Composing Schedulers using Effect Handlers

Deepali Ande
IIT Madras, India

KC Sivaramakrishnan
IIT Madras, India

Abstract

OCaml 5 introduces effect handlers as a mechanism for concurrent programming. With effect handlers, concurrency can be expressed in *direct-style* rather than in monadic-style as in Lwt [9] and Async [2]. Rather than baking in the notion of a thread scheduler, the compiler exposes delimited continuations, with the idea that different libraries may implement their own concurrency primitive, with their own schedulers. Under this setting, given that the notion of a concurrent *task* is tied to a particular scheduler, it is challenging to allow tasks from different schedulers to interact with each other. If this problem is not solved, the ecosystem runs the risk of repeating the schism between Lwt and Async in OCaml today.

In this paper, we observe that the composability of effect handlers permits composability of schedulers. The key idea is that we can use effect handlers to define a uniform interface for suspending and resuming tasks, which is implemented by each of the schedulers. On top of this, we define scheduler-agnostic synchronization structures that allows tasks from different schedulers to interact. We also report on how this mechanism can be extended to capture the notion of task cancellation that extends across different schedulers.

1 The challenge of scheduler composition

OCaml 5 introduces support for shared-memory parallelism and native concurrency in OCaml. OCaml is the first industrial-strength language to incorporate effect handlers. OCaml users may define their own domain-specific schedulers using effect handlers.

As an example, Eio [4] is an effect-based direct-style IO library for OCaml 5. The library aims to be a replacement for monadic style libraries such as Lwt and Async. Unlike monadic concurrency libraries, Eio uses effect handlers to implement concurrent tasks. Eio introduces structured concurrency using *switches* where groups of tasks may be waited on together for completion. While Eio supports multiple domains, to retain the simplicity of program reasoning, Eio does not multiplex tasks across multiple domains. Another user of effect handlers is Domainslib [3], a library for nested parallel programming. Domainslib tasks are created and their results consumed using async/await primitives. Domainslib scheduler uses a lock-free work-stealing queue to distribute tasks across the available workers.

It is conceivable that an application may want to use both Eio and Domainslib at the same time. For example, a database management system in OCaml 5 may want to handle concurrent client sessions using Eio while offloading query processing to Domainslib. Another example is the Tezos blockchain baker nodes which may use Eio for communicating with the rest of the network while offloading serialisation and cryptography to Domainslib. Unfortunately, while both Eio and Domainslib use effect handlers to implement tasks, they have incompatible mechanisms for suspending and resuming threads.

Consider the example of an Eio task performing `await` on a Domainslib promise. If the Eio and Domainslib handlers (`run` functions) are arranged such that the Domainlib handler encloses the Eio handler,

```
Domainslib.run (fun _ ->
  Eio.run (fun _ ->
    ...
    Domainslib.await ...;
    ...))
```

then, the `await` function call may block the entire Eio scheduler since `await` is handled at `Domainslib.run` which encloses `Eio.run`. We call this issue as *vertical composition* of schedulers. Alternatively, the application might have Domainslib and Eio schedulers running on different domains. If an Eio task performs `await`, then there will be no Domainslib handler in scope, and the program will raise `Unhandled` exception. We call this the *horizontal composition* problem.

There are already several alternative concurrency libraries being proposed for OCaml 5 [1]. Unless the issue of scheduler composition is addressed, we may have the situation where different concurrency libraries remain incompatible, leading to either the library authors maintain compatibility with several different concurrency libraries or (more likely) a split in the library ecosystem.

2 Proposal

We propose that the libraries agree on a common interface that captures the notion of suspending and resuming tasks. Each library implements this interface. Such an interface may be used to implement blocking synchronisation structures such as MVars and Channels. The simplified version of the interface is given below:

```
type 'a resumer = 'a -> unit
type _ Effect.t +=
  Suspend : ('a resumer -> unit) -> 'a t
```

Conference'17, July 2017, Washington, DC, USA
.

Whenever the synchronization structure (say MVar or Channel) wants to block the current user-level thread on a particular condition, it performs Suspend f. At the handler, i.e, the scheduler, f is applied to a "resumer" function. This resumer function, when applied to the result, adds the blocked thread to the scheduler so that it resumes with the result. The blocking operation squirrels away this resumer in the synchronization structure's state. Now that the current thread is blocked in the synchronization structure, the handler switches control to the next runnable thread in the scheduler. The full-fledged interface will also need to handle racy access to the synchronization structure from multiple domains (the suspend operation may fail) and the cancellation of suspended tasks (the resume operation may fail).

We have built a prototype of the scheduler-agnostic MVar implementation implementation[1]. As one can see in the MVar implementation, the MVar does not refer to any concrete scheduler. The prototype shows how to compose two schedulers where one schedules tasks in FIFO order and the other schedules them in LIFO order. The schedulers are composed horizontally, and the example shows that the tasks from these two schedulers can coordinate using the MVar implementation without blocking other tasks from their respective schedulers. We have also experimented with larger examples that utilises Domainslib and Eio together in the same application.

3 Related work

OCaml is not the first language to support concurrency natively. Undelimited continuations using `call/cc` are well known in the literature and are implemented by various language in the Lisp family as well as Standard ML compilers such as SML/NJ and MLton. While there have been many implementation of concurrency libraries such as Concurrent ML [7], the focus is not composition of several schedulers.

Language such as Go and GHC Haskell support lightweight threads. Go has goroutines and GHC Haskell has threads. These lightweight tasks are implemented directly by the compiler and the runtime system, with the scheduler being part of the runtime system. Given that the language bakes in the concurrency model, the problem of composition does not exist. The downside of this approach is that the runtime does not specialise the scheduler for specific purposes as is the case with Domainslib and Eio.

The Rust programming language does not itself impose a concurrent and parallel programming model, leaving the libraries to implement them. Tokio [8] is an asynchronous runtime for Rust that has similar goals as Eio in OCaml. Rayon [5] is a data-parallel programming library that has similar goals as Domainslib in OCaml. Using these libraries has the same challenges as the ones that we are trying to solve in this work. However, unlike our proposal which aims to develop a scheduler agnostic solution that works across any scheduler that implements this interface, Rust's solution is a *bespoke* one-shot channel [6] to communicate between Tokio and Rayon. These bespoke solutions by definition do not work with other concurrency libraries.

[1] https://github.com/kayceesrk/code-snippets/tree/master/scheduler_parateric_mvar

References

[1] Affect 2022. *Composable concurrency primitives for OCaml 5.0.* https://erratique.ch/software/affect

[2] Async 2022. *A hybrid approach to asynchronous programming.* https://opensource.janestreet.com/async/

[3] Domainslib 2022. *A library for nested parallel programming.* https://github.com/ocaml-multicore/domainslib

[4] Eio 2022. *Effects-based direct-style IO for multicore OCaml.* https://github.com/ocaml-multicore/eio

[5] Rayon 2022. *A data parallelism library for the Rust programming language.* https://docs.rs/rayon/latest/rayon/

[6] Rayon Tokio Crate 2021. *Rayon Tokio Crate.* https://github.com/andybarron/tokio-rayon

[7] John H. Reppy. 2007. *Concurrent Programming in ML.* Cambridge Univ. Press.

[8] Tokio 2022. *An asynchronous runtime for the Rust programming language.* https://tokio.rs/

[9] Jérôme Vouillon. 2008. Lwt: A Cooperative Thread Library. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML* (Victoria, BC, Canada) *(ML '08)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/1411304.1411307