Verifying an Effect-Based Cooperative Concurrency Scheduler in Iris

Adrian Dapprich
Department of Computer Science
Saarland University

Advisors: Prof. Derek Dreyer & Prof. François Pottier

April 2, 2024

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, and for languages with effect handlers there exists the program logic Hazel which is built on top of the Iris separation logic framework.

We use Hazel to 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 Iris framework 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 Hazel language to include multi-threading 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 operating system threads, 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, OCaml 5's Eio).

In this thesis we 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 [?]. In order to formally verify the code that uses effect handlers we use the Hazel program logic by de Vilhena [6, 1].

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 "this program will compute the n-th Fibonacci number for a given input n".

Separation Logic We express specifications as logical propositions and do all reasoning in a separation logic called *Iris* [3]. Separation logics [?] are based on Hoare logic, which has 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 $l \mapsto v$, 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 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. Separation logics have been successfully applied in many program verification developments [?, ?, ?] as they are useful for modular reasoning about stateful and multithreaded programs.

Iris Iris is not only a separation logic, but a whole separation logic framework implemented in Coq that can be instantiated with different programming languages. This allows us to layer different *program logics* on top of the base separation logic, which contain additional reasoning rules about the evaluation

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

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. 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 comes from the fact that this proposition is by definition the weakest precondition P that makes the Hoare triple $\{P\}\, e\, \{v.\, Q\, v\}$ true and 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 its specification in terms of the assumptions P and conclusion Q. It will also establish 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 will 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 will never be observably violated.

The standard language for Iris is called heaplang and is an ML-type 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 Hazel language for Iris.

Effect Handlers Effect handlers are a versatile concept explored in some research languages [**koka**, **eff**] and now also implemented in OCaml 5. The OCaml 5 implementation brings with it an extensible variant type `Effect.t`, meaning one can add new constructors to the type, and a keyword `perform` to perform an effect, which transfers control to an appropriate effect handler. 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.

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 will treat unhandled effects as an error and crash the program. So to prove the safety of OCaml 5 programs we must additionally establish their effect safety.

Hazel & Protocols In our development we use the Iris language Hazel by de Vilhena [6, 1] which formalizes an ML-like language with effect handlers. We will restate the most important concepts but for a deeper understanding we recommend reading [6].

[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 E : int -> bool Effect.t *)
effect E : int -> bool

(* Evaluating perform with a value of type 'a Effect.t will transfer
 * control to the enclosing handler and (possibly) return with 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 will be evaluated every time.
 * When e is reduced to a value, the non-effect branches are
 * used for pattern matching as usual. *)
match e () with
| v -> ...
| effect (E v) k -> ...
```

Figure 1: Example for the effect handler syntax.

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 syntax is inspired by session types, which also define a type of communication protocol. 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 will receive 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 $\{P\}\,handler\ v\,\{w,\ Q\ w\}$, where we explicitly name the handler that will handle the effect. But the client only indirectly invokes the effect handler by evaluating a *perform* expression, so in practice we prove the following Hoare triple.

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

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 $\mathscr{f}\,\#\,\Psi$ to give a name to protocols. Our example effect `E` from figure 1 could therefore be formalized using the following protocol $\Psi_{\mathtt{E}}$.

maybe more expressive example

$$\Psi_{\mathtt{E}} \coloneqq \mathtt{E}\,\#\,!\,i\,(i)\,\{i \in \mathtt{int}\}.\ ?\,b\,(b)\,\{b \in \mathtt{bool}\}$$

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

[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

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 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.

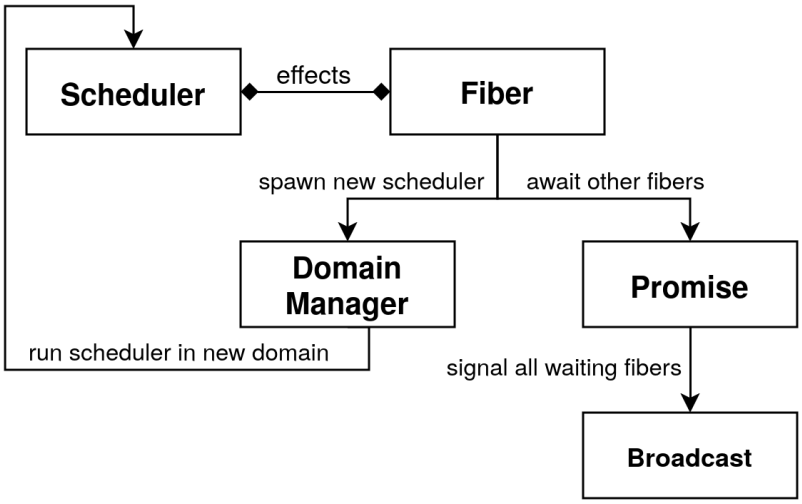

Figure 2: Eio Module Hierarchy

Figure 2 shows the simplified module hierarchy of the concepts we focus on. A standard arrow stands for a direct source code dependency from one module to another. The diamond arrow between `Scheduler` and `Fiber` stands for the implicit dependency that fibers perform effects which are handled by code in the scheduler module.

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 [4], but Eio customizes the implementation so we had to adapt the proof. We discuss this process in section 3. Fibers in Eio also have access to *thread-local variables* by performing a *GetContext* effect, which is discussed in section 4. They are thread-local in the sense that they are shared between all fibers of one scheduler. Finally, we discuss our addition of multi-threading to the Hazel operational semantics in order to model running schedulers in different threads. This turned out to be technically trivial, so we only discuss it in 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 multi-threading to Hazel's operational semantics, which shows we can reason about programs that use both **multi-threading** and **effect handlers**.

2 Verifying a Simplified Eio Scheduler with Promises

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

Cooperative concurrency schedulers for user-level threads (i.e. *fibers*) are commonly treated in the literature on effect handlers [2, 5, 6] to because they are a good example for the usefulness of manipulating delimited continuations with effect handlers. Generally, the architecture contains an effect handler as the scheduler and fibers are normal functions which perform effects to yield execution. This is because performing an effect causes execution to jump to the enclosing effect handler (i.e. the scheduler), providing it with the rest of the fiber's computation in the form of a delimited continuation. The scheduler keeps track of a collection of these continuations and by invoking one of them the next fiber is scheduled. This approach is also used in Eio.

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

2.1 Implementation

Let us first get an idea of how different components of the core Eio fiber abstraction interact by looking at their types. `Scheduler.run`[4] is the main entry point to Eio. It runs a scheduler and is provided a function which represents the first fiber to be executed. A scheduler runs the main fiber and all forked-off fibers in a single thread. However, a fiber can also spawn new schedulers in separate threads to run other fibers in parallel as detailed in appendix B. The `Fiber.fork_promise` function is used to spawn fibers in the current scheduler. The function returns a promise holding the eventual return value of the new fiber. The promise is thread-safe so that it can be shared with fibers running in different threads. The `Promise.await` function can be used by any fiber to wait until the value of a promise is available. Common problems like deadlocks are not prevented in any way and are the responsibility of the programmer.

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

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 runs the main fiber that is passed as an argument.

The `run_queue` (line 6) contains closures that will immediately invoke the continuation of an effect. This represents ready fibers which can continue execution from the point where they performed an effect. The `next` function (line 7) pops one fiber (i.e. function) from the `run_queue` and executes it. If no more ready fibers remain – either because all fibers terminated or there is a deadlock – the `next` function returns and the scheduler exits.

The inner `execute` function (line 12) is called once on each fiber to evaluate it and handle any performed effects.

Value Case

The non-effect case of the `match` (line 14) 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 24) that is read when the scheduler exits (line 25).
- 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.

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

```
1  effect Fork : (unit -> 'a) -> unit
2  type 'a waker : 'a -> unit
3  effect Suspend : ('a waker -> unit) -> 'a
4
5  let run (main : unit -> 'a) : 'a option =
6    let run_queue = Lf_queue.create () in
7    let next () =
8      match Lf_queue.pop run_queue with
9      | None -> ()
10     | Some cont -> cont ()
11   in
12   let rec execute fiber =
13     match fiber () with
14     | () -> next ()
15     | effect (Fork fiber) k ->
16         Lf_queue.push run_queue (fun () -> invoke k ());
17         execute fiber
18     | effect (Suspend register) k =>
19         let waker = fun v -> Lf_queue.push run_queue (fun () -> invoke k v) in
20         register waker;
21         next ()
22   in
23   let result = ref None in
24   execute (fun () -> result := main ());
25   !result
```

Figure 3: Implementation of `Scheduler.run`

Fork Case

Handling a *Fork* effect (line 15) is simple because it only carries a new `fiber` to be executed, so the handler recursively calls the `execute` function (line 17) to execute it immediately. The execution of the original fiber is paused due to performing an effect and its continuation `k` is placed in the run queue so that it can be scheduled again (line 16). This prioritizes the execution of a new fiber and is a design decision by Eio. It would be equally valid to place the `fiber` argument in the run queue instead.

Suspend Case

Handling a *Suspend* effect (line 18) 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 will not continue execution. So the `waker` function "wakes up" a fiber by placing its continuation `k` into the run queue (line 19). The `register` function is called by the scheduler right after the fiber suspends execution (line 20) and is responsible for installing `waker` as a callback at a suitable place (or even call it directly). For example, to implement promises, the `waker` function is installed in a data structure that will call `waker` after the promise is fulfilled.

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

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

2.1.2 `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 17) and spawns the provided function as a new fiber using the *Fork* effect (line 22). Promises are always created in a `Waiting` (or unfulfilled) state and

```
(* promise.ml *)
type 'a t = Done of 'a | Waiting of Broadcast.t

let create () : 'a t =
  let bcst = Broadcast.create () in
  Atomic.create (Waiting bcst)

let fulfill p result =
  match Atomic.get p with
  | Done _ -> error "impossible"
  | Waiting bcst ->
      Atomic.set p (Done result);
      Broadcast.signal_all bcst

(* fiber.ml *)
let fork_promise (f : unit -> 'a) : 'a Promise.t =
  let p = Promise.create () in
  let fiber = fun () ->
    let result = f () in
    Promise.fulfill p result
  in
  perform (Fork fiber)
  p
```

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

calling `Promise.fulfill` will move it to the `Done` state, at which point the final value can be retrieved. The meaning of the `Broadcast.t` is explained in the next section.

When `f` is reduced to a value `result`, `Fiber.fork_promise` then fulfills the promise with that value (line 20), which will signal all fibers waiting for that result to wake up (line 13). The major difference with respect to the implementation of de Vilhena is that promises in Eio are entirely handled by the fiber, and not in the effect handler code of the scheduler. This achieves a better separation of concerns and simplifies the logical state needed for the proof.

2.1.3 `Promise.await`

```
let make_register (p: 'a t) (bcst: Broadcast.t) : (unit waker -> unit) =
  fun waker ->
    let register_result = Broadcast.register bcst waker in
    match register_result with
    | None -> ()
    | Some register_handle ->
      match Atomic.get p with
      | Done result ->
          if Broadcast.try_unregister register_handle
          then waker ()
          else ()
      | Waiting _ -> ()

let await (p: 'a t) : 'a =
  match Atomic.get p with
  | Done result -> result
  | Waiting bcst ->
    let register = make_register p bcst
    perform (Suspend register);
    match Atomic.get p with
    | Done result -> result
    | Waiting _ -> error "impossible"
```

Figure 5: Implementation of `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 a *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 difference is 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 the `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 `Called` means that it already called the supplied callback because the function detected the signal while it was running. Otherwise, a return value of `Registered register_handle` means that the callback was registered but the signal might have been already sent but missed. It is the responsibility of the caller to check if the signal was already sent and if it was, to then cancel the registration again This is done by calling the `Broadcast.try_unregister` function on the `register_handle`, which returns a boolean indicating the cancellation status. If the cancellation was successful, the previously registered callback will not be called when `Broadcast.signal_all` is executed. The implementation and specification of the functions will be expanded upon in section 3, for now we just explain their usage in the context of `Promise.await`.

```
1  type t
2  type signal = unit -> unit
3  type register_result = Called | Registered of register_handle
4  type register_handle
5
6  val create : unit -> t
7  val register : t -> signal -> 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 23) then the fiber should wait until that is the case, so it performs a *Suspend* effect (line 25). The `register` function passed to the effect will register the `waker` function using `Broadcast.register` (line 9). When at some point the `Broadcast.signal_all` function is called – this happens in `Fiber.fork_promise` – all registered `wakers` will be called in turn. Recall that calling a `waker` function will enqueue the fiber that performed the *Suspend* effect in the scheduler's run queue so that it can continue execution.

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

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, after the *Suspend* effect returns (line 26) 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 21) 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 9) and the call to `Broadcast.signal_all` in the other thread[5]. If `Broadcast.register` detects that it lost the race, it will directly call the `waker` function and

[5]There is a possible race to set an atomic location that holds the state of the registration. Interested readers are directed to the implementation linked in [4].

return `Called`. 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 13), and – if it is fulfilled – try to cancel the `waker` registration. The cancellation will fail if the `waker` function was already called. Otherwise, the cancellation succeeds and the `register` function has the responsibility of calling `waker` itself (line 16).

The only **safety** concerns in the above implementation are `Fiber.fork_promise` expecting the promise to be unfulfilled after the fiber has finished execution (figure 4 line 10), and `Promise.await` expecting the promise to be fulfilled in the last match (figure 5 line 28). In both cases, the program would crash (signified by the `error` expression) if the expectation is violated. So to establish the safety of Eio we wish to prove that the expectations always hold, and the two `error` expressions are never reached. In the next section we show how the former situation is addressed by defining an affine permission to fulfill a promise, and the latter is a consequence of the protocol of the *Suspend* effect.

2.2 Specification

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

2.2.1 Protocols

First we look at the protocols for the *Fork* and *Suspend* effect that are shown in figure 7. In Hazels' protocol syntax they are formalized in the following way, where the precondition of *Suspend* is given the name *isRegister* to describe the behavior of the fiber-defined `register` function.

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

Figure 7: Definition of *Coop* Protocol with *Fork* & *Suspend* Effects.

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* protocol itself. This means spawned off fibers can again perform *Fork* and *Suspend* effects. The $\mathit{ewp}\ (e)\ \langle \mathit{Coop}\rangle\ \{\top\}$ is guarded behind a later modality because of the recursive occurrence of the *Coop* protocol. Since promise handling is done entirely in the fibers and the *Fork* effect just hands off the fiber to the scheduler, the protocol is simplified in two ways compared to the original from the case study of de Vilhena. First, the scheduler does not interact with the return value of the fiber, so the *ewp* has a trivial postcondition. Second, because the scheduler does not create and return the promise, the protocol itself also has a trivial postcondition.

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

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

PS-Create

$$\frac{}{\dot{\models}\exists\gamma.\ \mathit{promiseWaiting}\ \gamma * \mathit{promiseWaiting}\ \gamma}$$

PS-Combine

$$\frac{\mathit{promiseWaiting}\ \gamma * \mathit{promiseWaiting}\ \gamma}{\square\ \mathit{promiseDone}\ \gamma}$$

PS-Contra

$$\frac{\mathit{promiseWaiting}\ \gamma * \mathit{promiseDone}\ \gamma}{\bot}$$

Figure 8: Rules for the *promiseWaiting* γ and *promiseDone* γ.

example, in the `Promise.await` specification below, we ensure that the promise must be fulfilled before the effect returns by instantiating `P` with a resource that says the promise is fulfilled.

2.2.2 Logical State

The most basic ghost state we track is whether a promise is fulfilled or not. If a promise `p` is unfulfilled, two copies of *promiseWaiting* γ exist, one owned by the fiber and one by the invariant that tracks the state of all promises. When fulfilling the promise, both copies can be combined and converted to a persistent *promiseDone* γ resource. The *promiseWaiting* γ and *promiseDone* γ resources cannot exist at the same time. This design allows us to deduce the current state of the promise depending on if we own a *promiseWaiting* γ or a *promiseDone* γ. This is formalized in the rules in figure 8.

Maybe use meta_tokens to hide gamma

Other pieces of ghost state are *PInvInner*, *isPromise*, and *Ready* described in figure 9. *PInvInner* tracks additional resources for all existing promises by using an authoritative map which contains for each promise: a location p holding its current program value, a ghost name γ that is used for the *promiseWaiting* γ and *promiseDone* γ resources, and a predicate Φ that describes the value the promise will eventually hold.

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

PromiseInv is an invariant that wraps *PInvInner* so that we can share it.

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

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

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

Figure 9: Logical State Definitions for the Verification of Scheduler & Promise Modules

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

2.2.3 `Scheduler.run`

The interesting part about the scheduler specification 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* 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.

Spec-Run
$$\frac{ewp\ (main\ ())\ \langle Coop\rangle\ \{v,\ \square\,\Phi\ v\}}{ewp\ (run\ main)\ \langle\bot\rangle\ \{v,\ \square\,\Phi\ v\}}$$

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

Explain why it's hard

2.2.4 `Fiber.fork_promise`

The specification Spec-ForkPromise expresses that we receive from $fork\_promise$ a promise value 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, f obeys the *Coop* 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 executing.

Spec-ForkPromise
$$\frac{PromiseInv * ewp\ (f\ ())\ \langle Coop\rangle\ \{v,\ \square\,\Phi\ v\}}{ewp\ (fork\_promise\ f)\ \langle Coop\rangle\ \{p,\ isPromise\ p\ \Phi\}}$$

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 will call *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* predicate, instantiating P with $(\lambda\ v,\ \ulcorner v = ()\urcorner * promiseDone\ \gamma)$ so that we receive a $promiseDone\ \gamma$ resource when the effect returns.

Spec-MakeRegister
$$\frac{PromiseInv * isPromise\ p\ \Phi * isBroadcast\ bcst}{ewp\ (make\_register\ p\ bcst)\ \langle\bot\rangle\ \{reg,\ isRegister\ reg\ (\lambda\ v,\ \ulcorner v = ()\urcorner * promiseDone\ \gamma)\}}$$

Spec-Await
$$\frac{PromiseInv * isPromise\ p\ \Phi}{ewp\ (await\ p)\ \langle Coop\rangle\ \{v,\ \square\,\Phi\ v\}}$$

In figures 10 and 11 we give Hoare-style proof annotations for the two functions `make_register` and `Promise.await` from figure 5. 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.

$$\mathit{isCallback}\ \mathit{cb}\ R \triangleq R \mathrel{-\!\!*} \mathit{ewp}\ (\mathit{cb}\ ())\ \langle\bot\rangle\ \{\top\}$$

$$\mathit{isBroadcastRegisterResult}\ r\ \mathit{cb}\ R \triangleq (\ulcorner r = \mathit{Called}\urcorner) \lor (\ulcorner r = \mathit{Registered}\ h\urcorner * \mathit{isBroadcastRegisterHandle}\ h\ \mathit{cb}\ R)$$

$$\mathit{isBroadcastRegisterHandle} : \mathit{Val} \to \mathit{Val} \to \mathit{iProp} \to \mathit{iProp}$$

Spec-BroadcastRegister

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

Spec-BroadcastTryCancel

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

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 $\mathit{promiseDone}\ \gamma$ and pass as the callback a `waker` function, which has the precondition $(\lambda\ v,\ \ulcorner v = ()\urcorner * \mathit{promiseDone}\ \gamma)$ as described above. The result of `Broadcast.register` is either a value *Called*, which expresses that it called the callback directly, or a register handle, which can be used to call `Broadcast.try_unregister`.

`Broadcast.try_unregister` will attempt to cancel a previous registration identified by the given *handle*. If the cancellation is successful, we get back the *isCallback* resource so that we can call the callback in `make_register`.

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 $\mathit{isRegister}\ \mathit{register}\ (\lambda\ v,\ \ulcorner v = ()\urcorner * \mathit{promiseDone}\ \gamma)$. Then, we get access to the $\mathit{promiseDone}\ \gamma$ 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.

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

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

$\left\{\begin{array}{l} PromiseInv * isPromise\ p * isBroadcast\ bcst * \\ (promiseDone\ \gamma \mathrel{-\!\!*} ewp\ (waker\ ())\ \langle\bot\rangle\ \{\top\}) \end{array}\right\}$

```
    let regres = Broadcast.register bcst waker in
```
[apply Spec-BroadcastRegister]

$\left\{\begin{array}{l} PromiseInv * isPromise\ p * isBroadcast\ bcst * \\ isBroadcastRegisterResult\ regres \end{array}\right\}$

```
    match regres with
```
[CA on regres]

---

1. $\{regres = None\}$

```
    | None -> ()
```
[by done]

---

2. $\left\{\begin{array}{l} PromiseInv * isPromise\ p * isBroadcast\ bcst * \\ regres = Some\ handle * isBroadcastRegisterHandle\ handle \end{array}\right\}$

```
    | Some handle ->
```
[open *PromiseInv*, lookup p using $isPromise\ p$]

$\left\{\begin{array}{l} \cancel{PromiseInv} * isBroadcast\ bcst * \\ isBroadcastRegisterHandle\ handle * PromiseState\ p\ \gamma\ \Phi \end{array}\right\}$

```
      match Atomic.get p with
```
[CA on *PromiseState*]

2.1. $\left\{\begin{array}{l} \cancel{PromiseInv} * isBroadcast\ bcst * \\ isBroadcastRegisterHandle\ handle * \\ p \mapsto Done\ result * promiseDone\ \gamma \end{array}\right\}$

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

$\left\{\begin{array}{l} isBroadcast\ bcst * isBroadcastRegisterHandle\ handle * \\ promiseDone\ \gamma \end{array}\right\}$

```
          if Broadcast.try_unregister handle
```
[apply Spec-BroadcastTryCancel, CA on return valu

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

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

```
          then waker ()
```
[by apply]

2.1.2. $\{\top\}$

```
          else ()
```
[by done]

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

```
        | Waiting _ -> ()
```
[close *PromiseInv*, by done]

Figure 10: Annotated proof of Spec-MakeRegister.

`let await (p: 'a t) : 'a =`	
$\{PromiseInv * isPromise\ p\}$	[open *PromiseInv*, lookup p using *isPromise* p]
$\{\cancel{PromiseInv} * isPromise\ p * PromiseState\ p\ \gamma\ \Phi\}$	
`  match Atomic.get p with`	[CA on *PromiseState*]
1. $\left\{\begin{array}{c}\cancel{PromiseInv}* \\ p \mapsto Done\ result * \square(\Phi\ result)\end{array}\right\}$	
`  | Done result ->`	[close *PromiseInv*]
$\{PromiseInv * \square(\Phi\ result)\}$	
`  result`	[by assumption]
2. $\left\{\begin{array}{c}\cancel{PromiseInv} * isPromise\ p* \\ p \mapsto Waiting\ bcst * isBroadcast\ bcst\end{array}\right\}$	
`  | Waiting bcst ->`	[close *PromiseInv*]
$\left\{\begin{array}{c}PromiseInv * isPromise\ p* \\ isBroadcast\ bcst\end{array}\right\}$	
`  let register = make_register p bcst`	[apply Spec-MakeRegister]
$\left\{\begin{array}{c}PromiseInv * isPromise\ p* \\ isRegister\ register\end{array}\right\}$	
`  perform (Suspend register);`	[protocol of *Suspend* with $(P := \lambda v,\ \ulcorner v = () \urcorner * promiseDone\ \gamma)$]
$\left\{\begin{array}{c}PromiseInv * isPromise\ p* \\ promiseDone\ \gamma\end{array}\right\}$	[open *PromiseInv*, lookup p using *isPromise* p]
$\left\{\begin{array}{c}\cancel{PromiseInv} * promiseDone\ \gamma \\ *PromiseState\ p\ \gamma\ \Phi\end{array}\right\}$	
`  match Atomic.get p with`	[CA on *PromiseState*]
2.1. $\left\{\begin{array}{c}\cancel{PromiseInv}* \\ p \mapsto Done\ result * \square(\Phi\ result)\end{array}\right\}$	
`  | Done result ->`	[close *PromiseInv*]
$\{PromiseInv * \square(\Phi\ result)\}$	
`  result`	[by assumption]
2.2. $\left\{\begin{array}{c}\cancel{PromiseInv} * promiseDone\ \gamma* \\ p \mapsto Waiting\ bcst * promiseWaiting\ \gamma\end{array}\right\}$	
`  | Waiting _ ->`	[specialize PS-Contra]
$\{\cancel{PromiseInv} * \bot\}$	
`  error "impossible"`	[by contradiction]

Figure 11: Hoare-style proof of Spec-Await.

2.2.6 Comparison of Logical State

$$Ready\ q\ \phi\ k \triangleq \forall v.\ \square\phi(v) \mathrel{-\!\!*} \triangleright PromiseInv\ q \mathrel{-\!\!*} \triangleright isQueue\ q\ (Ready\ q\ (\lambda w.\ w = ())) \mathrel{-\!\!*} ewp\ (k\ v)\ \langle\bot\rangle\{\_.\mathrm{True}\}$$

replace png with latex

Since our logical state definitions are based on a case study of de Vilhena, we want to give a short comparison of what had to be changed when adapting it to our model of Eio. First, in the original development the *Ready* predicate fulfills two roles.

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

PromiseInv and *isQueue* were both necessary as preconditions because they are not persistent and need to be passed around explicitly.

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

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

For the second use case of describing the continuations in a promise's waiting-queue we now have another specialized version of *Ready*. As explained in the next section, a broadcast has an invariant $P\ v \mathrel{-\!\!*}$ *ewp* (*callback* ()) $\langle \bot \rangle\ \{\top\}$ for all stored *callback*s. This is just *Ready* where $P\ v$ replaces $\Phi\ v$, and it is the same P as in the definition of the *Suspend* effect.

3 Verifying Eio's Broadcast

In this section we describe the *broadcast* data structure that Eio uses to implement of *promises*. Broadcasts are a customization of the recently developed *CQS* data structure [4]. 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. 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.

In the case of Eio an "execution context" is an Eio fiber. CQS is multithreaded by design, so fibers can use the adapted broadcast functions to synchronize with fibers running in another thread. In the following we describe the behavior of Eio's *broadcast*, highlight differences to the *original CQS*, and explain how we adapted the verification of the original CQS for our development.

3.1 Operations of Broadcast

The original CQS supports three operations that are interesting to us: *suspend*, *resume*, and *tryCancel*. The equivalent operations in a broadcast are *register*, *signalAll*, and *tryUnregister*, respectively. 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.

suspend/register In the original CQS, an execution context that wants to wait for an event performs a suspend operation. This operation creates and returns a new future that is used to stop execution 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 (fibers) to suspend until an event happens (a promise is fulfilled). So in the broadcast data structure, instead of returning a future, the register operation takes a callback as an additional argument and registers it to be called when the event happens.

resume/signalAll As a dual to suspend, a resume operation in the original CQS completes a single registered future so that the language runtime resumes the associated execution context. For broadcast, this is replaced by the signalAll operation, which invokes all callbacks that are registered with the data structure. Eio uses signalAll 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.

tryCancel/tryUnregister The semantics of the tryCancel operation do not markedly change. It can be used by an execution context to cancel the future returned by a suspend operation, so that it will not be completed by a call to resume. Analogously, tryUnregister tries to undo the registration of a callback, so that it will not be invoked in a call to signalAll. The operation will fail if a corresponding resume or signalAll happens first.

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 it is lock-free and fibers can run on different threads, there can be a race between concurrent register, tryUnregister, and signalAll operations. Possible interleavings and the necessity of the tryUnregister were explained in section 2.1.3.

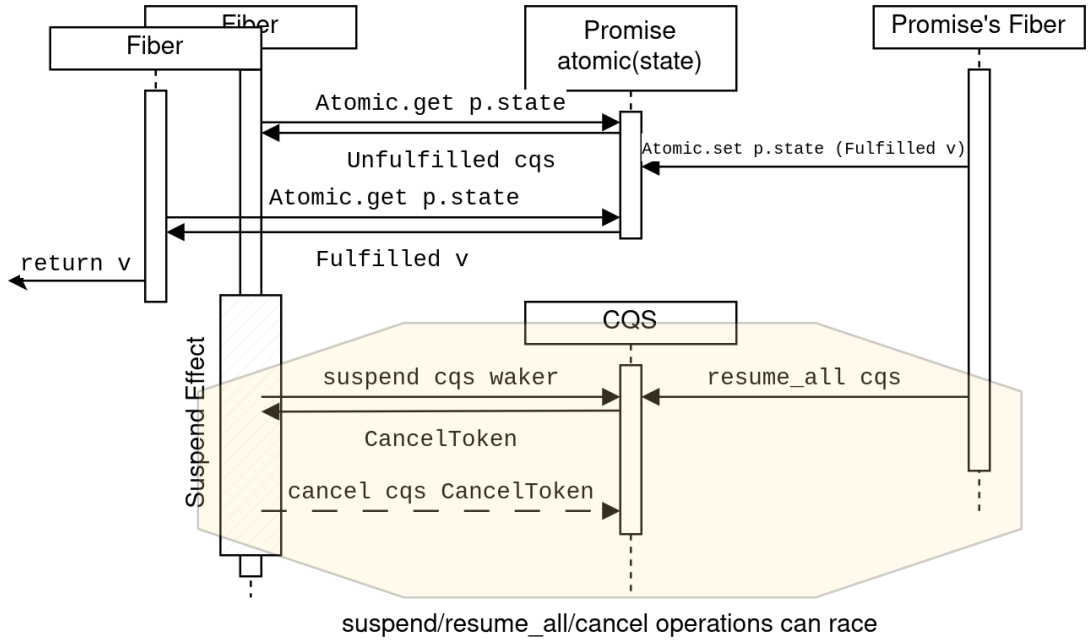

Figure 12: Usage of Broadcast in the Context of a Promise

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 logical state of all existing cells. The possible logical states for a single cell are shown in figure 13, where the arrows are annotated with the operation that causes a state transition.

The logical state of a cell is initialized to **EMPTY** when it is reached by the register pointer[7]. When a register and signalAll operation happen concurrently, they race to set the value of the empty cell. If the signalAll operation wins, it writes a token value into the cell and the logical state becomes **SIGNALLED**. The register operation can then read the token and invoke its callback directly. The logical state is thus **INVOKED**. If instead the register operation wins the race, it writes the callback into the cell, so the logical state takes the right path to **CALLBACK(waiting)**. Then there can be another race between concurrent tryUnregister and signalAll operations. Both try to overwrite the callback with a token value, which changes the logical state to **CALLBACK(invoked)** or **CALLBACK(unregistered)**, respectively, depending on the winner.

3.3 Verification of Broadcast

In the following we describe the specifications we proved for the functions implemented in Eio's `Broadcast` module. Note that all specifications 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 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

[6]Using segments instead of single cells in the linked list is an optimization to amortize the linear runtime of linked list operations
[7]As opposed to the original CQS, in broadcast the signal pointer will never overtake the register pointer.

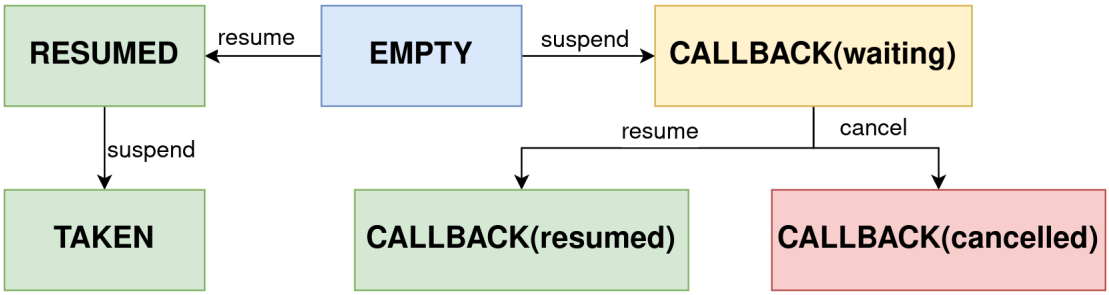

Figure 13: State Transition Diagram for a Single Cell.

position. Because of technical differences in handling these pointers between the original CQS implementation of the paper [4] 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. 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.

$$\textsc{Spec-BroadcastCreate}\quad \frac{\mathit{inv\_heap\_inv}}{\mathit{ewp}\ (\mathit{create}\ ())\ \langle \bot \rangle\ \{\mathit{bcst},\ \mathit{isBroadcast}\ \mathit{bcst} * \mathit{signalAllPermit}\}}$$

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. We instantiate R with *promiseDone* γ so that the callback transports the knowledge that the promise has been fulfilled. *isCallback* is not persistent because the callback must be invoked only once.

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

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 will immediately invoke the callback and return a `Called` value.

$$\textsc{Spec-BroadcastRegister}\quad \frac{\mathit{isBroadcast}\ \mathit{bcst} * \mathit{isCallback}\ \mathit{callback}\ R}{\mathit{ewp}\ (\mathit{register}\ \mathit{bcst}\ \mathit{callback})\ \langle \bot \rangle\ \{r,\ \mathit{isBroadcastRegisterResult}\ r\ \mathit{callback}\ R\}}$$

3.3.3 `Broadcast.try_unregister`

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

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

$$\frac{\textsc{Spec-BroadcastTryCancel} \qquad \mathit{isBroadcastRegisterHandle}\ h\ \mathit{cb}\ R}{\mathit{ewp}\ (\mathit{try\_unregister}\ h)\ \langle\bot\rangle\ \{b,\ \mathit{if}\ b\ \mathit{then}\ \mathit{isCallback}\ \mathit{cb}\ R\ \mathit{else}\ \top\}}$$

3.3.4 `Broadcast.signal_all`

To call `Broadcast.signal_all` the unique *signalAllPermit* resource is needed, along with a duplicable R, so that it can be used to invoke multiple callbacks. The function does not return any resources because its only effect is making an unknown number of fibers resume execution, which we cannot easily formalize in Iris.

$$\frac{\textsc{Spec-BroadcastSignalAll} \qquad \mathit{isBroadcast}\ \mathit{bcst} * \square\, R * \mathit{signalAllPermit}}{\mathit{ewp}\ (\mathit{signalAll}\ \mathit{bcst})\ \langle\bot\rangle\ \{\top\}}$$

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.

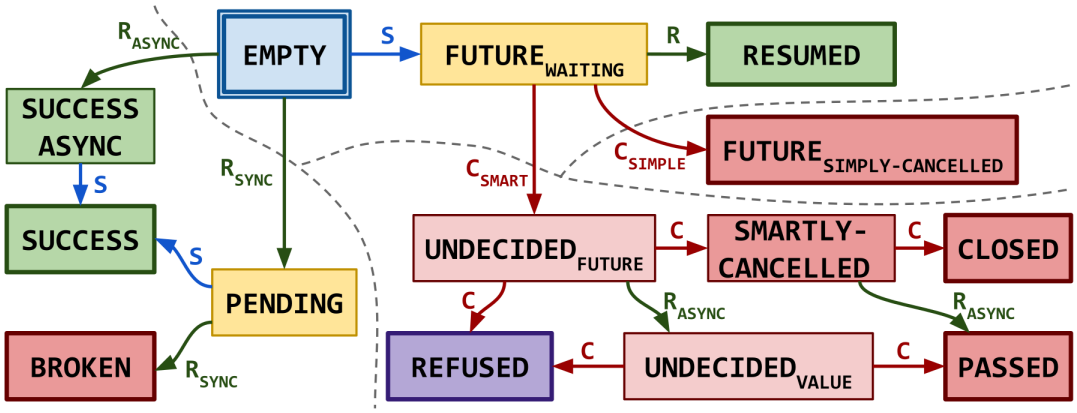

Figure 14: Cell States in the Original CQS from [4] (page 42).

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.

```
let fiber1 () =
  let ctx = perform (GetContext ()) in
  let ctx2 = perform (GetContext ()) in
  assert (ctx.tlv == ctx2.tlv)

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

Figure 15: Constructed example of safety for thread-local variables.

4 Extending the Scheduler with Thread-Local Variables

So far we have looked at a protocol *Coop* that has two effects which suffice to model fibers that can suspend and fork off new fibers. But in Eio fibers 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. Two example use-cases are per-scheduler 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, it is still possible for fibers to arbitrarily modify the whole dictionary, so for demonstration purposes we model thread-local variables as a single mutable reference that is part of the context record: `ctx.tlv`. Properties we want to prove about thread-local variables are:

1. Each time a fiber performs a *GetContext* effect it will receive 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 15. Note that these are only the most basic properties showing that `ctx.tlv` acts like a normal reference, but one that can be accessed via an effect. To enable modular proofs of concrete fibers using thread-local variables, we include in our logical state a predicate T on the stored value that can be instantiated by fibers as needed.

4.1 Changes to Logical State

To handle thread-local variables in our development we must change both the implementation and logical state definitions. The necessary changes to the implementation are trivial, so we just refer to the mechanization[8]. For the logical state we define *fiberResources* that a fiber receives when it starts running and relinquishes when it stops. The new definitions are described in figure 16. *tlvAg* δ *tlv* is used to show the uniqueness of the location *tlv*. *isFiberContext* δ *tlv* represents the context that is tracked for each fiber, where δ is a shorthand for multiple ghost names. It expresses that the location *tlv* is a thread-local variable which maps to some value v satisfying T. The predicate T is hidden behind a *savedPred* indirection to make the mechanization easier. *fiberResources* δ is then used to abstract away the concrete location *tlv*. Finally, we must change the definition of *Ready* to require *fiberResources* as a precondition because it is needed to invoke the continuations saved in the scheduler's run-queue.

The effect protocols of *Fork* and *Suspend* are amended so that they pass *fiberResources* from a fiber to the scheduler and from there to the next running fiber via the protocol pre- and postconditions as shown

[8] TODO insert link

$$
\begin{aligned}
\mathit{tlvAg}\ \delta\ \mathit{tlv} &\triangleq \boxed{\mathit{agree}(\mathit{tlv})}^{\delta} \qquad \text{Persistent}(\mathit{tlvAg}\ \delta\ \mathit{tlv}) \\
\mathit{isFiberContext}\ \delta\ \mathit{tlv} &\triangleq \mathit{tlvAg}\ \delta\ \mathit{tlv} * \exists T\ v.\ \mathit{tlv} \mapsto v * \mathit{savedPred}\ \delta\ T * T\ v \\
\mathit{fiberResources}\ \delta &\triangleq \exists\ \mathit{tlv}.\ \mathit{isFiberContext}\ \delta\ \mathit{tlv} \\
\mathit{Ready}\ \delta\ f &\triangleq \mathit{fiberResources}\ \delta\ \mathrel{-\!\!*}\ \mathit{ewp}\ (f\ ())\ \langle \bot \rangle\ \{\top\}
\end{aligned}
$$

Figure 16: Logical State Definitions for the Verification of Scheduler & Promise Modules

in figure 17. The *Fork* effect now also passes the concrete reference that should be used as the thread-local variable of the new fiber. A fiber uses the *GetContext* effect to receive the fiber context value and a copy of *tlvAg*. This is used to show that the reference *ctx.tlv* is equal to the one from *fiberResources* that the fiber already owns so that the contained points-to predicate can be used.

The crux is that now the protocol $\mathit{Coop}\ \delta$ is parameterized by the ghost name δ that identifies the thread-local variable. This so that both the fiber and the scheduler agree on this ghost name.

$$
\begin{aligned}
\mathit{Coop}\ \delta \triangleq \quad & \mathit{Fork}\ \#\ !\ \mathit{tlv}\ e\ ((\mathit{tlv}, e))\ \big\{\mathit{fiberResources}\ \delta\ T * \mathit{tlvAg}\ \delta\ \mathit{tlv} * \\
& \qquad \triangleright\ (\mathit{fiberResources}\ \delta\ T \mathrel{-\!\!*} \mathit{ewp}\ (e)\ \langle \mathit{Coop} \rangle\ \{\mathit{fiberResources}\ \delta\ T\})\big\}. \\
& \quad ?\ ()\ \{\mathit{fiberResources}\ \delta\ T\} \\
& \mathit{Suspend}\ \#\ !\ \mathit{reg}\ P\ (\mathit{reg})\ \{\mathit{fiberResources}\ \delta\ T * \mathit{isRegister}\ \mathit{reg}\ P\}. \\
& \quad ?\ y\ (y)\ \{\mathit{fiberResources}\ \delta\ T * P\ y\} \\
& \mathit{GetContext}\ \#\ !\ ()\ \{\top\}.\ ?\ \mathit{ctx}\ (\mathit{ctx})\ \{\mathit{tlvAg}\ \delta\ \mathit{ctx.tlv}\}
\end{aligned}
$$

Figure 17: Definition of extended $\mathit{Coop}\ \delta$ protocol with *Fork*, *Suspend*, and *GetContext* effects.

These changes suffice to prove the safety of the two examples in figure 15.

5 Evaluation

6 Conclusion

6.1 Future Work

- Transport lemma from heap lang to extended languages to avoid adding many heap lang features to these languages.
- Extending the case study with more operating system primitives.
- There are other approaches to concurrency in OCaml using effect handlers without the higher-order suspend effect. It would be interesting to also study them and see if the formalization turns out to be easier.

Appendix

A Translation Table

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

B Towards A multithreaded Scheduler

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

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

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

Adding Invariants to Hazel

Invariants in Iris are used to share resources between threads. They encapsulate a resource to be shared and can be opened for a single atomic step of execution. During this step the resource can be taken out of the invariant and used in the proof but at the end of the step the invariant must be restored.

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

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

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

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

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

How we add support for the iInv tactic to use invariants more easily.

Adding Multi-Threading to Hazel

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

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

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

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

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

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

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

The scheduler `run` and therefore also the `spawn_scheduler` function don't have interesting return values, so this part of the specification is uninteresting. What is more interesting is that they encapsulate the possible effects the given function `f` performs.

C A Note on Cancellation

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

References

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

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

[3] 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.

[4] 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.

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

[6] 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.