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` 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 the basic building blocks for using invariants. In the following we explain how we added a multi-threaded operational semantics and enabled the use of invariants.

```
Note: OCaml 5's Memory Model
In the OCaml 5 memory model, *atomic variables* are needed to share memory without introduci
Instead of modelling atomic variables in Hazel, we continue to use normal references as the
```

Adding Invariants to Hazel

Invariants in Iris are used to share resources between threads. They encapsulate the 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.

An Iris language must support the usage of invariants and Hazel did already have the basic elements of support. It defined a ghost cell to hold the invariants and proved an invariant access lemma which allows opening an invariant provided the current expression is atomic. We only had to provide proofs for which evaluation steps are atomic in order to use invariants. 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.

[TODO something about effects?] [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` and `Join` operations on top and prove their specifications. For `Join` we already need invariants as the point-to assertion for the done flag must be shared between the two threads.

```
[TODO specs for CAS & Join]
[TODO logical state to prove join]
```

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 which is based on the Eio and 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. [todo how exactly does Eio do it? it's probably not spinning but using some kind of OS poll feature to wake up a blocked fiber]

```
[TODO spec for spawning new scheduler]
[TODO explain how it follows directly from definition of fork & join]
```

Verifying an Adaptation of CQS

- What is CQS used for
- What is the logical state used for the verification
- How we adapt CQS (use promises & and resume_all)
- How we change the logical state to match our adapted version.

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