Verifying an Effect-Based Cooperative Concurrency Scheduler in Iris

Adrian Dapprich

Feb 1, 2024

Sections

1. Introduction

- As a motivation for the work: program verification, safety and why we care about it.
- Iris and how we can prove safety
- Eio and how it provides concurrency primitives
- Effect Handlers with a simple example
- Effect Handler formalization in Iris using protocols

2. Simple Scheduler

- The simplified code of the mock scheduler that Eio provides (actual schedulers differ per OS and intergrate with OS primitives)
- What is the difference between this scheduler and the one from Paolo's paper
  The gist is that using a concurrent queue and handling promises in the fibers allows many simplifications in the scheduler & the logical state.
- Explain how the Fork/Suspend effect work and how fibers use them.
- Explain how the scheduler implements these effects.
- What are the safety concerns in this implementation (mainly in the implementation for await)
- What logical state in Iris do we use to model the behavior.
- What are some interesting parts of the proofs.

3. Towards A Multi-Threaded Scheduler

OCaml 5 added not only effect handlers but also the ability to use multiple threads of execution, which are called *domains* (in the following we use the terms interchangeably). Each domain in OCaml

5 corresponds to one system-level thread and the usual rules of multi-threaded execution apply, i.e. domains are preemtively scheduled and can share memory. Eio defines an operation to make use of multi-threading by forking off a new thread and running a separate scheduler in it. So while each Eio scheduler is only responsible for fibers in a single thread, fibers can await and communicate with fibers running in other threads.

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

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

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

Adding Invariants to Hazel

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

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

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

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

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

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

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

Adding Multi-Threading to Hazel

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

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

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

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

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

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

```
EWP e <| \bot |> { \top }
---------------------------------------
EWP (Fork e) <| \Phi |> { x, x = () }
```

[TODO explain how that proof works.]

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

```
Lemma spawn_spec (Q : val → iProp Σ) (f : val) :
  EWP (f #()) <| \bot |> {{ Q }} -∗ EWP (spawn f) {{ v, ∃ (l: loc), ⌜ v = #l ⌝ ∗ join_handle l Q }}.

Lemma join_spec (Q : val → iProp Σ) l :
  join_handle l Q -∗ EWP join #l {{ v, Q v }}.

Definition spawn_inv (γ : gname) (l : loc) (Q : val → iProp Σ) : iProp Σ :=
  ∃ lv, l ↦ lv ∗ (⌜lv = NONEV⌝ ∨
                  ∃ w, ⌜lv = SOMEV w⌝ ∗ (Q w ∨ own γ (Excl ()))).

Definition join_handle (l : loc) (Q : val → iProp Σ) : iProp Σ :=
  ∃ γ, own γ (Excl ()) ∗ inv N (spawn_inv γ l Q).
```

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

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

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

```
Lemma spawn_scheduler_spec (Q : val -> iProp Σ) (f: val) :
  promiseInv -∗ EWP (f #()) <| Coop |> {{ _, True }} -∗
    EWP (spawn_scheduler f) {{ _, True }}.
```

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

4. Verifying Eio's Customized CQS

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

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

Operations of CQS

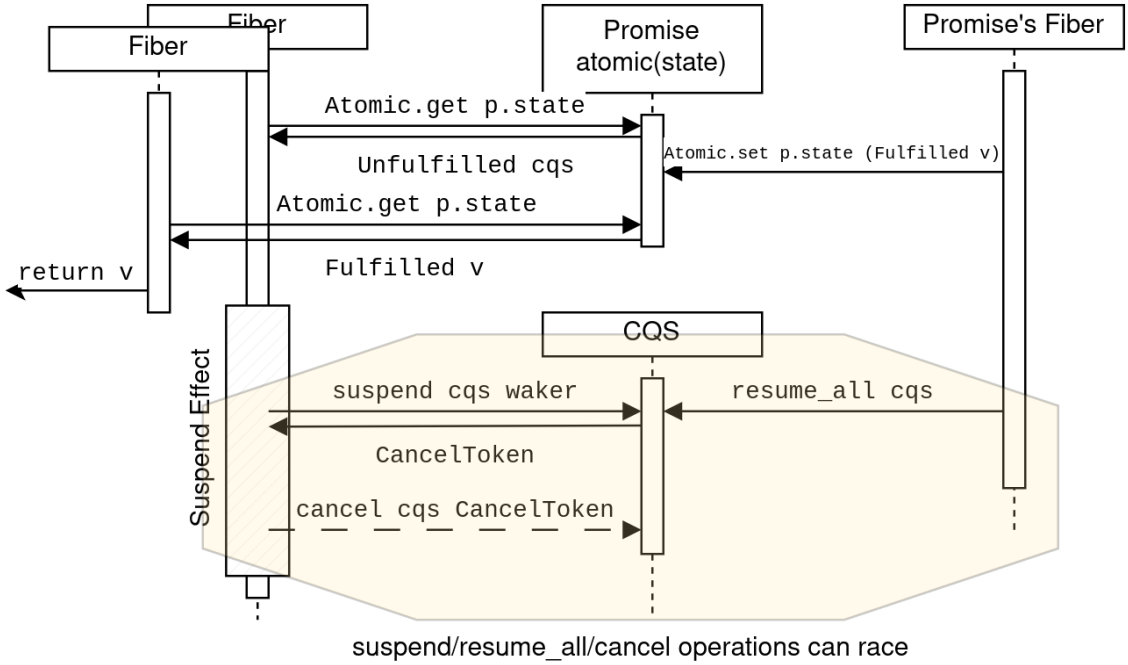
The original CQS supports three operations that are interesting to us. In a *suspend operation* the requesting execution context wants to wait until signalled. It places a handle to itself in the datastructure and is expected to stop execution afterwards. But before it actually stops execution it can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

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

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

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

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



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

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

```
type t
type request

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

Implementation and Logical Interface of CQS

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

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

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

Verification of the `Broadcast` Module

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

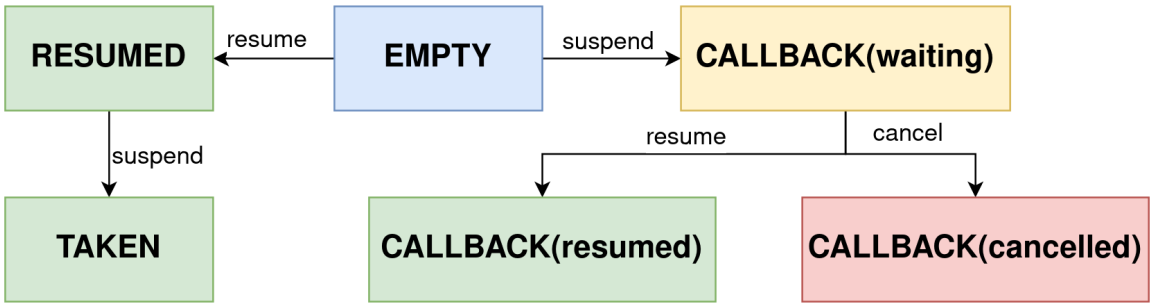
The first major change was replacing the future-based interface of the suspend operation with a callback-based interface. In the original CQS, performing a suspend operation returns a new future, which is also inserted as the handle into the queue. The execution context can then use the future to stop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

This changes the logical state of CQS only slightly. The original CQS tracked the state of the future for each cell and managed *futureCancellation* and *futureCompletion* tokens. In the customized CQS we analogously track the state of the callback for each cell and manage *callbackInvokation* and *callbackCancellation* tokens.

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

Aside: Implementation of resume_all
Eio implements resume_all by atomically increasing the *resume pointer* by some number n, instead of just 1 like in the original resume.
Because of technical differences between the infinte array implementation in the CQS mechanization & the infinite array implementation
Instead, I actually defined resume_all simply as a loop over a resume operation.
Since resume_all is only called once I posit that this verification is still valid but I still want to verify Eio's resume_all and remove this aside.

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



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

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

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

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

```
Theorem suspend_spec γ q k:
 {{{ is_cqs γ q ∗
     is_waker V' k }}}
   suspend q k
 {{{ v, RET v; ⌜v = NONEV⌝ v
          ∃ γk r, ⌜v = SOMEV r⌝ ∗
                is_suspend_result γ γk r k }}}.
```

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

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

```
Theorem try_cancel_spec γ q γk r k:
  {{{ is_cqs γ q ∗
      is_suspend_result γ γk r k }}}
    cancel r
  {{{ (b: bool), RET #b; if b then is_waker V' k
                    else True }}}.
```

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

```
Theorem resume_all_spec γ q:
  {{{ is_thread_queue γ q ∗
      □ V' ∗
      resume_all_permit γ }}}
    resume_all q
  {{{ RET #(); True }}}.
```

Features Removed from Original CQS

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

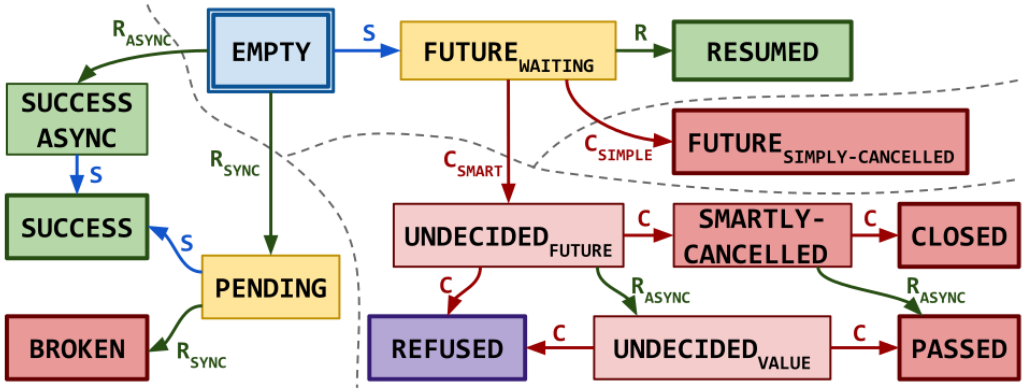


Figure 14: The state transition system for a single cell from the logical perspective.

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

Verification of the **Promise** Module

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

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

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

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

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

Aside: All wakers are eventually called.
This statement is purely based on a reading of the code. It might be possible to formally prove this with an approach like Iron [ref] or Transfinite Iris [ref] because it is a liveness property.
But for the Iron approach it is unclear to us how to formulate the linearity property.

5. Extending the Scheduler with Thread-Local Variables

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

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.

Evaluation

Conclusion

Bibliography

- A Formally-Verified Framework For Fair Synchronization in Kotlin Coroutines
- A Separation Logic for Effect Handlers
- A Type System for Effect Handlers and Dynamic Labels
- Retrofitting Effect Handlers onto OCaml
- Retrofitting Parallelism onto OCaml