Verifying an Effect-Based Cooperative Concurrency Scheduler in Iris

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

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.

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
Instead of modelling atomic variables in Hazel, we continue to use normal references because
```

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
```

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

Definition spawn_inv (  : gname) (l : loc) (Q : val → iProp Σ) : iProp Σ :=
    lv, l   lv ∗ ( lv = NONEV
                   w,  lv = SOMEV w  ∗ (Q w   own   (Excl ()))).

Definition join_handle (l : loc) (Q : val → iProp Σ) : iProp Σ :=
    , own   (Excl ()) ∗ inv N (spawn_inv   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 :=
  ( : "f",
    let: "new_scheduler" := ( : <>, 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.

Verifying Eio's Customized CQS

CQS [ref] is a formally verified implementation of a synchronization primitive which allows execution contexts to wait until signalled. The exact nature of the execution context is abstracted away but it must support stopping and resuming execution. 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 uses a customized version of CQS adapted from the one presented in the paper [ref] in the form of the `Broadcast` module mentioned in section 1, which allows fibers to wait until another fiber signals an event. In this section we mainly describe the behavior of Eio's *customized CQS* (or *broadcast*) and highlight differences to the *original CQS*. If something applies to both the customized and original version we just use the term *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 so it places a handle to itself in the datastructure. Afterwards, it is supposed to stop execution but some time passes between placing the handle into the datastructure and actually stopping execution. During this time the execution context can try to cancel the suspend request using the *cancel operation*. Finally, a *resume operation* takes one handle out of the datastructure and tries to resume execution of the associated execution context, passing it the value that was provided to the operation. This fails if the suspend request had already been successfully cancelled.

Eio's customized CQS supports a new operation called the *resume all operation*. As the name implies, it is basically a resume operation that applies to all currently saved handles. Eio's implementation of CQS – in the form of the `Broadcast` module – is used to implement the *promise* functionality, which allows fibers to wait for the complection of another fiber and retrieve its final value. When a promise is fulfilled all waiting fibers must be signalled, which is why this operation was added.

```
Aside: Implementation of resume_all
Eio has a custom implementation of resume_all on the level of the infinite array (more on th
Because of technical differences between the infinte array implementation in the CQS mechani
So our verification work we actually define resume_all simply as a loop over the original re
By analyzing the code of Eio we conclude that the Eio's resume_all and our resume_all behave
If time permits we will verify Eio's resume_all and remove this aside.
```

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. Since the promise could have been fulfilled in the meantime, the fiber then again atomically loads the state of the promise.

- If it has not been fulfilled the fiber does not need to do anything because it will eventually be resumed by a resume all operation invoking the `waker` it placed into the broadcast.
- 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 `waker`s **before** the `waker` of this fiber was placed in the broadcast. Then this `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 placed 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 as we saw in section 1 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 pr
like Iron [ref] or Transfinite Iris [ref] because it is a liveness property.
```

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

First we want to mention that the original CQS supports a couple of additional features like a synchronous mode for suspend and resume, and also a smart cancellation mode. These features blow up 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. 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.

Also, this reduced the state space of the original CQS shown below (taken from the original paper) to something more manageable for us when understanding the proofs. [TODO clean up diagram and replace FUTURE by CALLBACK]

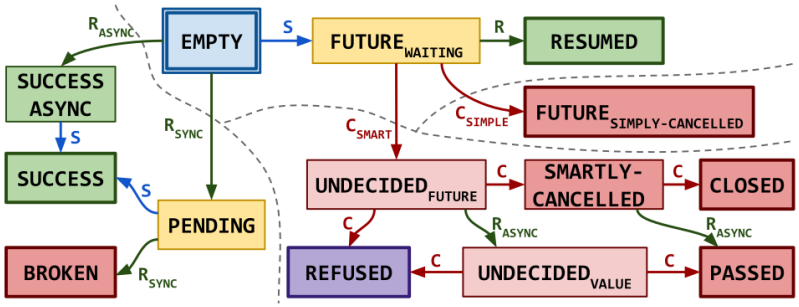

Figure 14: The state transition system for a single cell from the logical perspective.

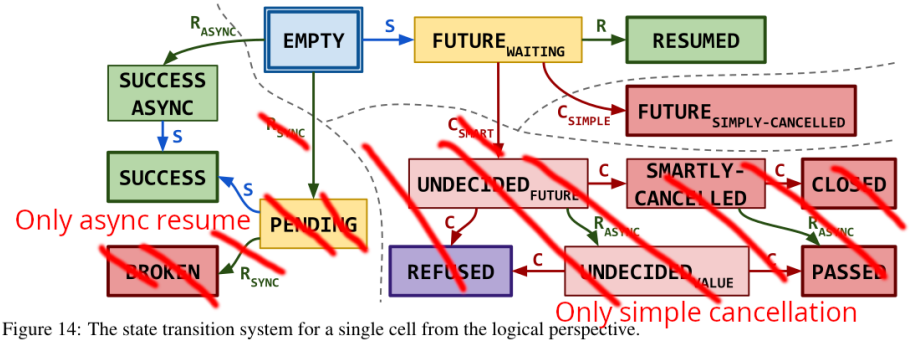

Figure 14: The state transition system for a single cell from the logical perspective.

The first major change that we did was replacing the future-based interface with a callback-based interface. In the original CQS, doing a suspend operation creates a future, places it in the datastructure as the handle and returns it. The execution context can then use the future to stop execution because it is assumed there is a runtime that allows suspending until the completion of a future. But Eio cannot use this future-based interface because it uses the customized CQS to *build* the runtime that allows fibers to suspend until the completion of a promise. Instead, Eio implements CQS with a callback-based interface where the fiber doing the suspend operation passes in a callback as the handle and afterwards the fiber implicitly stops execution. Doing a resume operation analogously invokes the callback, instead of completing the future. Eio's *promise* uses the `waker` from the `Suspend` effect as the callback so that on invokation the fiber is placed in the run queue and will resume execution.

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