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 stop execution until signalled. The nature of the execution context is kept abstract but it must support stopping execution and resuming with some value. 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 custom version of CQS adapted from the paper [ref] in the form of the `Broadcast` module from section 1. In this section we mainly describe the behavior of Eio's *customized CQS* (or `Broadcast` module), highlight differences to the *original CQS*, and discuss 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*.

Operations in 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 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 difference execution context. It 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 *resume operation*. This fails if the *suspend operation* 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 to implement Eio's *promise* so that **all** waiting fibers can be signalled when a promise is fulfilled.

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*. 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.

There is a logical resource called the `thread_queue_state n`, which tracks the number `n` of active cells, which is the length of the queue. In normal usage of CQS, there is supposed to be an atomic variable in some outside scope which encodes the length of the queue in its value and this logical resource is then kept in an associated invariant. Changing the logical state of the queue, i.e. its length, is done using *enqueue* and *dequeue registration* operations.

In the case of Eio, however, the exact length of the queue is irrelevant because the *resume-all operation* will always set the length to 0, so in our customized version we keep this logical resource in the invariant of CQS itself. This somewhat simplifies the usage of CQS proofs and also moves the *enqueue* and *dequeue registration* out of the public API because they are now done internally.

Specification of the Operations In the following we describe the specifications we proved for the three operations `suspend`, `cancel` and `resume_all`, 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.

First we want to mention that the original CQS supports a 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. 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.

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

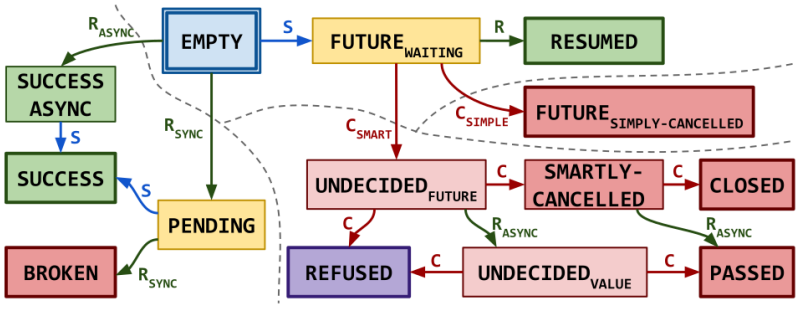

Figure 14: The state transition system for a single cell from the logical perspective.

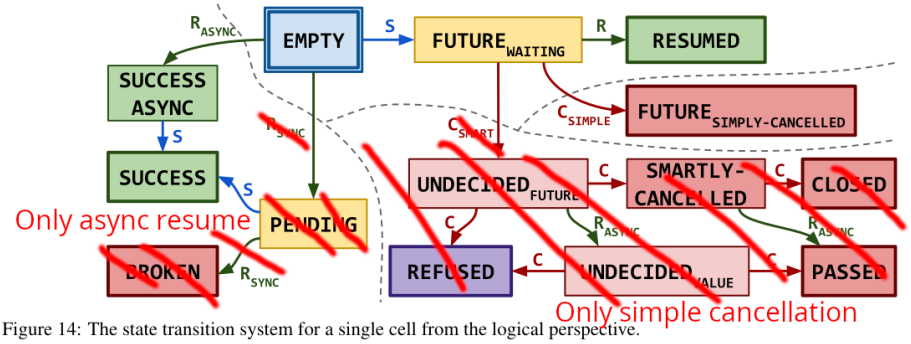

Figure 14: The state transition system for a single cell from the logical perspective.

The first major change was replacing the future-based interface of the suspend operation with a callback-based interface. In the original CQS, performing a suspend operation creates a future, saves 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 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 performing the suspend operation passes in a callback as the handle and afterwards the fiber implicitly stops execution. Performing a resume operation analogously invokes the callback, instead of completing the future. Concretely, Eio's *promise* uses the `Suspend` effect to stop execution of

the fiber and passes the `waker` as the callback so that when it is invoked the fiber is placed back into the run queue of its scheduler.

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`.

```
Aside: Implementation of resume_all
Eio implements resume_all by atomically increasing the *resume pointer* by some number n, ir
Because of technical differences between the infinte array implementation in the CQS mechani
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 sti
```

suspend operation The big `is_thread_queue` resource with all the ghost names identifies `e` and `d` as a CQS (instead of just one value, the queue is represented in terms of the *suspend* and *resume pointers*). 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`. For Eio we instantiate `V'` with `promise_state_done p` so that the callback transports the knowledge that the promise has been fulfilled. `is_waker` is non-duplicable because the callback must be invoked only once and it might be accessed from a different thread.

If there is a concurrent *resume-all operation*, the callback is invoked immediately and nothing is returned. Otherwise the callback is pushed to the queue and the `is_thread_queue_suspend_result` (TODO too long, rename) resource acts as the cancellation permit.

```
Theorem suspend_spec a tq e d res e d k:
  {{{ is_thread_queue a tq e d res e d *
      is_waker V' k }}}
    suspend e k
  {{{ v, RET v; v = NONEV
                  k v', v = SOMEV v' *
                        is_thread_queue_suspend_result tq a k v' k }}}.
```

cancel operation The specification of the *cancel operation* is a lot simplified compared to the original. The `is_thread_queue_suspend_result` resource

is used as a permission token and in order to find the callback that should be cancelled from the queue. If the callback had already been invoked the operation returns `false` and no resources are returned to the caller. At this point the caller trusts that the callback will be invoked at some point but as discussed above we do not express this in the specification. Otherwise, the cell is set to a cancelled state and the permission to invoke the callback is returned.

```
Theorem try_cancel_spec a tq e d res e d k r k:
  {{{ is_thread_queue a tq e d res e d *
      is_thread_queue_suspend_result tq a k r k }}}
    try_cancel r
  {{{ (b: bool), RET #b; if b then
                           is_waker V' k *
                           cell_is_cancelled k
                 else True }}}.
```

resume all operation The specification of the *resume-all operation* is also a lot simplified compared to the specification of the original *resume operation* because removing unused features from CQS reduced the state space and because cancelled cells are simply ignored by *resume-all*. 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.

Again since we do not prove linear usage of callbacks there are no resources returned. The function only knows that some number of callbacks were successfully invoked, but the caller cannot do anything with this information.

```
Theorem resume_all_spec a tq e d res e d n:
  {{{ is_thread_queue a tq e d res e d *
       V' *
      resume_all_permit res }}}
    resume_all e d
  {{{ RET #(); True }}}.
```

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