable function that satisfies the $isRegister_1$ predicate, instantiating W with $(\lambda\ v.\ \ulcorner v = () \urcorner * osAssigned\ \gamma)$ so that we obtain an *osAssigned* γ resource when the effect returns. This then allows us to refute the error case in the final match.

Spec-MakeRegister

$$\frac{promiseInv * isPromise\ \gamma\ p\ \Phi * isBroadcast\ bcst}{ewp\ (make\_register\ p\ bcst)\ \langle\bot\rangle\ \{reg.\ isRegister_1\ reg\ (\lambda\ v.\ \ulcorner v = () \urcorner * osAssigned\ \gamma)\}}$$

Spec-Await

$$\frac{promiseInv * isPromise\ \gamma\ p\ \Phi}{ewp\ (await\ p)\ \langle Coop_1 \rangle\ \{v.\ \square\ \Phi\ v\}}$$

In figures 10 and 11 we give Hoare-style proof annotations for the two functions *make_register* and *await*. The proof of Spec-MakeRegister uses the specifications of some broadcast functions. We briefly explain these specifications and their logical state definitions now and expand upon them in section 3.3.

$$isCallback\ cb\ R \triangleq R \twoheadrightarrow ewp\ (cb\ ())\ \langle\bot\rangle\ \{\top\}$$

$$\begin{aligned} isBroadcastRegisterResult\ r\ cb\ R \triangleq\ & (\ulcorner r = Invoked \urcorner) \\ & \vee (\ulcorner r = Registered\ h \urcorner * isBroadcastRegisterHandle\ h\ cb\ R) \end{aligned}$$

$$isBroadcastRegisterHandle : Val \to Val \to iProp \to iProp$$

Spec-BroadcastRegister

$$\frac{isBroadcast\ bcst * isCallback\ callback\ R}{ewp\ (register\ bcst\ callback)\ \langle\bot\rangle\ \{r.\ isBroadcastRegisterResult\ r\ callback\ R\}}$$

Spec-BroadcastTryCancel

$$\frac{isBroadcastRegisterHandle\ h\ cb\ R}{ewp\ (try\_unregister\ h)\ \langle\bot\rangle\ \{b.\ if\ b\ then\ isCallback\ cb\ R\ else\ \top\}}$$

The function `Broadcast.register` takes a callback *cb* that satisfies the *isCallback* predicate to register it in the broadcast data structure. This predicate is structurally similar to *isWaker* and, in fact, in the proof of Spec-MakeRegister we instantiate the precondition R with *osAssigned* γ and pass as the callback a `waker` function, which has the precondition $(\lambda\ v.\ \ulcorner v = () \urcorner * osAssigned\ \gamma)$ as described above. The result of `Broadcast.register` is either a value *Invoked*, which expresses that it called the callback directly, or a register handle, which can be used to call `Broadcast.try_unregister`.

`Broadcast.try_unregister` attempts to cancel a previous registration identified by the given *handle*. If the cancellation is successful, we receive a *isCallback* resource which shows that we can safely call the callback again.

Hoare-Style Proofs for Spec-MakeRegister and Spec-Await In the proof below an opened invariant *Inv* is represented as ~~*Inv*~~ and resources that are not needed for the rest of the proof are dropped implicitly.

The proof of Spec-MakeRegister is straightforward and follows from the specifications of `Broadcast.register` and `Broadcast.try_unregister`. For Spec-Await, the crux is that we define Spec-MakeRegister so that it returns a *register* function which satisfies $isRegister_1$ *register* $(\lambda\ v.\ \ulcorner v = () \urcorner * osAssigned\ \gamma)$. Then, we get access to the *osAssigned* γ resource when the *Suspend* effect returns, and we can refute the case of the promise still being unfulfilled when checking the state of promise again for the last time.

SPEC-MAKEREGISTER

$$\frac{\mathit{promiseInv} * \mathit{isPromise}\ \gamma\ p\ \Phi * \mathit{isBroadcast}\ \mathit{bcst}}{\mathit{ewp}\ (\mathit{make\_register}\ p\ \mathit{bcst})\ \langle\bot\rangle\ \{\mathit{reg}.\ \mathit{isRegister}_1\ \mathit{reg}\ (\lambda\ v.\ \ulcorner v = ()\urcorner * \mathit{osAssigned}\ \gamma)\}}$$

```
let make_register (p: 'a t) (bcst: Broadcast.t)
    : (unit waker -> unit) =
```
$\{\mathit{promiseInv} * \mathit{isPromise}\ \gamma\ p\ \Phi * \mathit{isBroadcast}\ \mathit{bcst}\}$

```
  fun (waker: unit waker) ->
```
[intro waker that satisfies *isWaker*]

$\{\mathit{promiseInv} * \mathit{isPromise}\ \gamma\ p\ \Phi * \mathit{isBroadcast}\ \mathit{bcst} * (\mathit{osAssigned}\ \gamma \mathrel{-\!\!*} \mathit{ewp}\ (\mathit{waker}\ ())\ \langle\bot\rangle\ \{\top\})\}$

```
    let regres = Broadcast.register bcst waker in
```
[apply SPEC-BROADCASTREGISTER with $R := \mathit{osAssigned}\ \gamma$]

$\{\mathit{promiseInv} * \mathit{isPromise}\ \gamma\ p\ \Phi * \mathit{isBroadcast}\ \mathit{bcst} * \mathit{isBroadcastRegisterResult}\ \mathit{regres}\}$

```
    match regres with
```
[case analysis on *regres*]

---

1. $\{\mathit{regres} = \mathit{None}\}$

```
    | None -> ()
```
[goal is trivial]

$\{\top\}$

---

2. $\{\mathit{promiseInv} * \mathit{isPromise}\ \gamma\ p\ \Phi * \mathit{isBroadcast}\ \mathit{bcst} * \mathit{regres} = \mathit{Some}\ \mathit{handle} * \mathit{isBroadcastRegisterHandle}\ \mathit{handle}\}$

```
    | Some handle ->
```
[open *promiseInv*, lookup p using *isPromise*]

$\{\cancel{\mathit{promiseInv}} * \mathit{isBroadcast}\ \mathit{bcst} * \mathit{isBroadcastRegisterHandle}\ \mathit{handle} * \mathit{promiseState}\ p\ \gamma\ \Phi\}$

```
      match Atomic.get p with
```
[case analysis on *promiseState*]

2.1. $\{\cancel{\mathit{promiseInv}} * \mathit{isBroadcast}\ \mathit{bcst} * \mathit{isBroadcastRegisterHandle}\ \mathit{handle} * p \mapsto \mathit{Done}\ \mathit{result} * \mathit{osAssigned}\ \gamma\}$

```
      | Done result ->
```
[close *promiseInv*]

$\{\mathit{isBroadcast}\ \mathit{bcst} * \mathit{isBroadcastRegisterHandle}\ \mathit{handle} * \mathit{osAssigned}\ \gamma\}$

```
        if Broadcast.try_unregister handle
```
[apply SPEC-BROADCASTTRYCANCEL, case analysis on return value]

2.1.1. $\{\mathit{osAssigned}\ \gamma * (\mathit{osAssigned}\ \gamma \mathrel{-\!\!*} \mathit{ewp}\ (\mathit{waker}\ ())\ \langle\bot\rangle\ \{\top\})\}$ [specialize assumption]

$\{\mathit{ewp}\ (\mathit{waker}\ ())\ \langle\bot\rangle\ \{\top\}\}$

```
        then waker ()
```
[by apply $\mathit{ewp}\ (\mathit{waker}\ ())\ \langle\bot\rangle\ \{\top\}$]

$\{\top\}$

2.1.2. $\{\top\}$

```
        else ()
```
[goal is trivial]

$\{\top\}$

2.2. $\{\cancel{\mathit{promiseInv}} * p \mapsto \mathit{Waiting}\ \_\}$

```
      | Waiting _ -> ()
```
[close *promiseInv*, goal is trivial]

$\{\top\}$

Figure 10: Annotated proof of SPEC-MAKEREGISTER.

SPEC-AWAIT

$$\frac{\mathit{promiseInv} * \mathit{isPromise}\ \gamma\ p\ \Phi}{\mathit{ewp}\ (\mathit{await}\ p)\ \langle \mathit{Coop}_1 \rangle\ \{v.\ \square\,\Phi\ v\}}$$

```
let await (p: 'a t) : 'a =
```

$\{\mathit{promiseInv} * \mathit{isPromise}\ \gamma\ p\ \Phi\}$ [open *promiseInv*, lookup p using *isPromise*]

$\{\cancel{\mathit{promiseInv}} * \mathit{isPromise}\ \gamma\ p\ \Phi * \mathit{promiseState}\ p\ \gamma\ \Phi\}$

```
  match Atomic.get p with
```
[case analysis on *promiseState*]

---

1. $\left\{\begin{array}{c}\cancel{\mathit{promiseInv}} * \\ p \mapsto \mathit{Done}\ \mathit{result} * \square(\Phi\ \mathit{result})\end{array}\right\}$

```
  | Done result ->
```
[close *promiseInv*]

$\{\mathit{promiseInv} * \square(\Phi\ \mathit{result})\}$

```
    result
```
[by assumption]

$\{\square(\Phi\ \mathit{result})\}$

---

2. $\left\{\begin{array}{c}\cancel{\mathit{promiseInv}} * \mathit{isPromise}\ \gamma\ p\ \Phi * \\ p \mapsto \mathit{Waiting}\ \mathit{bcst} * \mathit{isBroadcast}\ \mathit{bcst}\end{array}\right\}$

```
  | Waiting bcst ->
```
[close *promiseInv*]

$\left\{\begin{array}{c}\mathit{promiseInv} * \mathit{isPromise}\ \gamma\ p\ \Phi * \\ \mathit{isBroadcast}\ \mathit{bcst}\end{array}\right\}$

```
    let register = make_register p bcst
```
[apply SPEC-MAKEREGISTER]

$\left\{\begin{array}{c}\mathit{promiseInv} * \mathit{isPromise}\ \gamma\ p\ \Phi * \\ \mathit{isRegister}_1\ \mathit{register}\ (\lambda v.\ \ulcorner v = () \urcorner * \mathit{osAssigned}\ \gamma)\end{array}\right\}$

```
    perform (Suspend register);
```
[protocol of *Suspend* with $(W := \lambda v.\ \ulcorner v = () \urcorner * \mathit{osAssigned}\ \gamma)$]

$\left\{\begin{array}{c}\mathit{promiseInv} * \mathit{isPromise}\ \gamma\ p\ \Phi * \\ \mathit{osAssigned}\ \gamma\end{array}\right\}$

[open *promiseInv*, lookup p using *isPromise*]

$\left\{\begin{array}{c}\cancel{\mathit{promiseInv}} * \mathit{osAssigned}\ \gamma * \\ \mathit{promiseState}\ p\ \gamma\ \Phi\end{array}\right\}$

```
    match Atomic.get p with
```
[case analysis on *promiseState*]

2.1. $\left\{\begin{array}{c}\cancel{\mathit{promiseInv}} * \\ p \mapsto \mathit{Done}\ \mathit{result} * \square(\Phi\ \mathit{result})\end{array}\right\}$

```
    | Done result ->
```
[close *promiseInv*]

$\{\mathit{promiseInv} * \square(\Phi\ \mathit{result})\}$

```
      result
```
[by assumption]

$\{\square(\Phi\ \mathit{result})\}$

2.2. $\left\{\begin{array}{c}\cancel{\mathit{promiseInv}} * \mathit{osAssigned}\ \gamma * \\ p \mapsto \mathit{Waiting}\ \mathit{bcst} * \mathit{osWaiting}\ \gamma\end{array}\right\}$

```
    | Waiting _ ->
```
[specialize PS-CONTRA]

$\{\cancel{\mathit{promiseInv}} * \bot\}$

```
      error "impossible"
```
[by contradiction]

$\{\bot\}$

Figure 11: Annotated proof of SPEC-AWAIT.

3 Verifying Eio's Broadcast

In this section we describe the *broadcast* data structure of Eio. Broadcasts are a customization of the recently developed *CQS* data structure [9]. CQS (for CancellableQueueSynchronizer) is a lock-free synchronization primitive that allows execution contexts to wait until signalled. Its specification is already formally verified in Iris, so we were able to adapt the proofs to use them in our development[5]. CQS keeps the nature of an execution context 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. Broadcast is used in the same way, as it is used in the implementation of Eio's *promise*.

In the case of Eio an *execution context* is an Eio fiber. CQS is multithreaded by design, so fibers can use the broadcast operations to synchronize with fibers running in another thread. In the following we describe the behavior of Eio's *broadcast* and explain the specifications of the customized operations. Later we also highlight differences to the *original CQS*.

3.1 Operations of Broadcast

Broadcast has the following three main operations in its public interface: `register`, `signal_all`, and `try_unregister`. While we established Eio's broadcast as an implementation of a signalling mechanism where fibers can register callbacks to be notified about events, the original formulation of CQS uses a more abstract future-based interface for the same purpose. This is because it is assumed that the language runtime supports suspending an execution context until a future is completed. But Eio uses the broadcast data structure to **build** the runtime that allows its execution contexts (i.e. fibers) to suspend until an event happens (i.e. a promise is fulfilled), so a callback-based interface that takes `waker` function was chosen for the adapted operations.

`Broadcast.register` This operation takes a callback and saves it in the data structure, to be called when an event is signalled. Since all operations are lock-free, it can happen that a concurrent call to `signal_all` tries to modify the data structure while `register` is still active. If this case is detected, `register` immediately invokes the callback itself.

`Broadcast.signal_all` As a dual to `register`, the `signal_all` operation invokes all callbacks that are registered with the data structure. Eio uses `signal_all` instead of a `signal` operation that only invokes one callback in order to make the implementation of promises more straightforward. When a promise is fulfilled, **all** fibers waiting on its value can continue execution, so the fine-grained control of a single `signal` operation is not needed.

`Broadcast.try_unregister` The `try_unregister` operation tries to undo the registration of a callback. If the operation succeeds, the associated callback will not be invoked by `signal_all`. Otherwise, if a corresponding `signal_all` happens first, the operation fails.

To understand the broadcast operations better it is helpful to view them in the context in which they are used. Like in the original CQS, an interaction with a broadcast is always guarded by first accessing an atomic variable that holds the state of the outer synchronization construct, in this case the state of the promise. Since the whole data structure is lock-free, the atomic variable ensures that the operations have a synchronized view of the state. For example, a register operation is only attempted if the promise is not fulfilled yet. Figure 12 shows the possible interactions between fibers and a promise. The calls to `Atomic.get` and `Atomic.set` happen in the functions `Promise.await` and `Promise.fulfill`, as shown in section 2.1. If the promise is not fulfilled yet, `Promise.await` then performs a *Suspend* effect and calls `Broadcast.register` and `Broadcast.try_unregister` if necessary.

Note that because all operations are lock-free and fibers can run in different threads, there can be a race between concurrent `register`, `try_unregister`, and `signal_all` operations. Possible interleavings and the necessity of the `try_unregister` were explained in section 2.1.3.

[5] However, at the time of writing we proved our specifications in a subset of *heaplang* instead of *HH*. Since this subset is equivalent to *HH* without effects we postulate that our proofs are also valid for *HH*, but it will take some time to rewrite all proofs for *HH*.

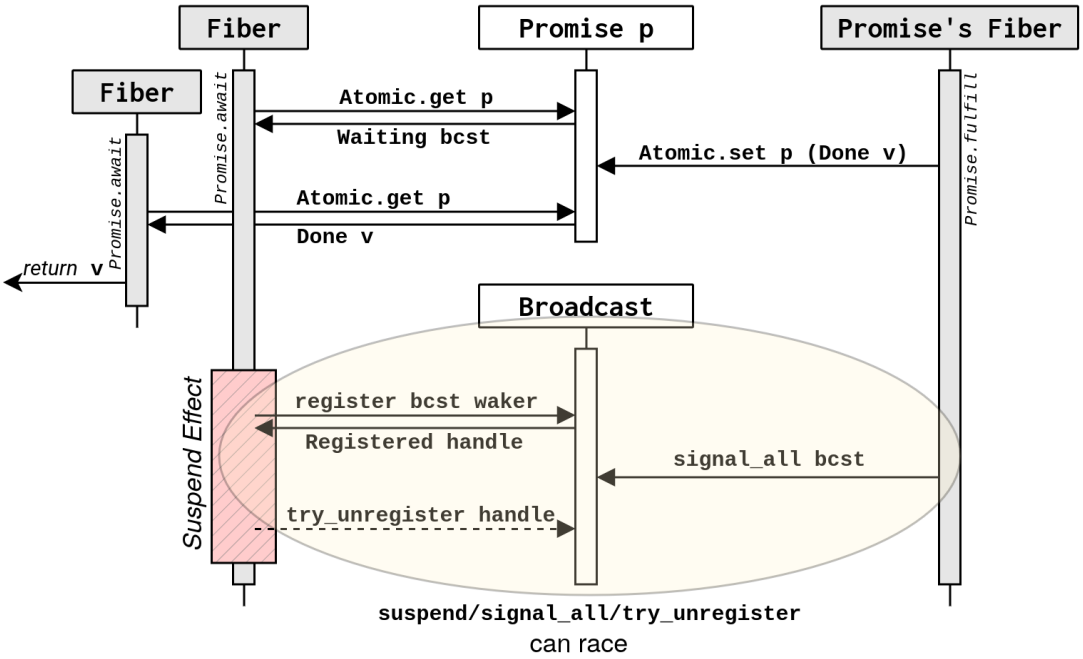

Figure 12: Usage of broadcast in the context of a promise.

3.2 Implementation and Logical Interface of Broadcast

Like the original CQS, broadcast is implemented as a linked list of arrays (called segments) that contain *cells*[6]. There are two pointers pointing to the beginning and end of the active cell range, the signal pointer and the register pointer, and cells not reachable from either pointer are garbage collected. There is a set of operations for manipulating the linked list and 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 callback and the logical state of the broadcast tracks the current state of all existing cells. The possible states for a single cell are shown in figure 13, where the arrows are annotated with the operation that causes a state transition.

The state of a cell is initialized to **EMPTY** when it is reached by the register pointer[7]. When a `register` and `signal_all` operation happen concurrently, they race to set the value of the empty cell. If the `signal_all` operation wins, it writes a token value into the cell and the state becomes **SIGNALLED**. The `register` operation can then read the token and invoke its callback directly. The state is thus **INVOKED**. If instead the `register` operation wins the race, it writes the callback into the cell, so the state takes the right path to $\textbf{CALLBACK}_{\textbf{waiting}}$. Then there can be another race between concurrent `try_unregister` and `signal_all` operations. Both try to overwrite the callback with a token value, which changes the state to $\textbf{CALLBACK}_{\textbf{invoked}}$ or $\textbf{CALLBACK}_{\textbf{unregistered}}$, respectively, depending on the winner. A `signal_all` ignores a lost race against `try_unregister` because it should disregard the callback in question, while `try_unregister` returns the outcome of the race as a boolean value. If it won, then the caller knows that it is now responsible to invoke the callback itself.

3.3 Verification of Broadcast

Note that the specifications of all broadcast operations obey the empty protocol because the code does not perform any effects. For all three operations, the Eio implementation and specification differs from what is already verified in the original CQS (e.g. due to some reordered instructions or a different control flow). However, the specifications of the underlying operations for manipulating cell pointers are modular

[6]Using segments instead of single cells in the linked list is an optimization to amortize the linear runtime of linked list operations.

[7]In the original CQS, a cell can also be initialized when it is reached by the signal pointer, but in broadcast the signal pointer never overtakes the register pointer.

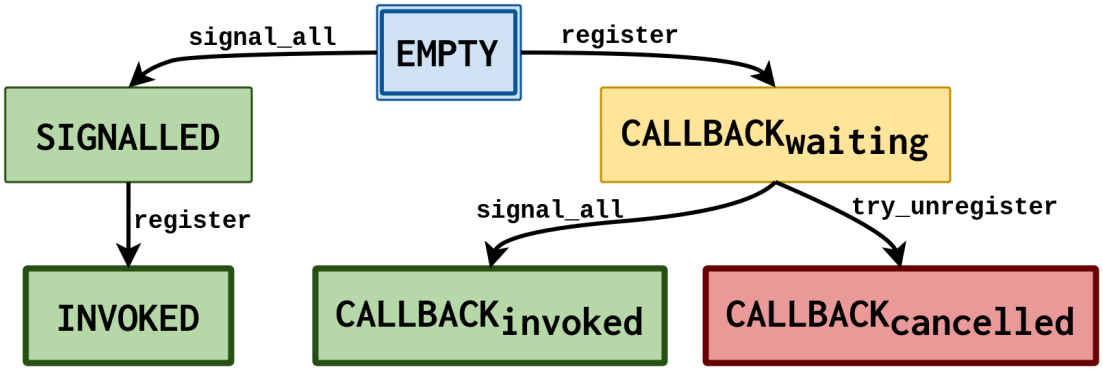

Figure 13: State transition diagram for a single cell.

enough to allow us to prove the new specifications for `Broadcast.create`, `Broadcast.register`, and `Broadcast.try_unregister`.

As for `Broadcast.signal_all`, Eio implements this function by atomically increasing the signal pointer by the number n of registered callbacks and then processing all n cells between the old and new pointer position. Because of technical differences in handling these pointers between the original CQS implementation of the paper [9] and the broadcast implementation of Eio we opted to verify a different implementation of `Broadcast.signal_all`, that increments the signal pointer n times in a loop and processes a single cell each time. We argue this does not change the observable behaviors of the function since we ensure that it can only be called once.

3.3.1 `Broadcast.create`

The only precondition to create a new broadcast is the proposition *inv_heap_inv*. This is a piece of ghost state defined by the Iris standard library that models invariant locations, which are locations that can always be read. That means they cannot be explicitly deallocated and can only exist in a garbage-collected setting, like OCaml 5. The implementation of the linked list uses this internally.

The function returns a broadcast instance *bcst*, along with the persistent *isBroadcast bcst* proposition that shows the value actually is a broadcast. We also obtain the unique resource *signalAllPermit*, which is held by the enclosing promise and allows calling the `Broadcast.signal_all` function once.

Spec-BroadcastCreate

$$\frac{\mathit{inv\_heap\_inv}}{\mathit{ewp}\ (\mathit{create}\ ())\ \langle\bot\rangle\ \{\mathit{bcst}.\ \mathit{isBroadcast}\ \mathit{bcst} * \mathit{signalAllPermit}\}}$$

3.3.2 `Broadcast.register`

A register operation takes a callback *cb* and the associated resource *isCallback cb R* which represents the permission to invoke the callback when a precondition R is fulfilled. *isCallback* is not persistent because the callback must be invoked only once.

$$\begin{aligned}
\mathit{isCallback}\ \mathit{cb}\ R &\triangleq R \twoheadrightarrow \mathit{ewp}\ (\mathit{cb}\ ())\ \langle\bot\rangle\ \{\top\} \\
\mathit{isBroadcastRegisterResult}\ r\ \mathit{cb}\ R &\triangleq \quad (\ulcorner r = \mathit{Invoked}\urcorner) \\
&\qquad \vee (\ulcorner r = \mathit{Registered}\ h\urcorner * \mathit{isBroadcastRegisterHandle}\ h\ \mathit{cb}\ R) \\
\mathit{isBroadcastRegisterHandle} &: \mathit{Val} \to \mathit{Val} \to \mathit{iProp} \to \mathit{iProp}
\end{aligned}$$

The `Broadcast.register` function tries to insert a callback into the next cell designated by the register pointer. If it succeeds the function returns a `Registered handle` value that can be used by `Broadcast.try_unregister`. But if the cell is already in the **SIGNALLED** state, the function immediately invokes the callback and returns an `Invoked` value.

Spec-BroadcastRegister

$$\frac{\mathit{isBroadcast}\ \mathit{bcst} * \mathit{isCallback}\ \mathit{callback}\ R}{\mathit{ewp}\ (\mathit{register}\ \mathit{bcst}\ \mathit{callback})\ \langle \bot \rangle\ \{r.\ \mathit{isBroadcastRegisterResult}\ r\ \mathit{callback}\ R\}}$$

3.3.3 `Broadcast.try_unregister`

Given a handle h and the associated *isBroadcastRegisterHandle h cb R* resource, `Broadcast.try_unregister` tries to cancel the registration of the callback.

If the callback had already been invoked by `Broadcast.signal_all` (i.e. the state is $\mathbf{CALLBACK}_{\mathbf{invoked}}$) the function returns `false` and no resources are returned to the caller. Otherwise, the permission to invoke the callback *isCallback cb* is returned.

Spec-BroadcastTryCancel

$$\frac{\mathit{isBroadcastRegisterHandle}\ h\ \mathit{cb}\ R}{\mathit{ewp}\ (\mathit{try\_unregister}\ h)\ \langle \bot \rangle\ \{b.\ \mathit{if}\ b\ \mathit{then}\ \mathit{isCallback}\ \mathit{cb}\ R\ \mathit{else}\ \top\}}$$

3.3.4 `Broadcast.signal_all`

To call `Broadcast.signal_all` the unique *signalAllPermit* resource is needed along with a persistent R, so that it can be used to invoke multiple callbacks. The function does not return any resources because its only effect is invoking an unknown number of callbacks, none of which return any resources themselves.

Spec-BroadcastSignalAll

$$\frac{\mathit{isBroadcast}\ \mathit{bcst} * \square\, R * \mathit{signalAllPermit}}{\mathit{ewp}\ (\mathit{signal\_all}\ \mathit{bcst})\ \langle \bot \rangle\ \{\top\}}$$

3.4 Changes from the 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 development we removed support for these features. This reduced the state space of a cell shown in figure 14 to a more manageable size when adapting the proofs.

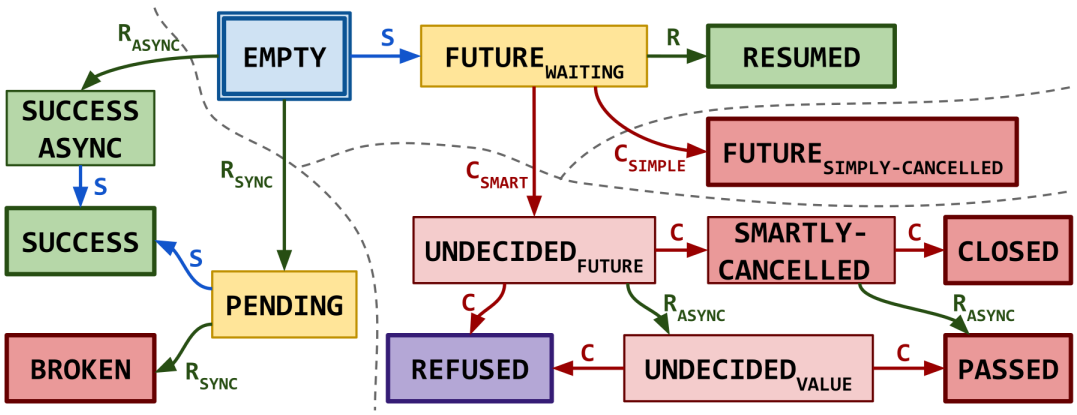

Figure 14: Cell states in the original CQS from [9] (page 42).

The part of the verification of the original CQS that we had to customize for Eio was originally 3600 lines of Coq code but – due to these simplifications – we could reduce it by approximately 1300 lines of Coq code. Additionally, there are 4000 lines of Coq code about lower-level functionality that we did not need to adapt when porting them to our development.

4 Adding Support for Multiple Schedulers

So far we have always considered the possibility of schedulers running in multiple threads when explaining design choices of Eio data structures like the run queue and promises. These additional schedulers are created using Eio's *domain manager* and in this section we discuss how we integrate the domain manager into our model. It exposes a function `Domain_manager.new_scheduler` (shown in figure 15) which, given some function `f`, forks a new thread, runs a scheduler with `f` as its main fiber and finally returns the result of `f`.

4.1 Implementation

To interact with system-level threads[8], the function uses the standard `thread_spawn` and `thread_join` functions exposed by many thread implementations, which fulfill the specifications below.

THREAD-SPAWN

$$\frac{ewp\ (f\ ())\ \langle\bot\rangle\ \{v.\ Q\ v\}}{ewp\ (thread\_spawn\ f)\ \langle\bot\rangle\ \{j.\ joinHandle\ j\ Q\}}$$

THREAD-JOIN

$$\frac{joinHandle\ j\ Q}{ewp\ (thread\_join\ j)\ \langle\bot\rangle\ \{v.\ Q\ v\}}$$

We implemented threads for the operational semantics of *HH* and proved the specifications for these functions as described in appendix B. This pair of functions is analogous to `Fiber.fork_promise` and `Promise.await` but on the level of threads. `thread_spawn` runs the given function `f` in a new thread and returns a *joinHandle* that can be used by `thread_join` to return the final result of `f ()`. The difference to fibers is that `thread_join` is considered *blocking* in the sense that the calling thread does not continue execution until the thread associated with the *joinHandle* has terminated.

```
1  let new_scheduler (f: unit -> 'a) : 'a =
2    let handle = ref None in
3    let register = (fun waker ->
4      let thread_fun = (fun () ->
5        let result = Scheduler.run f in
6        waker ();
7        result
8      ) in
9      let join_handle = thread_spawn thread_fun in
10     handle := Some join_handle
11   ) in
12   perform (Suspend register);
13   match !handle with
14   | None -> error "impossible"
15   | Some join_handle -> thread_join join_handle
```

Figure 15: Implementation of `Domain_manager.new_scheduler`.

The main complexity of the implementation of `Domain_manager.new_scheduler` comes from avoiding this blocking behavior. As this function is called by fibers, blocking the current thread would prevent the scheduler of the current thread from switching to another fiber. This situation must be avoided and is a source of great complexity when calling blocking operations from a cooperative concurrency setting.

`Domain_manager.new_scheduler` avoids blocking the current thread by suspending the calling fiber until the new thread is terminating. This is done by performing a *Suspend* effect (line 12) and forking the new thread inside the `register` function. The `thread_fun` runs the new scheduler and calls the `waker` function (line 6) only after the new scheduler has finished. The resulting join handle is saved in a reference (line 10) to be accessed after the original fiber wakes up. By the time the original fiber continues execution, the new thread will have terminated or terminate very soon, so it uses the join handle from the reference and retrieves the thread's final value (line 15) without blocking (or blocking very shortly).

[8] OCaml 5 uses the term *domain*, but we use the standard thread terminology for shared-memory execution contexts running in parallel.

4.2 Specification and Changes to Logical State

Because the function essentially delegates to `Scheduler.run`, it has the same type `(unit -> 'a) -> 'a`, and we were able to prove an analogous specification as shown below.

$$\frac{\textsc{Spec-NewScheduler}\quad ewp\ (f\ ())\ \langle Coop_2 \rangle\ \{v.\ \square\, \Phi\, v\}}{ewp\ (new\_scheduler\ f)\ \langle Coop_2 \rangle\ \{v.\ \square\, \Phi\, v\}}$$

However, we need to update several definitions to make this proof possible. One of them is the specification of the *Suspend* effect, which is why we now refer to the $Coop_2$ protocol. The match in line 13 is only safe because the reference is assigned a join handle in the `register` function which runs to completion before the effect returns. We track the status of the reference by reusing the *OneShotAssign* ghost-state from figure 8. By performing the *Suspend* effect we want to receive $osAssigned\ \gamma_{\texttt{handle}}$, where $\gamma_{\texttt{handle}}$ is specific to the `handle` reference. Consequently, we update the protocol and its related definitions (shown in figure 16) so that *Suspend* additionally returns a persistent resource S that results from calling the `register` function[9].

$$\begin{aligned}
isWaker\ wkr\ W &\triangleq \forall v.\ W\ v \mathrel{-\!\!*} ewp\ (wkr\ v)\ \langle \bot \rangle\ \{\top\} \\
isRegister_2\ reg\ W\ S &\triangleq \forall wkr.\ isWaker\ wkr\ W \mathrel{-\!\!*} ewp\ (reg\ wkr)\ \langle \bot \rangle\ \{\square\, S\} \\
Coop_2 \triangleq\ & Fork\ \#\ !\ e\ (e)\ \left\{\triangleright\, ewp\ (e\ ())\ \langle Coop_1 \rangle\ \{\top\}\right\}.?\ ()\ \{\top\} \\
& Suspend\ \#\ !\ reg\ W\ S\ (reg)\ \left\{isRegister_2\ reg\ W\ S\right\}.?\ y\ (y)\ \{W\ y * S\}
\end{aligned}$$

Figure 16: Definition of $Coop_2$ protocol with *Fork* & *Suspend* effects.

As a result of changing the effect postcondition of the effect, the continuation `k` that the effect handler receives for the *Suspend* effect now also has S as a precondition.

$$\forall v.\ W\ v \mathrel{-\!\!*} S \mathrel{-\!\!*} ewp\ (k\ ())\ \langle \bot \rangle\ \{\top\}$$

Recall that the `waker` function receives $W\ v$ and pushes `k` into the scheduler's run queue, where the queue invariant says that `k` must be callable as-is, without precondition. Since `waker` is called from a different thread – and might even be called before the `register` function finishes – it cannot supply the S. Therefore, when `k` is pushed into the run queue it is still missing the S to satisfy the queue invariant, so we must change the specification of the run queue to allow pushing elements that temporarily do not satisfy the queue invariant.

4.3 Specification for a Deferred Queue

We call such a queue a *deferred queue*, because the queue invariant can be temporarily violated but must be repaired eventually. We proved a suitable specification for a standard implementation of a multi-producer, single-consumer (or *mpsc*) queue. An mpsc queue has different resources for pushing and popping, the push resource is persistent and can be shared with multiple threads, while the pop resource is unique. The specification of our deferred queue is shown in figure 17.

isQueue is the invariant containing the state of the queue and is therefore persistent. *isQueueReader* is the same as *isQueue* with an additional token to make it unique and represents the pop permission. Using Spec-DQueueRegister for some S, the *isQueueReader* can be exchanged for an affine *fulfillPermission* S and a persistent *pushPermission* $\gamma\ S$. According to Spec-DQueuePush, the latter resource allows pushing elements that are missing an S to fulfill the queue invariant $I\ v$. We do this by internally converting the whole queue invariant to the predicate $I'\ v := S \mathrel{-\!\!*} I\ v$. Elements already in the queue, satisfying $I\ v$, can be trivially converted to this form by ignoring the S and new elements inserted by Spec-DQueuePush satisfy I' by definition. Since every element is now missing an S we can use Spec-DQueueFulfill with $\square\, S$ to restore the original invariant and get back the *isQueueReader* by supplying each element with one copy of S. As a result, the pop operation has a standard specification Spec-DQueuePop; if it returns an element v it always satisfies $I\ v$.

[9] We think it is possible to formulate the protocol with an arbitrary resource, but it would complicate the construction of the deferred queue that follows. For our purpose of proving Spec-NewScheduler the persistent S is enough.

Spec-DQueueCreate

$$\mathit{ewp}\ (\mathit{queue\_create}\ ())\ \langle\bot\rangle\ \{q.\ \forall I.\ \dot{\models}\ \mathit{isQueueReader}\ q\ I\}$$

Spec-DQueueRegister

$$\frac{\mathit{isQueueReader}\ q\ I}{\dot{\models}\ \mathit{isQueue}\ q\ I * \mathit{fulfillPermission}\ S * \exists\gamma.\ \mathit{pushPermission}\ \gamma\ S}$$

Spec-DQueueFulfill

$$\frac{\mathit{isQueue}\ q\ I * \mathit{fulfillPermission}\ S * \square\, S}{\dot{\models}\ \mathit{isQueueReader}\ q\ I}$$

Spec-DQueuePush

$$\frac{\mathit{isQueue}\ q\ I * \mathit{pushPermission}\ \gamma\ S * \triangleright(S \mathrel{-\!\!*} I\ v)}{\mathit{ewp}\ (\mathit{queue\_push}\ q\ v)\ \langle\bot\rangle\ \{\top\}}$$

Spec-DQueuePop

$$\frac{\mathit{isQueueReader}\ q\ I}{\mathit{ewp}\ (\mathit{queue\_pop}\ q)\ \langle\bot\rangle\ \{v'.\ \mathit{isQueueReader}\ q\ I * (\ulcorner v' = \mathit{None}\urcorner \vee \exists v.\ \ulcorner v' = \mathit{Some}\ v\urcorner * I\ v)\}}$$

Figure 17: Specification of a deferred queue.

4.4 Integrating the Deferred Queue into the Scheduler

Since our deferred queue reuses the code from the `Queue` module, we do not need to update the source code of `Scheduler.run`. But we must amend the proof of Spec-Run, namely the case for handling the *Suspend* effect, shown below because the queue specifications changed.

```
let rec execute fiber =
  match fiber () with
  (* ... *)
  | effect (Suspend register), k =>
      let waker = fun v -> Queue.push run_queue (fun () -> continue k v) in
      register waker;
      next ()
```

First, to do a pop operation, the `execute` function now needs the unique *isQueueReader* resource. Since we call `execute` recursively for each fiber, we must pass this resource to each fiber continuation when running it. The queue invariant *Ready* therefore becomes recursive, where *isQueueReader* is passed into and out of every continuation in the queue.

$$\begin{aligned}\mathit{Ready}_2\ \gamma\ q\ f \triangleq\ & \triangleright\, \mathit{isQueueReader}\, q\, (\mathit{Ready}_2\ \gamma\ q) \mathrel{-\!\!*} \\ & \mathit{ewp}\ (f\ ())\ \langle\bot\rangle\ \{\mathit{osAssigned}\, \gamma * \triangleright\, \mathit{isQueueReader}\, q\, (\mathit{Ready}_2\ \gamma\ q)\}\end{aligned}$$

Before constructing the `waker` function we must now use Spec-DQueueRegister to temporarily change the queue invariant to $I'\ f \coloneqq S \mathrel{-\!\!*} \mathit{Ready}_2\ \gamma\ q\ f$ and obtain a *pushPermission* $\gamma\ S$. Using the push permission we can then prove that `waker` still satisfies *isWaker* because the continuation `k` satisfies I'. After calling the `register` function we obtain the resource S and can use Spec-DQueueFulfill to restore the queue invariant and receive the *isQueueReader* resource. This resource is then passed to `next` to pop a continuation from the queue and then passed to the continuation itself to fulfill the precondition of Ready_2.

To summarize, using a non-standard *deferred* queue specification we were able to strengthen the specification of the *Suspend* protocol. This was needed to prove the specification of Eio's domain manager because it relies on pushing unsafe functions to the scheduler's run queue which become safe to execute by the time the scheduler attempts to pop the next element from the queue. Our deferred queue specification works generically for an mpsc queue without changing its code, and we conjecture that a stronger specification with a non-persistent S is provable, but unnecessary in our use case.

5 Adding Support for Thread-Local Variables

Previously we have looked at a protocol with two effects which suffice to model fibers that can suspend and fork off new fibers. But fibers in Eio can use an additional effect called *GetContext* that we discuss in this section. For each fiber the scheduler keeps track of context metadata, one part of which are *thread-local variables*. Thread-local variables are state that is shared between all fibers of one scheduler (hence thread-local) and a fiber gets access to them via the *GetContext* effect.

Since all fibers of one scheduler execute concurrently on one system-level thread, they have exclusive access to the thread-local variables while they are running. This allows a practical form of shared state without the overhead of synchronization primitives of multithreaded data structures. Some example use-cases are per-scheduler debug tracing of events, where all fibers of one scheduler write to a common log, and inter-fiber message passing, where fibers use a simple queue to exchange messages. Of course, this comes with the restriction that it is only usable for fibers running in the same thread.

In Eio thread-local variables are represented by a dictionary from variable names to arbitrary values and expose an intended API that only allows adding new entries. However, to integrate thread-local variables into our model we simplify this into a single reference. Properties we want to prove about our thread-local variable are:

1. Each time a fiber performs a *GetContext* effect it receives the same reference.
2. As long as a fiber does not perform other effects like *Fork* or *Suspend*, it holds exclusive ownership of the reference.

Code examples illustrating the properties are shown in figure 18. Note that these are only the most basic properties showing that `tlv` acts like a normal reference, but one that can be accessed via an effect. To enable modular proofs of concrete fibers using the thread-local variable, we include in our logical definition a predicate Ω on the stored value that can be instantiated by fibers as needed.

```
let fiber1 () =
  let tlv1 = perform (GetContext ()) in
  (* ... *)
  let tlv2 = perform (GetContext ()) in
  assert (tlv1 = tlv2)

let fiber2 () =
  let tlv = perform (GetContext ()) in
  let v = !tlv in
  (* some computation that does not perform Fork/Suspend *)
  (* ... *)
  assert (!tlv = v)
```

Figure 18: Constructed example of safety for thread-local variables.

5.1 Changes to Logical State

To handle a thread-local variable we must change both the source code and logical definitions. The necessary changes to the implementation are trivial, as the scheduler creates a new reference as part of setting up the runtime environment and passes it to the continuation of the *GetContext* handler as shown in figure 19.

```
let run init (main: unit -> 'a) : 'a =
  (* ... *)
  let rec execute tlv fiber =
    match fiber () with
    (* ... *)
    | effect (GetContext ()), k ->  continue k tlv
  in
  let tlv = ref init in
  execute tlv (fun () -> result := Some (main ()));
  (* ... *)
```

Figure 19: Changes to integrate a thread-local variable into the `Scheduler.run` code.

The updated definitions are described in figure 20. $\mathit{fiberResources}\ l_{tlv}\ \Omega$ represents all resources that a fiber owns while it is running, so each fiber specification now has this as a precondition. The predicate at the moment only includes the points-to connective of l_{tlv} and that its value satisfies Ω. Also, we must change the definition of *Ready* to include $\mathit{fiberResources}\ l_{tlv}\ \Omega$ in the pre- and postcondition so that each suspended fiber in the run queue gets access to the resources when it resumes execution.

$$\begin{aligned}\mathit{fiberResources}\ l_{tlv}\ \Omega &\triangleq \exists v.\ l_{tlv} \mapsto v * \Omega\ v \\ \mathit{Ready}_3\ l_{tlv}\ \Omega\ k &\triangleq \mathit{fiberResources}\ l_{tlv}\ \Omega \mathrel{-\!\!*} \mathit{ewp}\ (k\ ())\ \langle\bot\rangle\ \{\mathit{fiberResources}\ l_{tlv}\ \Omega\}\end{aligned}$$

Figure 20: Updated logical state definitions to model the thread-local variable.

The effect protocols of *Fork* and *Suspend* are amended so that they pass the fiber resources from a fiber to the scheduler and from there to the next running fiber via the protocol pre- and postconditions as shown in figure 21. A fiber uses the *GetContext* effect to receive the concrete l_{tlv} reference, and by virtue of running it already holds the fiber resources to safely dereference l_{tlv}. The crux is that now the whole protocol Coop_3 is parameterized by the location and the predicate on its value. We must do this instead of quantifying them at the effect-level so that the fiber and the scheduler agree on the concrete location.

$$\begin{aligned}\mathit{Coop}_3\ l_{tlv}\ \Omega \triangleq \quad & \mathit{Fork}\ \#\ !\ e\ (e)\ \{\mathit{fiberResources}\ l_{tlv}\ \Omega * \\ & \qquad \triangleright\ (\mathit{fiberResources}\ l_{tlv}\ \Omega \mathrel{-\!\!*} \mathit{ewp}\ (e\ ())\ \langle \mathit{Coop}_3\ l_{tlv}\ \Omega \rangle\ \{\mathit{fiberResources}\ l_{tlv}\ \Omega\})\}. \\ & \quad ?\ ()\ \{\mathit{fiberResources}\ l_{tlv}\ \Omega\} \\ & \mathit{Suspend}\ \#\ !\ \mathit{reg}\ W\ S\ (\mathit{reg})\ \{\mathit{fiberResources}\ l_{tlv}\ \Omega * \mathit{isRegister}_1\ \mathit{reg}\ W\}. \\ & \quad ?\ v\ (v)\ \{\mathit{fiberResources}\ l_{tlv}\ \Omega * W\ v\} \\ & \mathit{GetContext}\ \#\ !\ ()\ \{\top\}.\ ?\ (l_{tlv})\ \{\top\}\end{aligned}$$

Figure 21: Definition of extended Coop_3 protocol with *Fork*, *Suspend*, and *GetContext* effects.

These changes suffice to prove the safety of the two examples in figure 18. In the next section we will show an example of how one can use the Ω predicate to establish constraints on the thread-local variable.

6 Evaluation

We evaluate our model of Eio on a simple example program that uses all features supported by our implementation. The example program (shown in figure 22) may look contrived since it does not do any "useful" computation, but the value of Eio as a library comes from composing computations – not in what is computed concretely.

The program's `main_fiber` function forks off a new fiber `dispatch` (line 23) and communicates with it over a one-element channel represented by the thread-local variable. The channel is initially empty (first argument to `Scheduler.run` in line 29) and `dispatch` polls for data (line 15). `main_fiber` sends one integer `data` (lines 11,25) to `dispatch` which will then run two copies of `(work data)` (line 16) in separate threads, and sum their results. The `work` function simulates time passing using the `yield` function (line 5) and returns its first argument. Yield performs a *Suspend* effect but calls the `waker` function immediately, which has the effect of placing the current fiber at the back of the scheduler's run queue to give other fibers a chance to run.

The example program therefore uses the basic functions for forking and awaiting the completion of fibers, multiple schedulers running in different threads, as well as thread-local variables to communicate between fibers within one thread.

```
1  let yield () =
2    perform (Suspend (fun waker -> waker ()))
3
4  let work x () =
5    yield ();
6    x
7
8  let rec wait_for_data (tlv : tlv ref) =
9    match !tlv with
10   | None -> yield (); wait_for_data tlv
11   | Some data -> tlv := None; data
12
13 let dispatch () =
14   let tlv = perform (GetContext ()) in
15   let data = wait_for_data tlv in
16   let p1 = Fiber.fork_promise (fun () -> Domain_manager.new_scheduler (work data)) in
17   let p2 = Fiber.fork_promise (fun () -> Domain_manager.new_scheduler (work data)) in
18   let r1 = Promise.await p1 in
19   let r2 = Promise.await p2 in
20   r1 + r2
21
22 let main_fiber data () =
23   let p = Fiber.fork_promise dispatch in
24   let tlv = perform (GetContext ()) in
25   tlv := Some data;
26   Promise.await p
27
28 let main () =
29   Scheduler.run None (main_fiber 21)
```

Figure 22: Example program to use all implemented features.

We first give the final specifications of the most important components of our model library in figure 23. The specifications contain both extensions we discussed in sections 4 and 5.

$$\begin{aligned}
&\mathit{Ready}\ l_{tlv}\ \Omega\ \gamma\ q\ k \triangleq \mathit{fiberResources}\ l_{tlv}\ \Omega \twoheadrightarrow \\
&\qquad \triangleright\, \mathit{isQueueReader}\ q\ (\mathit{Ready}\ l_{tlv}\ \Omega\ \gamma\ q) \twoheadrightarrow \\
&\qquad\quad \mathit{ewp}\ (k\ ())\ \langle \bot \rangle\ \{\mathit{osAssigned}\ \gamma * \mathit{fiberResources}\ l_{tlv}\ \Omega * \triangleright\, \mathit{isQueueReader}\ q\ (\mathit{Ready}\ l_{tlv}\ \Omega\ \gamma\ q)\} \\
&\mathit{isWaker}\ \mathit{wkr}\ W \triangleq \forall v.\ W\ v \twoheadrightarrow \mathit{ewp}\ (\mathit{wkr}\ \ v)\ \langle \bot \rangle\ \{\top\} \\
&\mathit{isRegister}\ \mathit{reg}\ W\ S \triangleq \forall \mathit{wkr}.\ \mathit{isWaker}\ \mathit{wkr}\ W \twoheadrightarrow \mathit{ewp}\ (\mathit{reg}\ \ \mathit{wkr})\ \langle \bot \rangle\ \{\Box S\} \\
&\mathit{Coop}\ l_{tlv}\ \Omega \triangleq \quad \mathit{Fork}\ \#\ !\ e\ (e)\ \big\{\mathit{fiberResources}\ l_{tlv}\ \Omega * \\
&\qquad\qquad\qquad\qquad \triangleright\, (\mathit{fiberResources}\ l_{tlv}\ \Omega \twoheadrightarrow \mathit{ewp}\ (e\ \ ())\ \langle \mathit{Coop}\ l_{tlv}\ \Omega \rangle\ \{\mathit{fiberResources}\ l_{tlv}\ \Omega\})\big\}. \\
&\qquad\qquad\quad ?\ ()\ \{\mathit{fiberResources}\ l_{tlv}\ \Omega\} \\
&\qquad\quad \mathit{Suspend}\ \#\ !\ \mathit{reg}\ W\ S\ (\mathit{reg})\ \{\mathit{fiberResources}\ l_{tlv}\ \Omega * \mathit{isRegister}\ \mathit{reg}\ W\ S\}. \\
&\qquad\qquad\quad ?\ v\ (v)\ \{\mathit{fiberResources}\ l_{tlv}\ \Omega * W\ v * S\} \\
&\qquad\quad \mathit{GetContext}\ \#\ !\ ()\ \{\top\}.\ ?\ (l_{tlv})\ \{\top\}
\end{aligned}$$

Spec-Run

$$\frac{\Omega\ \mathit{init} * \forall l_{tlv}.\ \mathit{fiberResources}\ l_{tlv}\ \Omega \twoheadrightarrow \mathit{ewp}\ (\mathit{main}\ \ ())\ \langle \mathit{Coop}\ l_{tlv}\ \Omega \rangle\ \{v.\ \Box \Phi\ v * \mathit{fiberResources}\ l_{tlv}\ \Omega\}}{\mathit{ewp}\ (\mathit{run}\ \ \mathit{init}\ \ \mathit{main})\ \langle \bot \rangle\ \{v.\ \Box \Phi\ v * \mathit{fiberResources}\ l_{tlv}\ \Omega\}}$$

Spec-ForkPromise

$$\frac{\mathit{promiseIn}\ v * \mathit{fiberResources}\ l_{tlv}\ \Omega * \mathit{fiberResources}\ l_{tlv}\ \Omega \twoheadrightarrow \mathit{ewp}\ (f\ \ ())\ \langle \mathit{Coop}\ l_{tlv}\ \Omega \rangle\ \{v.\ \Box \Phi\ v * \mathit{fiberResources}\ l_{tlv}\ \Omega\}}{\mathit{ewp}\ (\mathit{fork\_promise}\ \ f)\ \langle \mathit{Coop}\ l_{tlv}\ \Omega \rangle\ \{p.\ \exists \gamma.\ \mathit{isPromise}\ \gamma\ p\ \Phi * \mathit{fiberResources}\ l_{tlv}\ \Omega\}}$$

Spec-Await

$$\frac{\mathit{promiseIn}\ v * \mathit{fiberResources}\ l_{tlv}\ \Omega * \mathit{isPromise}\ \gamma\ p\ \Phi}{\mathit{ewp}\ (\mathit{await}\ \ p)\ \langle \mathit{Coop}\ l_{tlv}\ \Omega \rangle\ \{v.\ \Box \Phi\ v * \mathit{fiberResources}\ l_{tlv}\ \Omega\}}$$

Spec-NewScheduler

$$\frac{\Omega'\ \mathit{init} * \forall l'_{tlv}.\ \mathit{fiberResources}\ l'_{tlv}\ \Omega \twoheadrightarrow \mathit{ewp}\ (\mathit{main}\ \ ())\ \big\langle \mathit{Coop}\ l'_{tlv}\ \Omega' \big\rangle\ \big\{v.\ \Box \Phi\ v * \mathit{fiberResources}\ l'_{tlv}\ \Omega'\big\}}{\mathit{ewp}\ (\mathit{new\_scheduler}\ \ \mathit{init}\ \ \mathit{main})\ \langle \mathit{Coop}\ l_{tlv}\ \Omega \rangle\ \{v.\ \Box \Phi\ v * \mathit{fiberResources}\ l_{tlv}\ \Omega\}}$$

Figure 23: Specification of the public interface of the Eio library model.

Using these specifications we proved the safety of the example program and its functional correctness by establishing specifications for each function as shown in figure 24. We can see that there is some amount of boilerplate (colored in blue). Each function that yields execution to another fiber by performing an effect – either directly or indirectly through another function call – needs $\mathit{fiberResources}\ l_{tlv}\ \Omega$, which signifies the ownership over the thread-local variable. Additionally, any fiber that wants to fork off another fiber using `Fiber.fork_promise` needs the persistent $\mathit{promiseInv}$ resource to interact with the global collection of promises. The predicate $\Omega_{chan}\ \gamma$ as part of $\mathit{fiberResources}\ l_{tlv}\ (\Omega_{chan}\ \gamma)$ restricts the thread-local variable l_{tlv} to be a channel for a single message n.

$$\begin{aligned}\Omega_{chan}\ \gamma\ v \triangleq\ & \ulcorner v = \mathit{None} \urcorner \\ & \vee\ \exists n.\ \ulcorner v = \mathit{Some}\ n \urcorner * \boxed{\mathrm{ag}(n)}^{\gamma}\end{aligned}$$

Spec-Work

$$\frac{\mathit{fiberResources}\ l_{tlv}\ \Omega}{\mathit{ewp}\ (\mathit{work}\ \ n\ ())\ \langle \mathit{Coop}\ l_{tlv}\ \Omega \rangle\ \{v.\ \ulcorner v = n \urcorner * \mathit{fiberResources}\ l_{tlv}\ \Omega\}}$$

Spec-WaitForData

$$\frac{\mathit{fiberResources}\ l_{tlv}\ (\Omega_{chan}\ \gamma)}{\mathit{ewp}\ (\mathit{wait\_for\_data}\ \ l)\ \langle \mathit{Coop} \rangle\ \left\{v.\ \exists n.\ \ulcorner v = n \urcorner * \boxed{\mathrm{ag}(n)}^{\gamma} * \mathit{fiberResources}\ l_{tlv}\ (\Omega_{chan}\ \gamma)\right\}}$$

Spec-Dispatch

$$\frac{\mathit{promiseInv} * \mathit{fiberResources}\ l_{tlv}\ (\Omega_{chan}\ \gamma)}{\mathit{ewp}\ (\mathit{dispatch}\ \ ())\ \langle \mathit{Coop} \rangle\ \left\{v.\ \exists n.\ \ulcorner v' = n + n \urcorner * \boxed{\mathrm{ag}(n)}^{\gamma} * \mathit{fiberResources}\ l_{tlv}\ (\Omega_{chan}\ \gamma)\right\}}$$

Spec-MainFiber

$$\frac{\mathit{promiseInv} * \mathit{fiberResources}\ l_{tlv}\ (\Omega_{chan}\ \gamma) * \boxed{\mathrm{ag}(n)}^{\gamma}}{\mathit{ewp}\ (\mathit{main\_fiber}\ n\ ())\ \langle \mathit{Coop} \rangle\ \{v.\ \ulcorner v = n + n \urcorner * \mathit{fiberResources}\ l_{tlv}\ (\Omega_{chan}\ \gamma)\}}$$

Spec-Main

$$\frac{}{\mathit{ewp}\ (\mathit{main}\ \ ())\ \langle \bot \rangle\ \{v.\ \ulcorner v = 42 \urcorner\}}$$

Figure 24: Specification of the example program.

While we proved the safety of this program (and of the core abstractions of the Eio library), the complete Eio library has more features that are still unexplored. This includes cancellation of fibers, resource management using switches and several operating system primitives like timers, so we cannot make any statements about programs using these features. Nevertheless, our model is an important step in the direction of proving the safety of Eio and programs that use it. Iris along with its features like ghost state and shareable invariants to reason about multithreaded and stateful code were key in this development.

7 Conclusion

7.1 Related Work

Concurrency With Effects There are other approaches to implementing cooperative concurrency with effects even within OCaml 5. One example is the Picos framework [12], an interoperability framework that defines a small set of data types and effects that can be reused by other cooperative concurrency libraries (such as Eio) in order to speak a common protocol and be mutually interoperable. Picos defines *computations* and *fibers*, which are mostly equivalent, respectively, to Eio's promises and fibers. The main difference is the *trigger* concept, which in Eio's terms can be thought of as a mutable reference to an optional `waker` function. The workflow to await a future result (i.e. a Picos computation) consists of first attaching an empty trigger to the computation and then assigning a `waker` function later. This is done by performing an *Await* effect after attaching the trigger so that the effect handler (implemented by a library such as Eio) then creates a `waker` function and assigns it to the trigger. When the computation finishes it will signal the trigger, which calls the `waker` function and consequently places the original fiber in the scheduler's run queue.

While the *Await* effect carrying a trigger is technically first-order – as opposed to Eio's higher-order *Suspend* effect carrying a `register` function – a trigger still contains higher-order state and the whole process resembles Landin's Knot for building recursive functions with higher-order state. So it is unclear to us whether a specification for the Picos primitives would be any simpler to prove or easier to use than the specification for Eio primitives we have presented so far.

Session Types as Effect Specifications Protocols in Hazel take some inspiration from session types but with the restriction that a protocol is always an infinite repetition of a single step, whereas session types usually allow defining a sequence of different steps. Current work by Tang [17] explores the connection between session types and effect protocols further and defines a lambda calculus $\lambda_{\text{eff}}^{\boxtimes}$ that uses a standard session type formalization for its effect system. This allows a programmer to define multistep protocols and even bidirectional effects where the handler and client swap roles. However, for our purposes Hazel is completely sufficient since multistep protocols can be simulated by Hazel's protocols and bidirectional effects are not possible in OCaml 5 to begin with.

7.2 Future Work

The work we presented so far suffices to prove the safety of programs that use a small subset of the full functionality provided by Eio. To improve the usefulness of our model and be applicable to more programs it would be desirable to incorporate more features into our model of Eio in future work, such as switches and support for cancelling fibers. While switches are mainly used for the hierarchical organization of fibers and to efficiently clean up fiber resources, cancellation poses some interesting safety questions because there are situations that must be avoided, such as being able to cancel a fiber twice.

Instead of growing the model of Eio it would also be interesting to extend the existing specifications. Mainly, we would be interested in proving that a scheduler will never "forget fibers". Since weakest preconditions in Iris do not prove termination our specifications have the unfortunate drawback of being fulfilled by functions that diverge. Because a scheduler explicitly handles fiber continuations it would be possible to accidentally drop a continuation which has the effect of making the fiber diverge, as well as any other fiber that awaits its result.

While we cannot solve the whole termination problem (since deadlocks are possible), intuitively we should be able to track the state of each fiber to ensure that when a fiber is captured in a continuation, the continuation is used linearly, which means that it is explicitly invoked or discarded at some point. This also extends to data structures that contain continuations like the scheduler's run queue and a broadcast, as they must never drop the contained continuations. To track the linear usage of continuations it could be helpful to draw inspiration from Iron [2], a separation logic built on Iris to enable reasoning about linear resources.

7.3 Results

In this thesis we have proven specifications for a subset of the Eio library, including a common denominator scheduler that controls fibers which can await the completion of promises in a multithreaded setting.

We have also defined and verified general and reusable protocols for the main three effects of Eio: *Fork*, *Suspend*, and *GetContext*. We showed that the function specifications and the effect protocols are enough to verify the safety (including effect safety) of an example program that uses all of our modelled features. Additionally, we have verified specifications for two nontrivial data structures used by Eio. For the broadcast data structure we adapted the existing proof of CQS by Koval et al. [9] and for the scheduler's run queue we proved a – to our knowledge novel – specification for a multi-producer single-consumer queue with a temporarily suspendable invariant. Finally, we have extended the original *HH* language with multithreading primitives and amended the adequacy result which shows that we can use this language to reason about programs that use both multithreading and effect handlers.

Bibliography

[1] Andrej Bauer and Matija Pretnar. "Programming with algebraic effects and handlers". In: *Journal of logical and algebraic methods in programming* 84.1 (2015), pp. 108–123.

[2] Aleš Bizjak et al. "Iron: Managing obligations in higher-order concurrent separation logic". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–30.

[3] Paulo De Vilhena. "Proof of Programs with Effect Handlers". PhD thesis. Université Paris Cité, 2022.

[4] 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.

[5] Angus Hammond et al. "An Axiomatic Basis for Computer Programming on the Relaxed Arm-A Architecture: The AxSL Logic". In: *Proceedings of the ACM on Programming Languages* 8.POPL (2024), pp. 604–637.

[6] Hans Hüttel et al. "Foundations of Session Types and Behavioural Contracts". In: *ACM Comput. Surv.* 49.1 (Apr. 2016). ISSN: 0360-0300. DOI: 10.1145/2873052. URL: https://doi.org/10.1145/2873052.

[7] Ralf Jung et al. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018), e20.

[8] Ralf Jung et al. "RustBelt: Securing the foundations of the Rust programming language". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–34.

[9] 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.

[10] Daan Leijen. *Algebraic effects for functional programming*. Tech. rep. Technical Report. MSR-TR-2016-29. Microsoft Research technical report, 2016.

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

[12] *Picos – Interoperable effects based concurrency*. https://github.com/ocaml-multicore/picos/. Accessed: 2024-04-04.

[13] Gordon D Plotkin and Matija Pretnar. "Handling algebraic effects". In: *Logical methods in computer science* 9 (2013).

[14] Xiaojia Rao et al. "Iris-wasm: Robust and modular verification of webassembly programs". In: *Proceedings of the ACM on Programming Languages* 7.PLDI (2023), pp. 1096–1120.

[15] John C Reynolds. "Separation logic: A logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74.

[16] *Session-Types Effect Handlers*. POPL24 Student Research Comptetition, https://openjdk.java.net/projects/loom/, and https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html. Accessed: 2024-04-18.

[17] *Session-Types Effect Handlers*. POPL24 Student Research Comptetition, https://thwfhk.github.io/files/session-handlers-popl24-src.pdf. Accessed: 2024-04-04.

[18] KC Sivaramakrishnan et al. "Retrofitting effect handlers onto OCaml". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 206–221.

[19] 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.

Appendix

A Translation Table

Below are some terms we used in this thesis and their corresponding terms as used in the source code of the Eio library and in our mechanization.

Thesis	Eio	Mechanization
`Fiber.fork_promise`	`Fiber.fork_promise`	`fork_promise`
`Promise.await`	`Promise.await`	`await`
`Scheduler.run`	`Sched.run`	`run`
`Domain_manager.new_scheduler`	`Domain_manager.run`	`new_scheduler`
`waker` function	`enqueue`	`waker`
`register` function	`f`	`register`
broadcast	`Broadcast` & `Cells`	`CQS`

B Adding Multithreading to *HH*

OCaml 5 added not only effect handlers but also the ability to run OCaml code in parallel using multiple system-level threads, called *Multicore OCaml*[10]. OCaml 5 adopts the name *domain* for a parallel thread of execution, although in the following we prefer to use the term thread. Each domain in OCaml 5 corresponds to one system-level thread and the usual rules of multithreaded execution apply, i.e. domains are preemptively scheduled and have shared memory. Eio defines a *domain manager* to make use of multithreading by spawning a new thread and running a separate scheduler in it. So while each Eio scheduler only runs fibers in a single thread, multiple threads can run separate schedulers and fibers can communicate with fibers running in separate threads.

Heaplang supports reasoning about multithreaded programs by implementing fork and join operations for threads and defining atomic steps in the operational semantics, which enables opening Iris *shared invariants* for one such step. In contrast, *HH* did not define any multithreaded operational semantics, but as the language is based on heaplang it contained most of the required building blocks already. In the following we explain which missing pieces we added to enable writing multithreaded programs in *HH*.

Atomic Variables 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 *HH*, we treat all references as memory locations that are potentially shared with different threads, because in the standard formulation of multithreaded operational semantics in Iris all memory operations are sequentially consistent by construction. That means even normal load and store operations on locations are treated as synchronized accesses. This seems to be the standard approach as it is the same in heaplang.

Extending the Operational Semantics

To allow reasoning in Hazel about multithreaded programs we need a multithreaded operational semantics for *HH*, as well as specifications for the new primitive operations *fork*, *cmpXchg* (*compare-and-exchange*) and *faa* (*fetch-and-add*).

The language interface of Iris provides a way to easily define a multithreaded semantics $\rightarrow_{mt}$ via a *thread-pool*, provided one defines a *thread-local* operational semantics $\rightarrow_t$. The thread-pool is a list of expressions that represents threads running in parallel. At each step, one expression 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.

$$\textsc{MTStep}\ \frac{(e,\sigma) \rightarrow_t (e',\sigma',es')}{(es_1 +\!\!+ e +\!\!+ es_2,\sigma) \rightarrow_{mt} (es_1 +\!\!+ e' +\!\!+ es_2 +\!\!+ es',\sigma')}$$

[10] While it was previously possible to run code in multiple threads using the `Thread` module, only one thread could execute OCaml code at a time, so it was not truly parallel.

We extend the existing thread-local operational semantics of *HH* with the expressions *fork*, *cmpXchg* and *faa*, taking their definitions from heaplang. Additionally, we need to prove specifications for the three operations. *cmpXchg* and *faa* are standard and the same as heaplang, so we will not discuss them here. The only interesting design decision in the case of *HH* is how effects and the *fork* expression interact. This design is guided by the fact that in OCaml 5 effects never cross thread-boundaries. An unhandled effect that propagates up to the top level is treated as an error and crashes the program. As such we must impose the empty protocol on a newly forked off thread, independent on what protocol Ψ the main thread obeys.

$$\textsc{Step-Fork}\ \frac{ewp\ (e)\ \langle \bot \rangle\ \{\top\}}{ewp\ (fork\ e)\ \langle \Psi \rangle\ \{v.\ v = ()\}}$$

Using these primitive operations we can then build the standard *CAS* (*compare-and-set*), *spawn*, and *join* operations on top and prove their specifications. These constructions and specifications were also taken from heaplang. Note that for *spawn* we must also impose the empty protocol on f as this expression will be forked off.

$$\textsc{Thread-Spawn}\ \frac{ewp\ (f\ ())\ \langle \bot \rangle\ \{v.\ Q\ v\}}{ewp\ (thread\_spawn\ f)\ \langle \Psi \rangle\ \{j.\ joinHandle\ j\ Q\}} \qquad \textsc{Thread-Join}\ \frac{joinHandle\ j\ Q}{ewp\ (thread\_join\ j)\ \langle \Psi \rangle\ \{v.\ Q\ v\}}$$

This allows us to implement standard multithreaded programs which also use effect handlers.

Using Invariants in Hazel

Invariants in Iris can be used to share resources between threads and are a crucial element to proving specifications of multithreaded programs. 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 the invariants and had an invariant access lemma which allows opening an invariant if the current expression is atomic. So in order to use invariants we only had to provide proofs for which expressions are atomic. We provided proofs for all primitive evaluation steps. The proofs are the same for all steps and always consist of two parts. First, we need to show that the expression is `head_atomic`, i.e. that it steps to a value in one step, which is true by definition of the operational semantics. Then, we need to show that all subexpressions are values, which is true by construction for the expressions we show to be atomic.

```
Definition head_atomic (e : expr) : Prop :=
  ∀ σ e' σ' efs,
    head_step e σ e' σ' efs → is_Some (to_val e').
Definition sub_exprs_are_values (e : expr) :=
  ∀ K e', e = fill K e' → to_val e' = None → K = [].
Lemma ectx_language_atomic e :
  head_atomic e → sub_exprs_are_values e → Atomic e.
Proof. (* ... *) Qed.

Instance load_atomic v : Atomic (Load (Val v)).
Proof. (* ... *) Qed.
Instance store_atomic v1 v2 : Atomic (Store (Val v1) (Val v2)).
Proof. (* ... *) Qed.
Instance cmpxchg_atomic v1 v2 v3 : Atomic (CmpXchg (Val v1) (Val v2) (Val v3)).
Proof. (* ... *) Qed.
```

Since performing an effect starts a chain of evaluation steps to capture the current continuation, it is never atomic. Therefore, invariants and effects do not interact in any interesting way.