Experiences with Effects

Thomas Leonard Craig Ferguson Patrick Ferris Sadiq Jaffer Tom Kelly KC Sivaramakrishnan Anil Madhavapeddy

OCaml Labs

The OCaml Users and Developers Workshop, Aug 2021

Overview

- Domains / effects / typed effects
- Introduction to effects
- Case study: Converting the Angstrom parser
- Eio concurrency library

Introduction to effects

- Resumable exceptions
- Multiple stacks

```
effect Foo : int → int

try
  println "step 1";
  let x = perform (Foo 2) in
  println "step %d" x
with effect (Foo n) k →
  println "step %d" n;
  continue k (n + 1)
```

Advantages of effects

- No difference between sequential and concurrent code.
  - No special monad syntax.
  - Can use `try`, `match`, `while`, etc.
  - No separate lwt or async versions of code.
- No heap allocations needed to simulate a stack.
- A real stack means backtraces and profiling tools work.

Case study: Angstrom

https://github.com/inhabitedtype/angstrom/

- A library for writing parsers
- Designed for network protocols
- Strong focus on performance

A toy parser

```
type 'a parser = state → 'a

let any_char state =
  ensure 1 state;
  let c = Input.unsafe_get_char state.input state.pos in
  state.pos <- state.pos + 1;
  c

let (*>) a b state =
  let _ = a state in
  b state
```

The Angstrom parser type

```
module State = struct
  type 'a t =
    | Partial of 'a partial
    | Lazy    of 'a t Lazy.t
    | Done    of int * 'a
    | Fail    of int * string list * string
  and 'a partial =
    { committed : int;
      continue  : Bigstringaf.t →
        off:int → len:int → More.t → 'a t }
end
type 'a with_state = Input.t → int → More.t → 'a
type 'a failure =
  (string list → string → 'a State.t) with_state
type ('a, 'r) success = ('a → 'r State.t) with_state
type 'a parser = { run : 'r.
  ('r failure → ('a, 'r) success → 'r State.t) with_state
}
```

Angstrom parsers

```
let any_char =
  ensure 1 { run = fun input pos more _fail succ →
      succ input (pos + 1) more
        (Input.unsafe_get_char input pos)
  }

let (*>) a b =
  { run = fun input pos more fail succ →
    let succ' input' pos' more' _ =
      b.run input' pos' more' fail succ in
    a.run input pos more fail succ'
  }
```

Angstrom : effects branch

https://github.com/talex5/angstrom/tree/effects

```
type 'a parser = state → 'a

let any_char state =
  ensure 1 state;
  let c = Input.unsafe_get_char state.input state.pos in
  state.pos <- state.pos + 1;
  c

let (*>) a b state =
  let _ = a state in
  b state
```

Parser micro-benchmark

```
let parser = skip_many any_char
```

	Time	MinWrds	MajWrds
Callbacks	750.63ms	160.04Mw	8,9944.00kw
Effects	57.81ms	-	-

13 times faster!

Parser micro-benchmark

```
let parser = skip_many any_char
```

	Time	MinWrds	MajWrds
Callbacks	750.63ms	160.04Mw	8,9944.00kw
Callbacks'	180.73ms	220.01Mw	9,659.00w
Effects	57.81ms	-	-

3 times faster!

Realistic parser benchmark

Parsing an HTTP request shows smaller gains:

	Time	MinWrds	MajWrds
Callbacks	60.30ms	9.28Mw	102.08kw
Effects	50.71ms	2.13Mw	606.30w

Using effects for backwards compatibility

```
effect Read : int → state
let read c = perform (Read c)

let parse p =
  let buffering = Buffering.create () in
  try Unbuffered.parse ~read p
  with effect (Read committed) k →
    Buffering.shift buffering committed;
    Partial (fun input →
      Buffering.feed_input buffering input;
      continue k (Buffering.for_reading buffering)
    )
```

(simplified)

Angstrom summary

- Slightly faster
- Much simpler code
- No effects in interface
- Can convert between callbacks and effects easily

Eio : an IO library using effects for concurrency

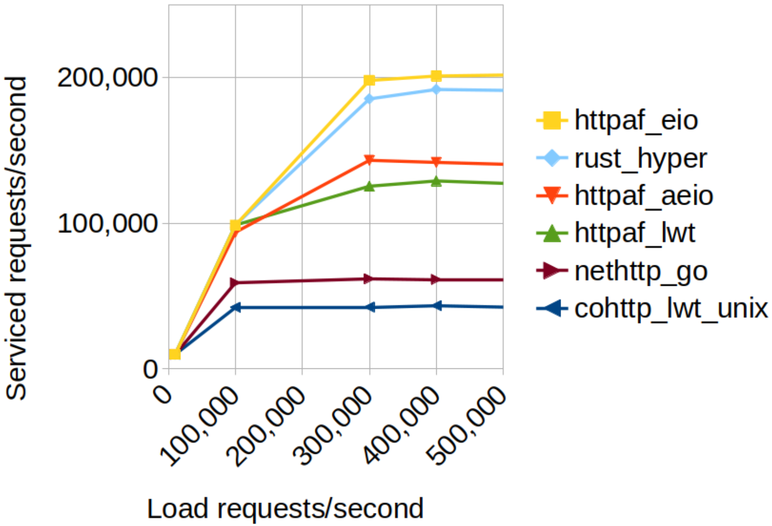
- Alternative to Lwt and Async
- Generic API that performs effects
- Cross-platform libuv effect handler
- High-performance io-uring handler for Linux

Eio example

```
let handle_connection =
  Httpaf_eio.Server.create_connection_handler
    ~config
    ~request_handler
    ~error_handler

let main ~net =
  Switch.top @@ fun sw →
  let socket = Eio.Net.listen ~sw net (`Tcp (host, port))
    ~reuse_addr:true
    ~backlog:1000
  in
  while true do
    Eio.Net.accept_sub ~sw socket handle_connection
      ~on_error:log_connection_error
  done
```

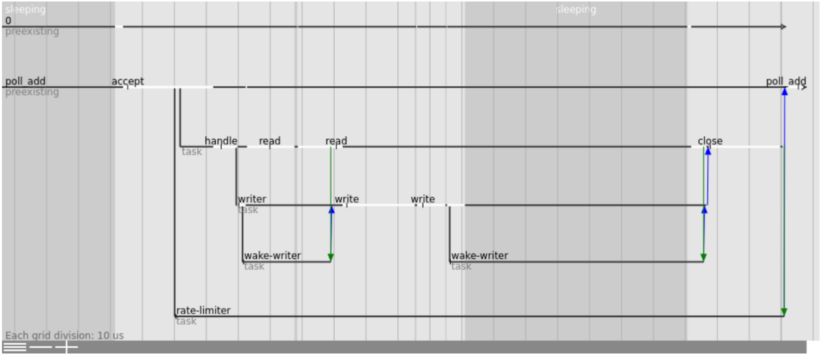
HTTP benchmark



100 concurrent connections. Servers limited to 1 core.

Eio : other features

- Structured concurrency
- OCaps security model
- Tracing support
- Supports multiple cores
- Still experimental



Summary

- Concurrency with effects works very well
- Effects have very good performance
- No bugs found in effects system during testing

https://github.com/ocaml-multicore/eio documentation shows how to try out OCaml effects.

Lwt example

```
let foo ~stdin total =
  let* n = Lwt_io.read_line stdin in
  Lwt_io.printlf "n/total = %d"
    (int_of_string n / total)
```

```
Fatal error: exception Division_by_zero
Raised at Lwt_example.foo in file "lwt_example.ml", line 6
Called from Lwt.[...].callback in file "src/core/lwt.ml", ...
```

- Backtrace doesn't say what called `foo`
- Closure with `total` allocated on the heap

Eio example

```
let foo ~stdin total =
  let n = read_line stdin in
  traceln "n/total = %d"
    (int_of_string n / total)
```

```
Fatal error: exception Division_by_zero
Raised at Eio_example.foo in file "eio_example.ml", line 11
Called from Eio_example.bar in file "eio_example.ml", line 15
...
```