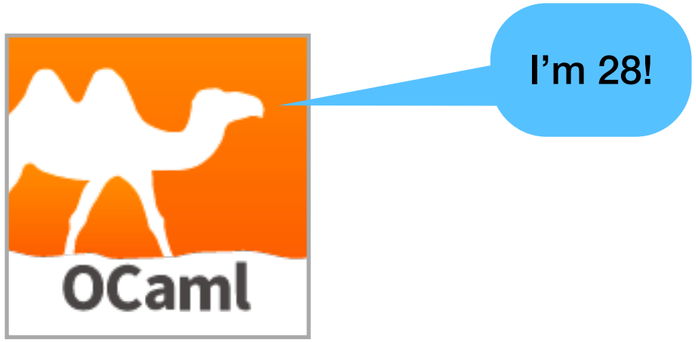
Concurrent Programming with Effect Handlers

“KC” Sivaramakrishnan

IIT MADRAS

Tarides



industrial-strength, pragmatic, functional programming language

Industry

FACEBOOK Tarides Bloomberg

docker ahrefs SimCorp

Microsoft Tezos Jane Street

Projects

hack flow COMPCERT

Higher-order functions

Hindley-Milner Type Inference

Powerful module system

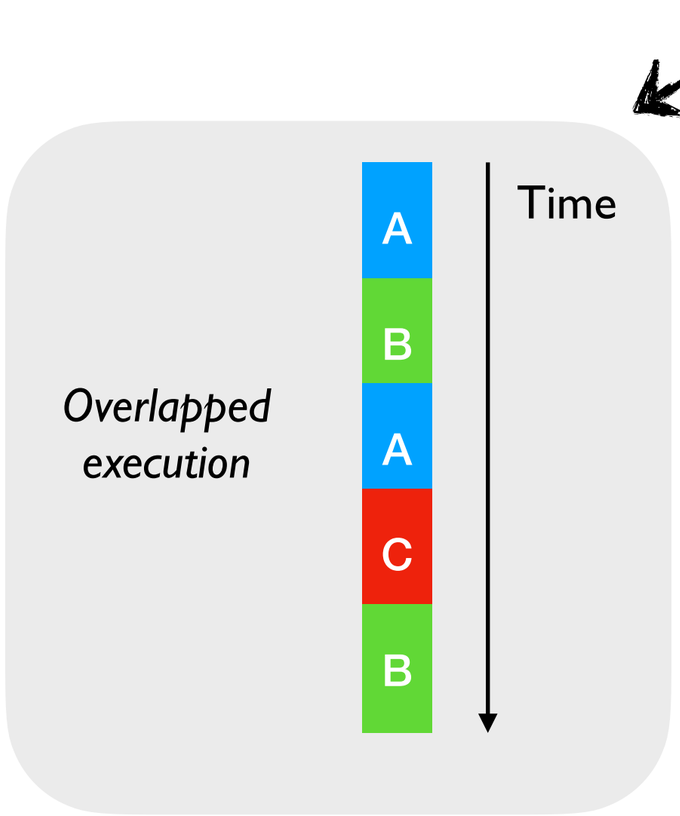
Functional core with imperative and object-oriented features

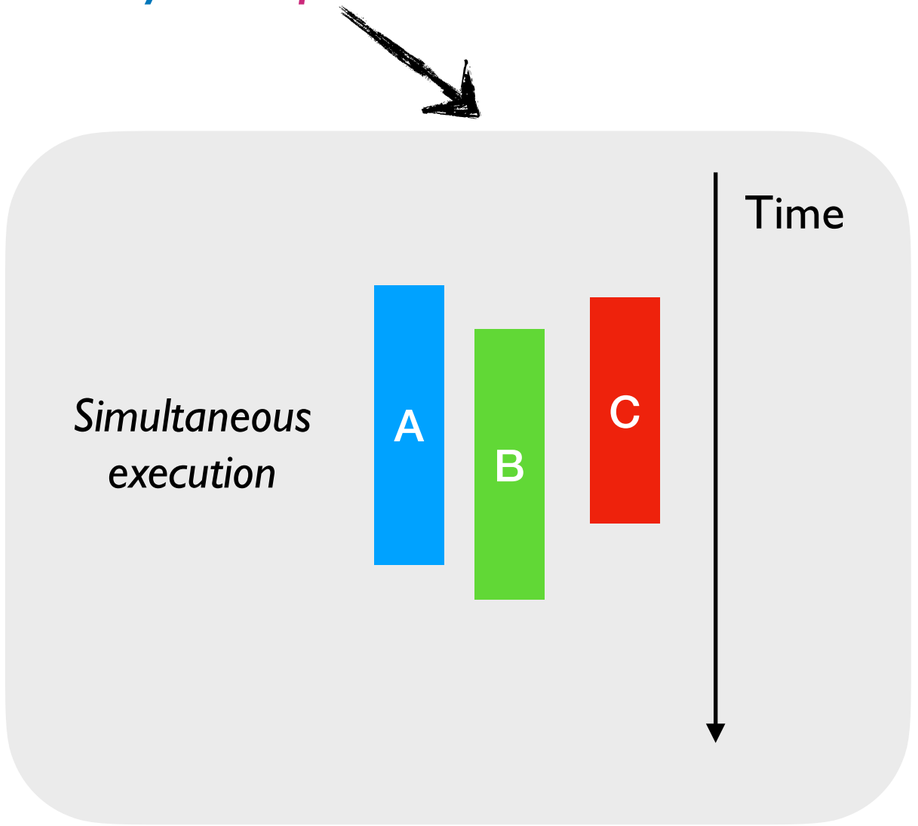
Native (x86, Arm, Power, RISC-V), JavaScript, WebAssembly

No multicore support!

Multicore OCaml ~> OCaml 5

- Brings native support for *concurrency* and *parallelism* to OCaml



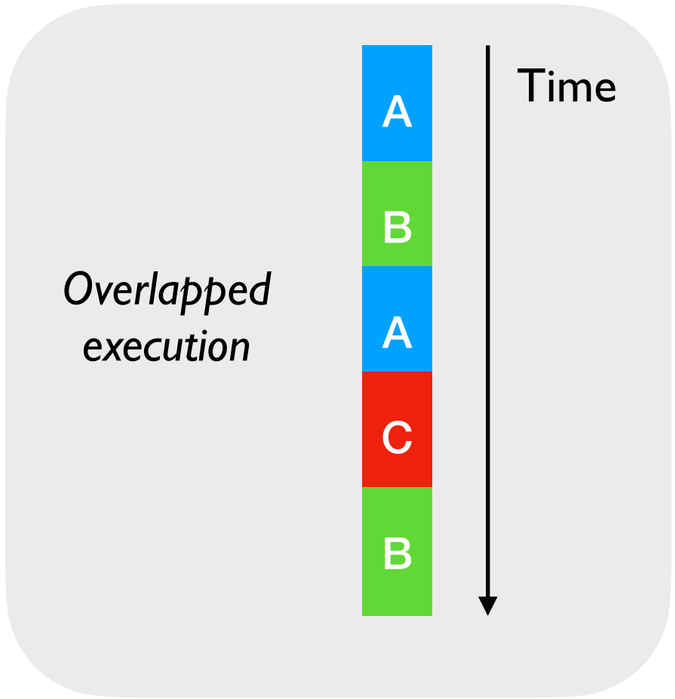


Effect Handlers

Domains

OCaml 5

- Adds native support for *concurrency* and *parallelism* to OCaml



Effect Handlers

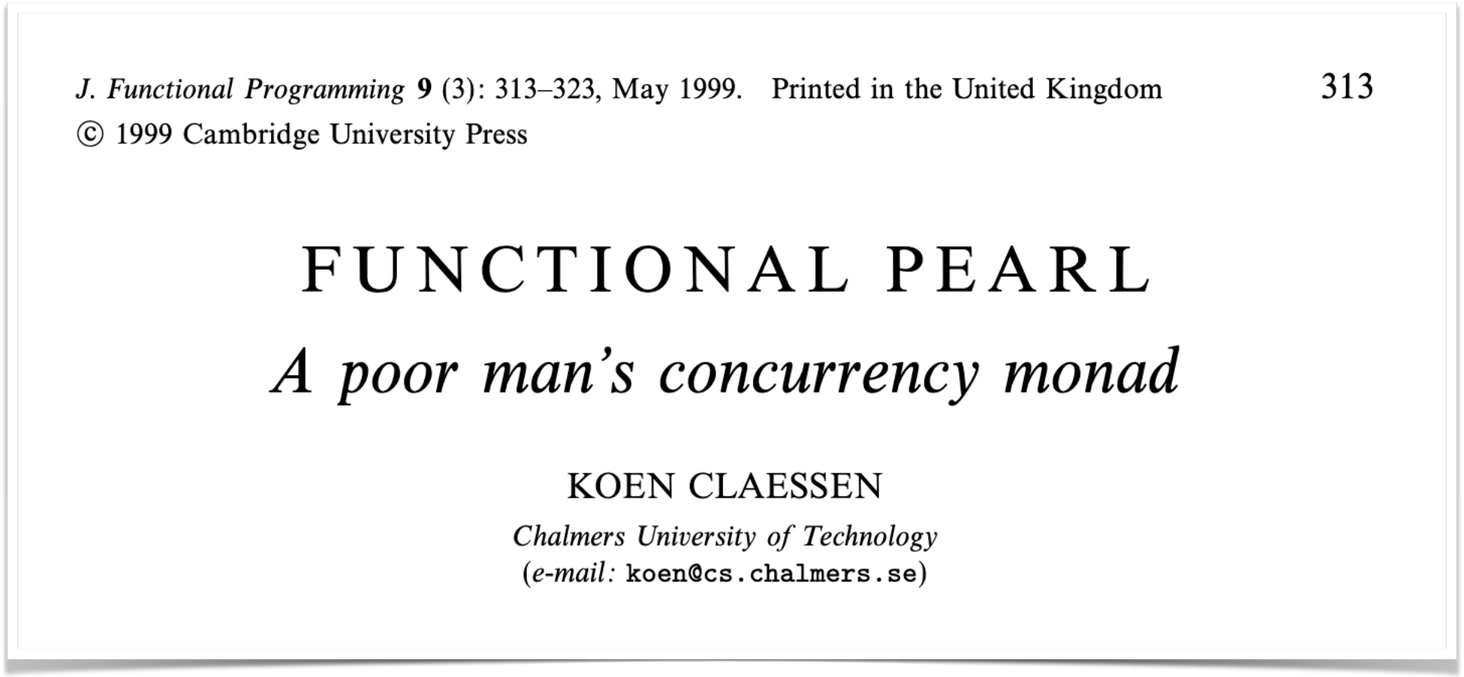
Domains

Concurrent Programming

- Computations may be *suspended* and *resumed* later
- Many languages provide concurrent programming mechanisms as *primitives*
  - **async/await** — JavaScript, Python, Rust, C# 5.0, F#, Swift, …
  - **generators** — Python, Javascript, …
  - **coroutines** — C++, Kotlin, Lua, …
  - **futures & promises** — JavaScript, Swift, …
  - **Lightweight threads/processes** — Haskell, Go, Erlang
- *Often include many different primitives in the same language!*
  - JavaScript has async/await, generators, promises, and callbacks

Conc. Prog. in OCaml 4

- No *primitive* support for concurrent programming in OCaml
  - ✦ **Lwt** and **Async** - concurrent programming *libraries*
  - ✦ Simulate concurrency through monads **>>=**



J. Functional Programming **9** (3): 313–323, May 1999. Printed in the United Kingdom

FUNCTIONAL PEARL

A poor man's concurrency monad

KOEN CLAESSEN
Chalmers University of Technology
(*e-mail:* koen@cs.chalmers.se)

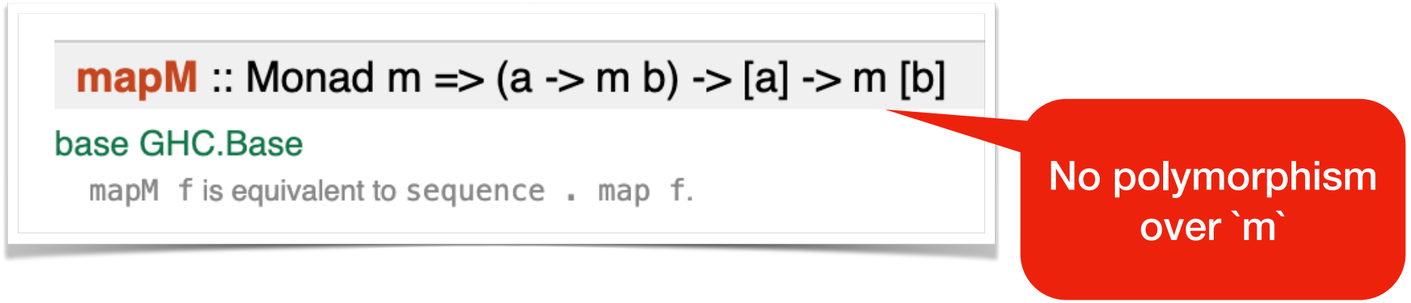
Conc. Prog. in OCaml 4

- Lwt and Async is *callback-oriented programming*

```
let main () =
  let url = "https://example.com" in
  let file_path = "titles.txt" in
  fetch_html url >>= fun html ->
  extract_titles html >>= fun titles ->
  save_titles_to_file titles file_path >>= fun () ->
  Lwt_io.printf "Titles saved to %s\n" file_path
```

Conc. Prog. in OCaml 4

- Lwt and Async suffer many *pitfalls* of programming with callbacks
  - ✦ No backtraces ⟹ debugging is harder
  - ✦ No built-in exceptions ⟹ OCaml language features cannot be used
  - ✦ More closures ⟹ more allocations ⟹ performance impact
- Monads are awkward to use in OCaml as there are no higher-kinded types



Function colour issue

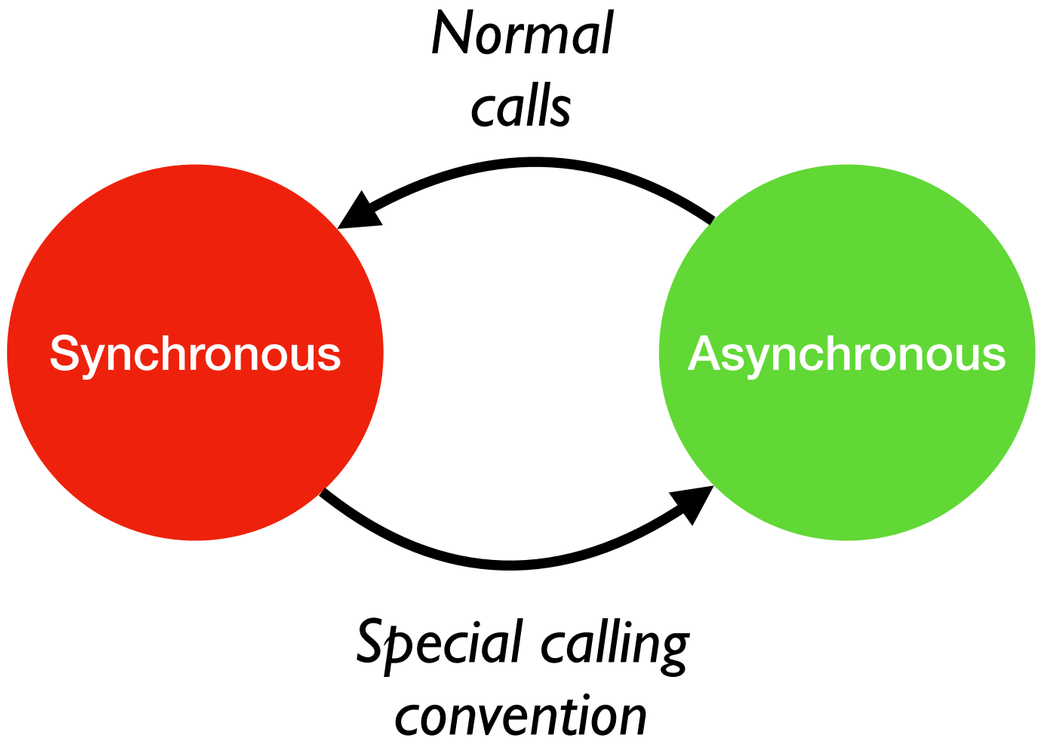
What Color is Your Function?

— Bob Nystrom

FEBRUARY 01, 2015 CODE DART GO JAVASCRIPT LANGUAGE LUA

I don't know about you, but nothing gets me going in the morning quite like a good old fashioned programming language rant. It stirs the blood to see someone skewer one of those "blub" languages the plebians use, muddling through their day with it between furtive visits to StackOverflow.

(Meanwhile, you and I, only use the most enlightened of languages. Chisel-sharp tools designed for the manicured hands of expert craftspersons such as ourselves.)



Native-concurrency Desiderata

- Avoid *function colouring* issue
- Avoid pitfalls of monadic concurrency / *callback-oriented programming*
- Keep the addition to the compiler *small*
  - OCaml is a volunteer-led effort

Add the *smallest* primitive that captures *many* concurrent programming patterns

Solution

Effect Handlers

- A mechanism for programming with *user-defined effects*
- *Modular* and *composable* basis of non-local control-flow me
  - ✦ Exceptions, generators, lightweight threads, promises, asynchron
- Effect handlers ~= *first-class, restartable exceptions*
  - ✦ Structured programming with *delimited continuations*

- Direct-style asynchronous I/O
- Generators
- Resumable parsers
- Probabilistic Programming
- Reactive UIs
- ....

https://github.com/ocaml-multicore/effects-examples

Exceptions in OCaml

```
exception Invalid_input of char

let get_choice () =
  print_endline "Do you want to continue [y/n]:";
  let c = input_char stdin in
  if c = 'y' then true
  else if c = 'n' then false
  else raise (Invalid_input c)
```

exception declaration

Raising the exception

Exceptions in OCaml

```
exception Invalid_input of char

let get_choice () =
  print_endline "Do you want to continue [y/n]:";
  let c = input_char stdin in
  if c = 'y' then true
  else if c = 'n' then false
  else raise (Invalid_input c)

let rec really_get_choice () =
  try get_choice ()
  with Invalid_input c ->
    print_endline "Invalid input. Please enter y or n.";
    really_get_choice ()
```

Exception handler

Demo

Exceptions in OCaml

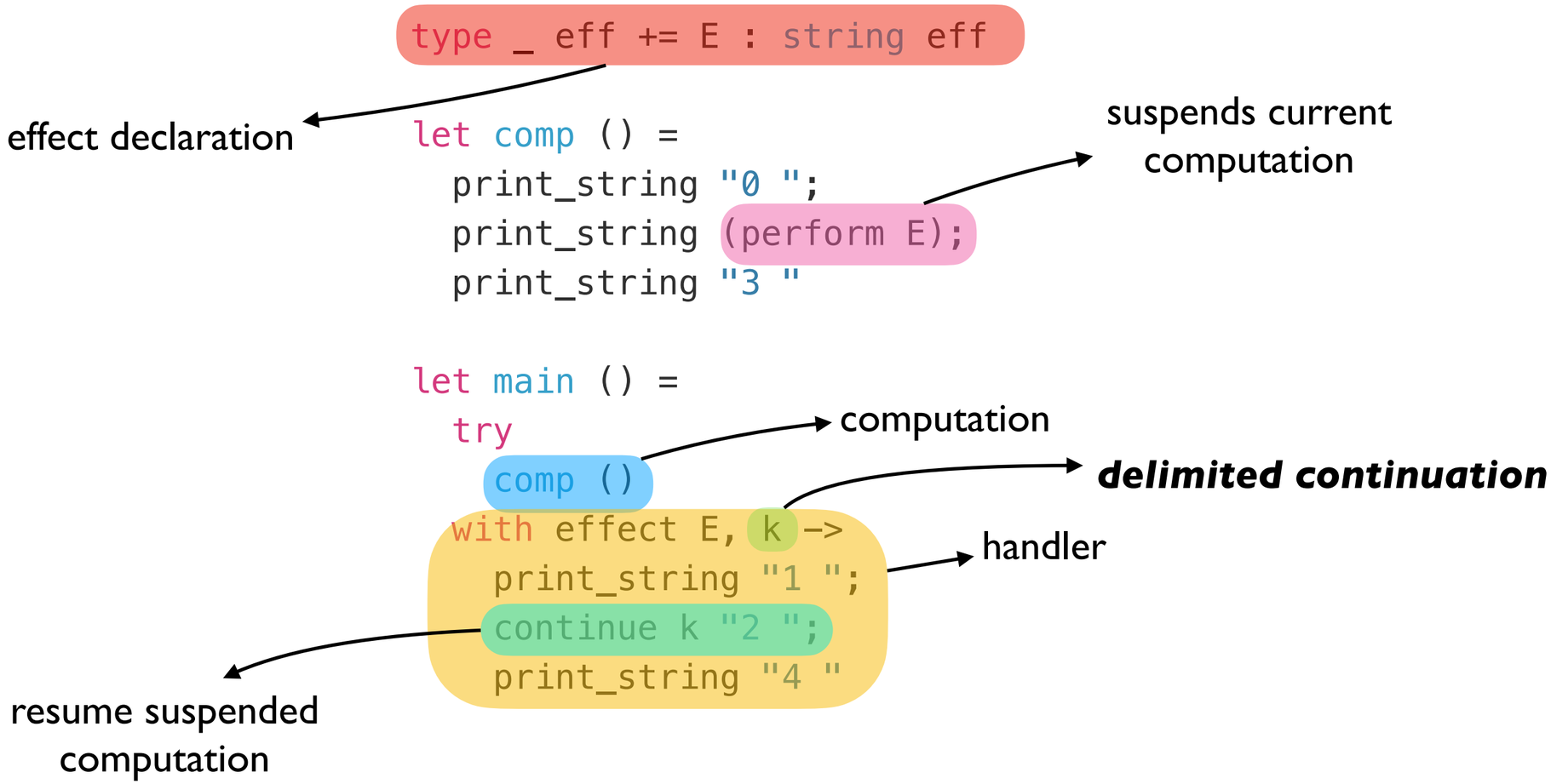
```
exception Invalid_input of char
```

is the same as

```
type exn += Invalid_input of char
```

`exn` is a built-in ***extensible variant type***

Effect handlers

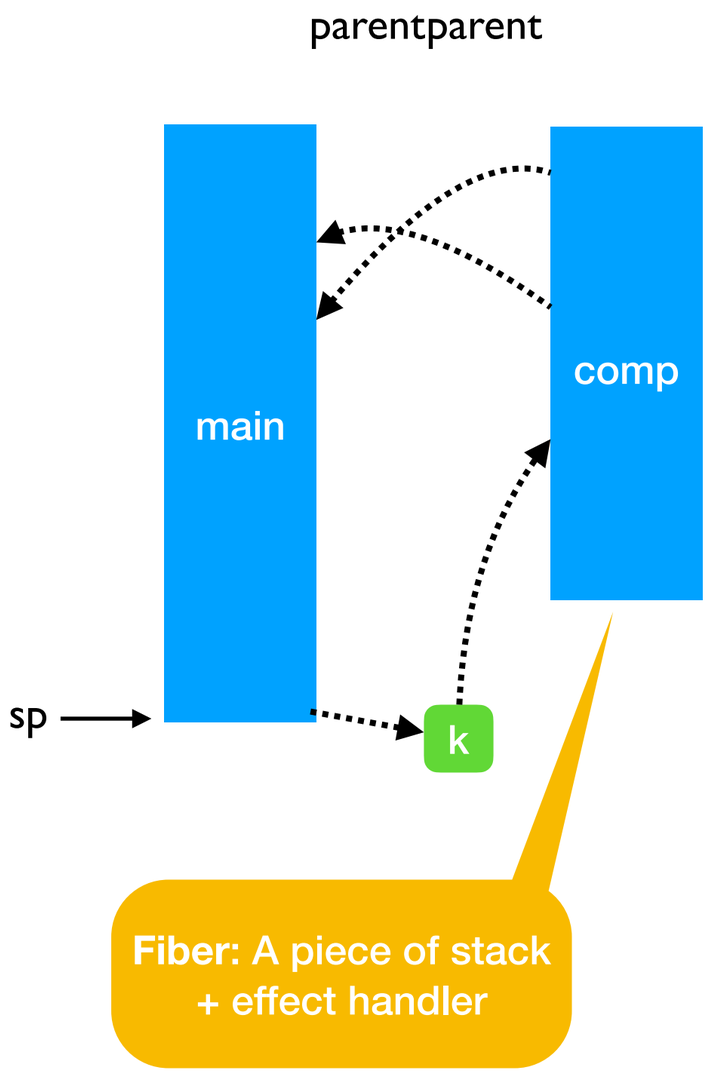


Stepping through the example

```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →

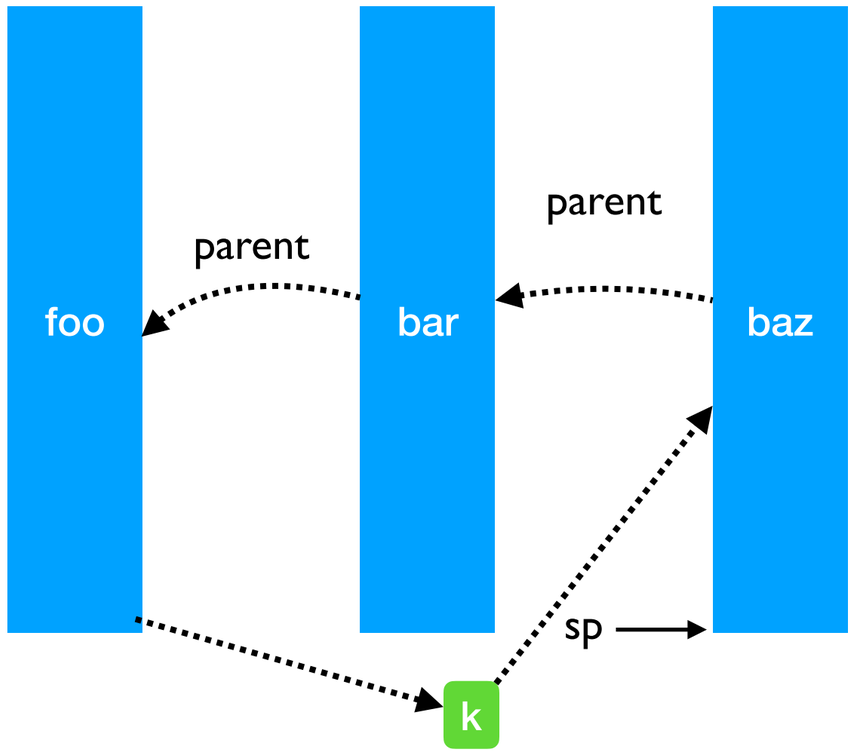
0 1 2 3 4



Handlers can be nested

```
type _ eff += A : unit eff
           | B : unit eff

let baz () =
  perform A

let bar () =
  try
    baz ()
  with effect B, k ->
    continue k ()

let foo () =
  try
    bar ()
  with effect A, k ->
    continue k ()
```

pc →



- Linear search through handlers
  - ✦ *Handler stacks shallow in practice*

Lightweight Threading

```
type _ eff += Fork  : (unit -> unit) -> unit eff
            | Yield : unit eff

let run main =
  ... (* assume queue of continuations *)
  let run_next () =
    match dequeue () with
    | Some k -> continue k ()
    | None -> ()
  in
  let rec spawn f =
    match f () with
    | () -> run_next () (* value case *)
    | effect Yield, k -> enqueue k; run_next ()
    | effect (Fork f), k -> enqueue k; spawn f
  in
  spawn main

let fork f = perform (Fork f)
let yield () = perform Yield
```

Lightweight threading

```
let main () =
  fork (fun _ ->
    print_endline "1.a";
    yield ();
    print_endline "1.b");
  fork (fun _ ->
    print_endline "2.a";
    yield ();
    print_endline “2.b")
;;
run main
```

```
1.a
2.a
1.b
2.b
```

Demo

Lightweight threading

```
let main () =
  fork (fun _ ->
    print_endline "1.a";
    yield ();
    print_endline "1.b");
  fork (fun _ ->
    print_endline "2.a";
    yield ();
    print_endline “2.b")
;;
run main
```

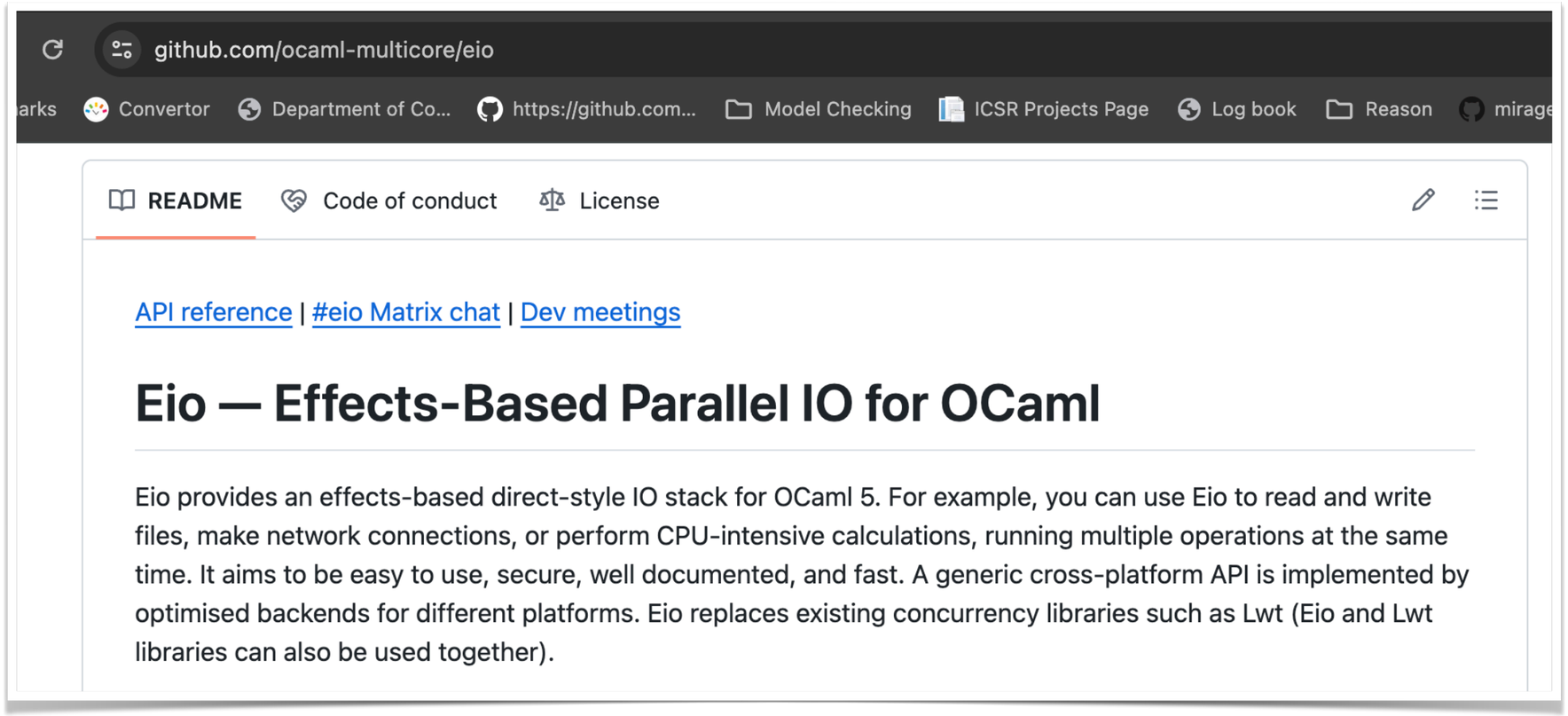
```
1.a
2.a
1.b
2.b
```

Ability to specialise scheduler unlike GHC Haskell / Go

- Direct-style (no monads)
- User-code need not be aware of effects
- No Async vs Sync distinction

Performance: WebServer

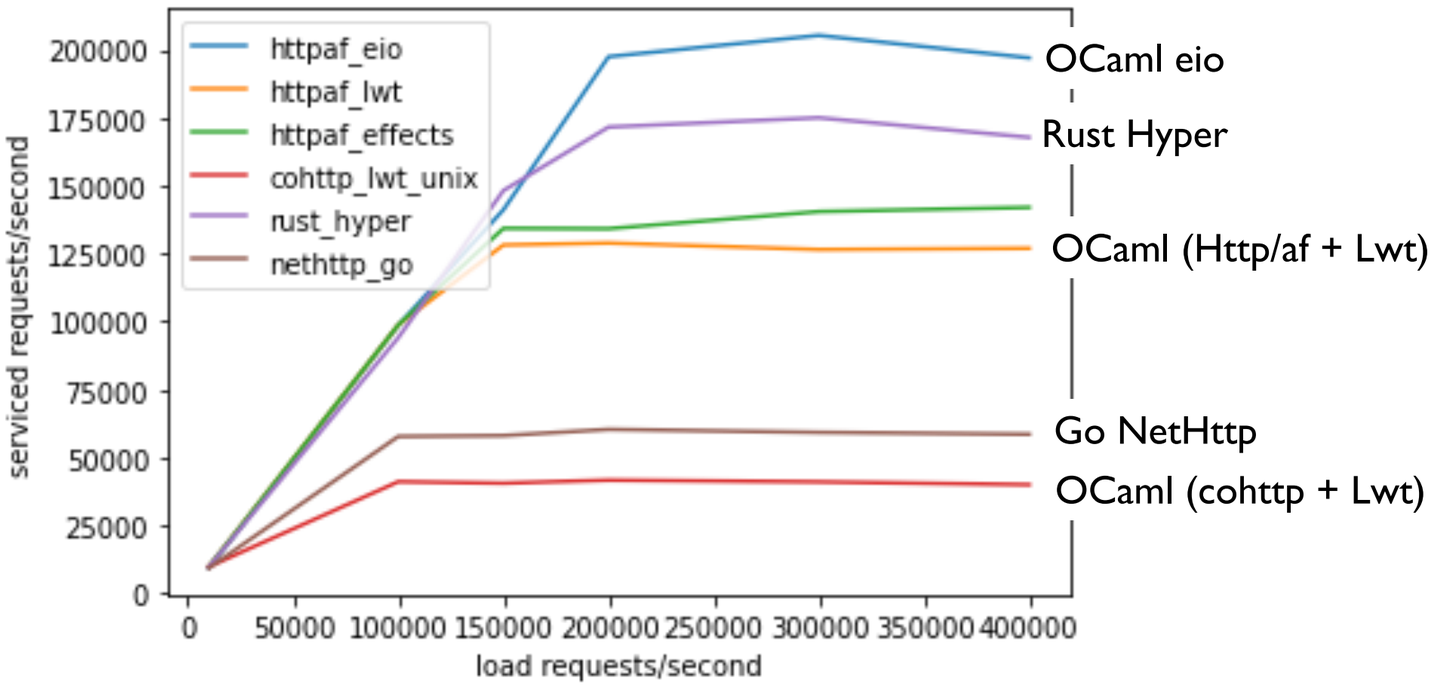
- **eio**: effects-based direct-style I/O
  - ✦ Multiple backends — epoll, select, io_uring



https://github.com/ocaml-multicore/eio

Performance: WebServer

- **eio**: effects-based direct-style I/O
  - ✦ Multiple backends — epoll, select, io_uring



100 open connections, 60 seconds w/ io_uring

https://github.com/ocaml-multicore/eio

Generators

- Generators — non-continuous traversal of data structure by yielding values
  - ✦ Primitives in JavaScript and Python

```
function* generator(i) {
  yield i;
  yield i + 10;
}
const gen = generator(10);

console.log(gen.next().value);
// expected output: 10

console.log(gen.next().value);
// expected output: 20
```

- Can be *derived automatically* from any iterator using effect handlers

Generators: effect handlers

```
module MkGen (S :sig
  type 'a t
  val iter : ('a -> unit) -> 'a t -> unit
end) : sig
  val gen : 'a S.t -> (unit -> 'a option)
end = struct
  let gen : type a. a S.t -> (unit -> a option) = fun l ->
    let module M =
      struct type _ eff += Yield : a -> unit eff end
    in
    let open M in
    let rec step = ref (fun () ->
      match S.iter (fun v -> perform (Yield v)) l with
      | () -> None
      | effect (Yield v), k ->
          step := (fun () -> continue k ());
          Some v)
    in
    fun () -> !step ()
end
```

Generators: List

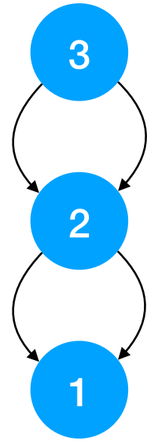
```
module L = MkGen (struct
  type 'a t = 'a list
  let iter = List.iter
end)
```

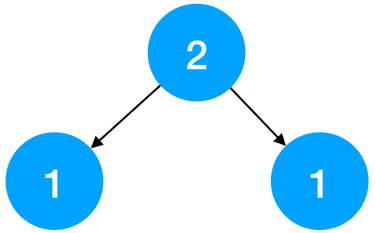
```
let next = L.gen [1;2;3]
next() (* Some 1 *)
next() (* Some 2 *)
next() (* Some 3 *)
next() (* None *)
```

Generators: Tree

```
type 'a tree =
| Leaf
| Node of 'a tree * 'a * 'a tree

let rec iter f = function
  | Leaf -> ()
  | Node (l, x, r) ->
      iter f l; f x; iter f r

module T = MkGen(struct
  type 'a t = 'a tree
  let iter = iter
end)
```

```
(* Make a complete binary tree of
   depth [n] using [O(n)] space
*)
let rec make = function
  | 0 -> Leaf
  | n -> let t = make (n-1)
         in Node (t,n,t)

 let t = make 2

 let next = T.gen t
 next() (* Some 1 *)
 next() (* Some 2 *)
 next() (* Some 1 *)
 next() (* None *)
```





Performance: Generators

- Traverse a complete binary-tree of depth 25
  - ✦ 2^{26} stack switches
- *Iterator* — idiomatic recursive traversal
- Generator
  - ✦ Hand-written generator (*hw-generator*)
    - ✤ Specialised for in-order traversal of binary trees
    - ✤ CPS translation + defunctionalization to remove intermediate closure allocation
  - ✦ Generator using effect handlers (*eh-generator*)

Performance: Generators

OCaml 5

Variant	Time (milliseconds)
Iterator (baseline)	202
hw-generator	837 (**3.76x**)
eh-generator	1879 (**9.30x**)

nodejs 14.07

Variant	Time (milliseconds)
Iterator (baseline)	492
generator	43842 (**89.1x**)

Retrofitting Challenges

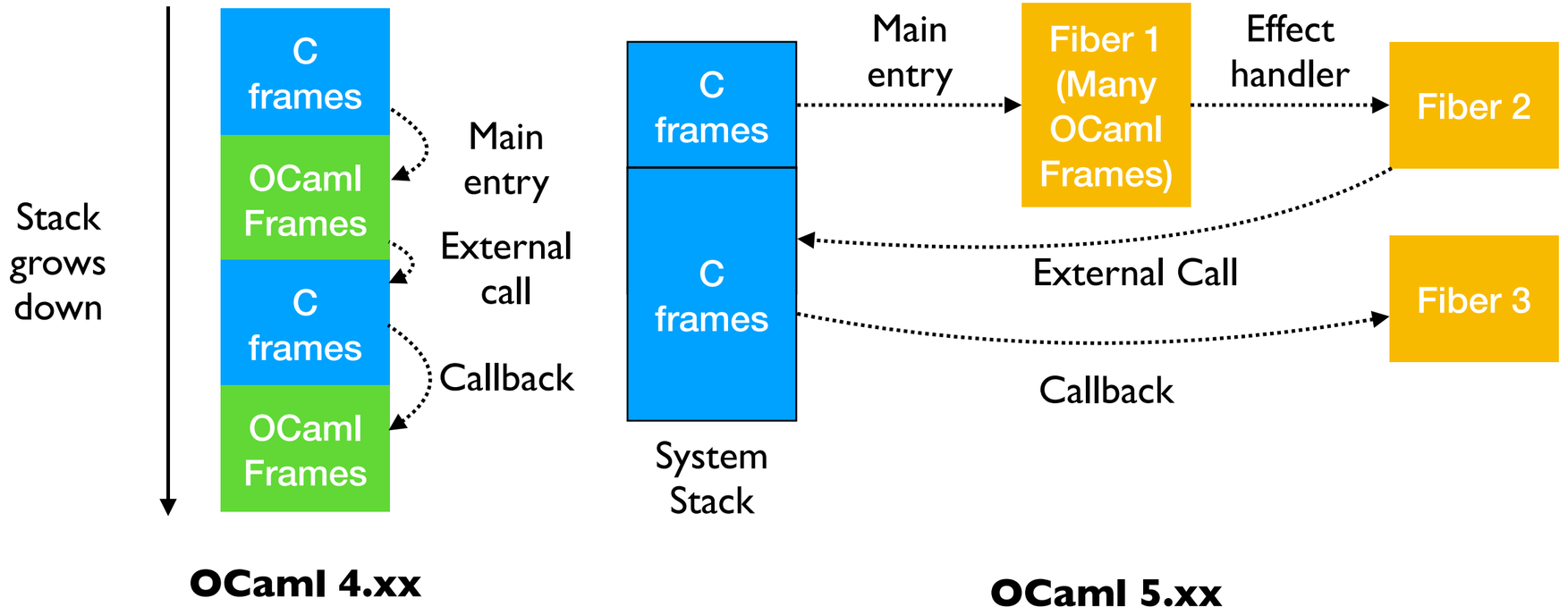
Retrofitting Challenges

- Millions of lines of legacy code
  - ✦ Written without *non-local control-flow* in mind
  - ✦ Cost of refactoring sequential code itself is *prohibitive*
- OCaml uses the *same system stack* for both OCaml and C
  - ✦ *Fast exceptions and FFI* between C and OCaml
  - ✦ *No stack overflow* checks needed
  - ✦ Excellent compatibility with debugging (gdb) and profiling (perf) tools

Must preserve
feature, tooling, performance
compatibility

Representing Stacks & Continuations

- A stack of runtime-managed, *dynamically growing* stack segments
  - ✦ No pointers into OCaml stack
  - ✦ *Need stack overflow checks* for OCaml code
- Switch to system stack for C calls



Representing Stacks & Continuations

- A s
  - ✦ N
  - ✦ N
- Swi

Switching stacks fast

- *One-shot* — capture and resumption does not involve copying frames
- No callee-saved registers in OCaml
  - ✦ Switching between stacks need not save & restore register state

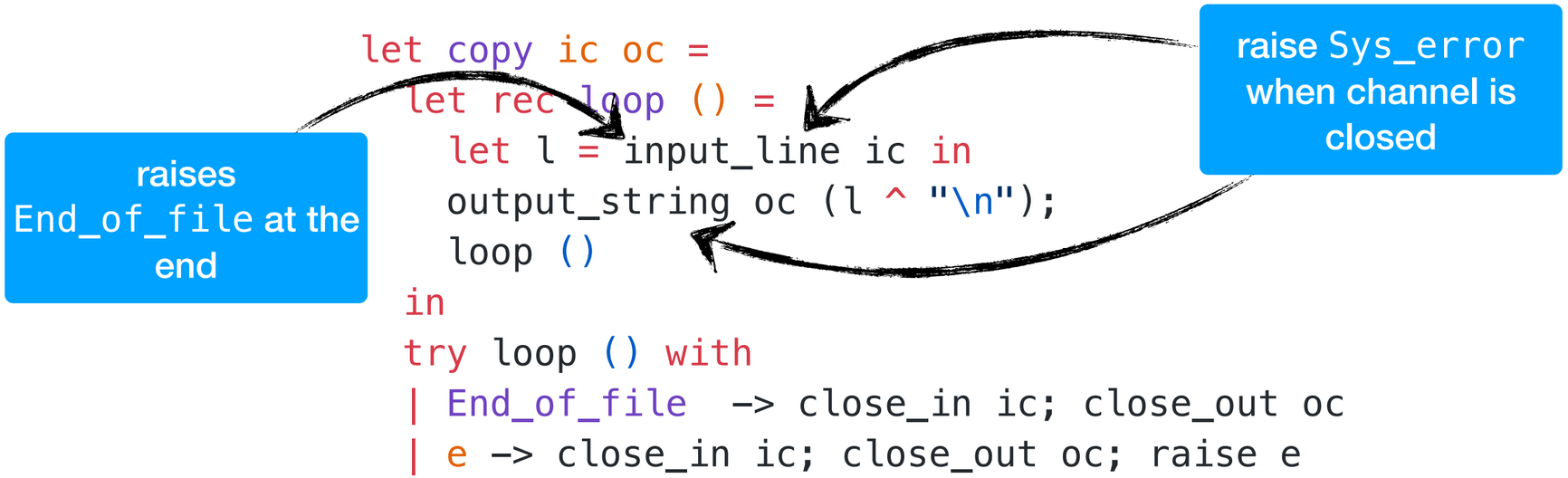
```
let foo () =
   (* a *)
  try
    (* b *)
    perform E
    (* d *)
  with effect E, k ->
    (* c *)
    continue k ()
    (* e *)
```

Instruction Sequence	Significance	Time (ns)
a to b	Create a new stack & run the computation	23
b to c	Performing & handling an effect	5
c to d	Resuming a continuation	11
d to e	Returning from a computation & free the stack	7

- Each of the instruction sequences involves a stack switch
- Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
  - ★ For reference, memory read latency is **90 ns** (local NUMA node) and **145 ns** (remote NUMA node)

Backwards Compatibility

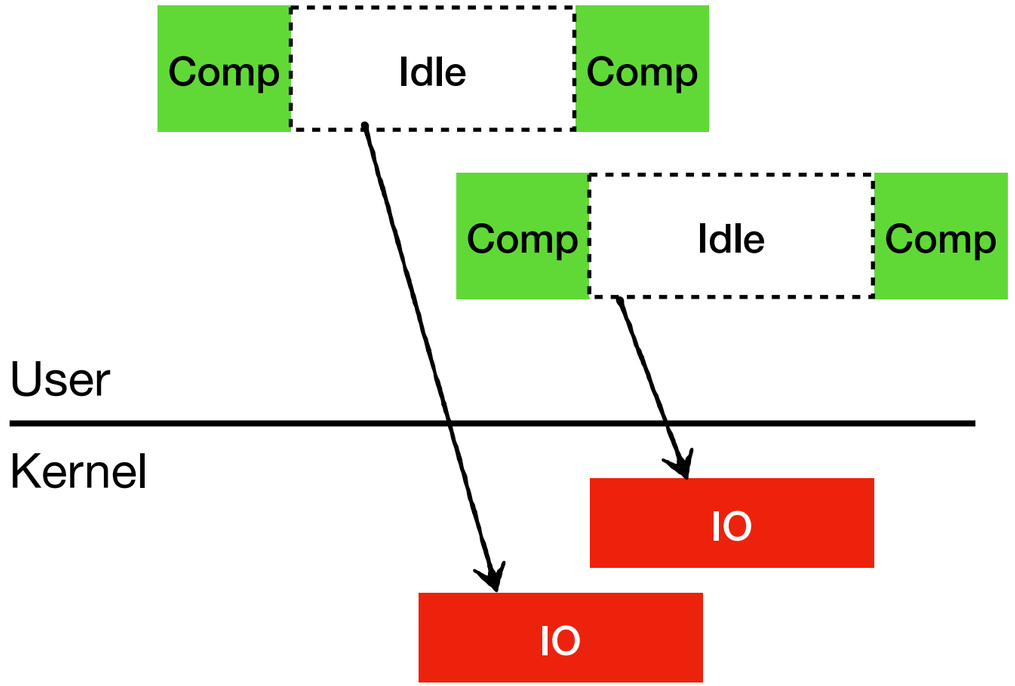
- OCaml is a systems programming language
  - ✦ Manipulates resources such as files, sockets, buffers, etc.
- OCaml code is written in *defensive style* to guard against exceptional behaviour and clear up resources



We would like to make this code transparently asynchronous

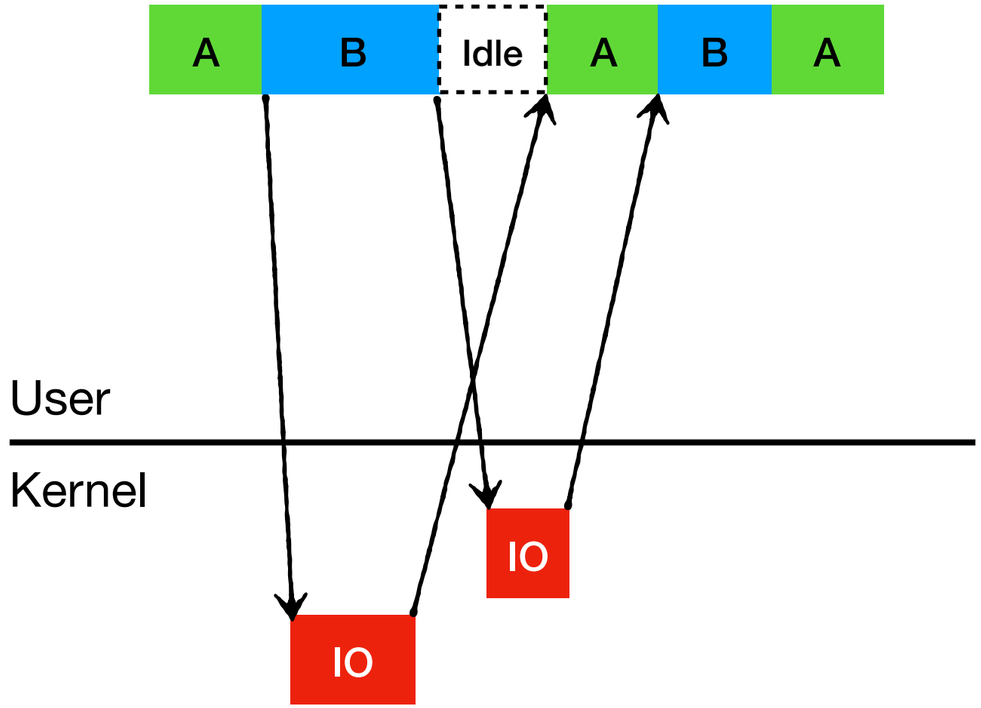
Why Asynchronous IO?

- Computation is *fast*; IO is *slow*
- Modern OSes can *batch and parallelise IO*
  - ✦ Select, Epoll, kqueue, io_uring, IOCP, GCD
- Creating multiple OS threads for IO parallelism is *expensive*



Why Asynchronous IO?

- Computation is *fast*; IO is *slow*
- Modern OSes can *batch and parallelise IO*
  - ✦ Select, Epoll, kqueue, io_uring, IOCP, GCD
- Creating multiple OS threads for IO parallelism is *expensive*
- Multiplex computations on the same OS thread and parallelise IO
  - ✦ *Suspend* and *resume* at IO operations



Asynchronous IO

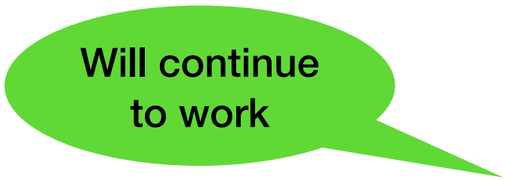
```
type _ eff += In_line : in_channel -> string eff
            | Out_str : out_channel * string -> unit eff

let input_line ic = perform (In_line ic)
let output_string oc s = perform (Out_str (oc,s))

let run_aio f = match f () with
| v -> v
| effect ... -> ...
| effect (In_line chan), k ->
    register_async_input_line chan k;
    run_next ()
| effect (Out_str (chan, s)), k ->
    register_async_output_string chan s k;
    run_next ()
```

- Continue with appropriate *value* when the asynchronous IO call returns
- But what about termination? — `End_of_file` and `Sys_error` *exceptional* cases.

Discontinue

```
discontinue k End_of_file
```

- We add a discontinue primitive to resume a continuation by raising an exception
- On `End_of_file` and `Sys_error`, the asynchronous IO scheduler uses discontinue to raise the appropriate exception




```
let copy ic oc =
  let rec loop () =
    let l = input_line ic in
    output_string oc (l ^ "\n");
    loop ()
  in
  try loop () with
  | End_of_file  -> close_in ic; close_out oc
  | e -> close_in ic; close_out oc; raise e
```

Linearity

- Resources such as sockets, file descriptors, channels and buffers are *linear* resources
  - ✦ Created and destroyed *exactly once*
- OCaml functions return *exactly once* with *value* or *exception*
  - ✦ Defensive programming already guards against exceptional return cases
- With effect handlers, functions may return *at-most-once* if continuation is not resumed
  - ✦ This breaks resource-safe legacy code

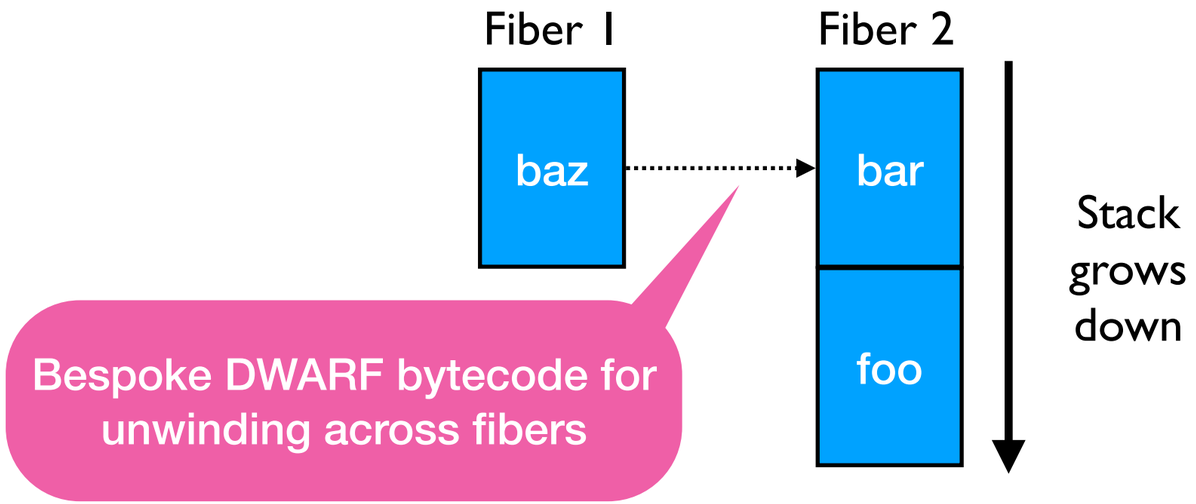
Linearity

```
type _ eff += E : unit eff
let foo () = perform E

let bar () =
  let ic = open_in "input.txt" in
  match foo () with
  | v -> close_in ic
  | exception e -> close_in ic; raise e

let baz () =
  try bar () with
  | effect E, _ -> () (* leaks ic *)
```

We *assume* that captured continuations are resumed *exactly once,* either using `continue` or `discontinue`

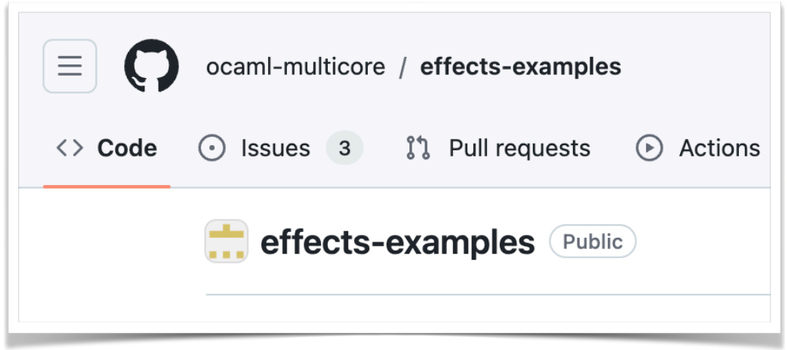
Backtraces

- OCaml has excellent compatibility with off-the-shelf debugging and profiling tools
  - ✦ GDB, LLDB, perf, libunwind, etc.
- DWARF — debugging information format
  - ✦ Unwind the program stack to get a backtrace, find the source variables
  - ✦ Bespoke bytecode format (Turing complete!), included in executables, interpreted at runtime

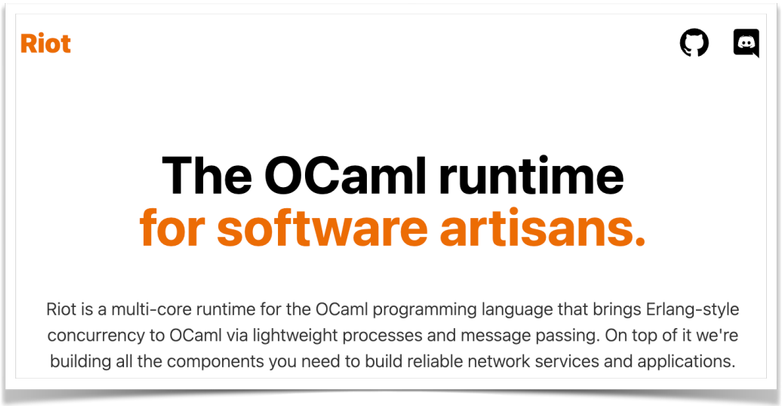


Demo

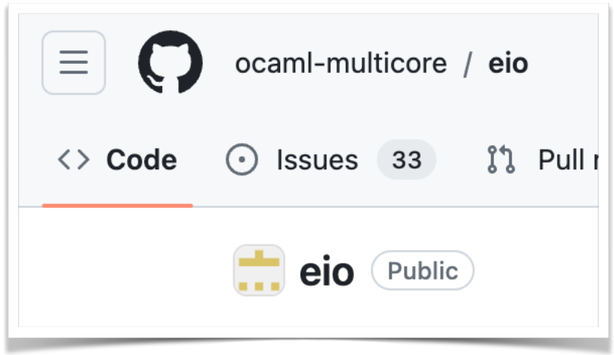
Join in the fun!



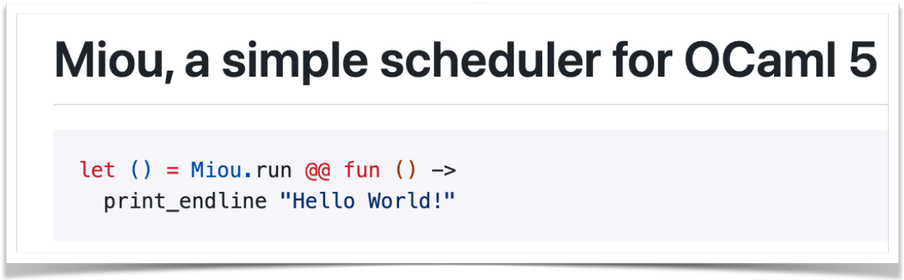
All examples from talk and more



Riot Stack

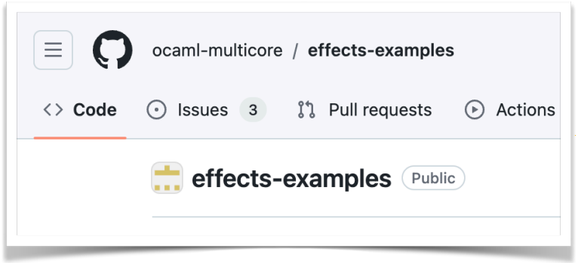


Eio

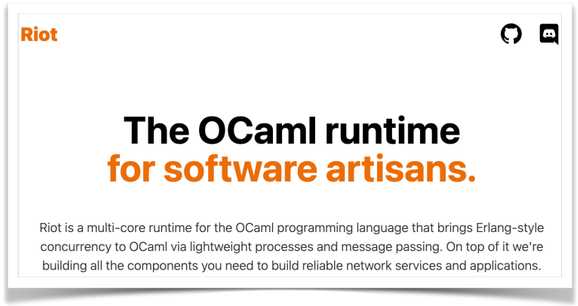


Miou

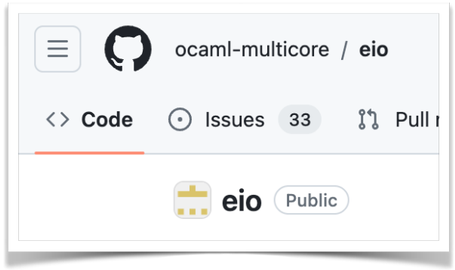
Join in the fun!



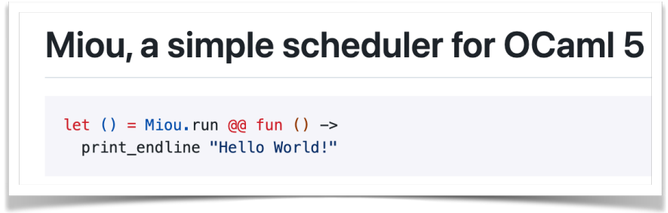
All examples from talk and more



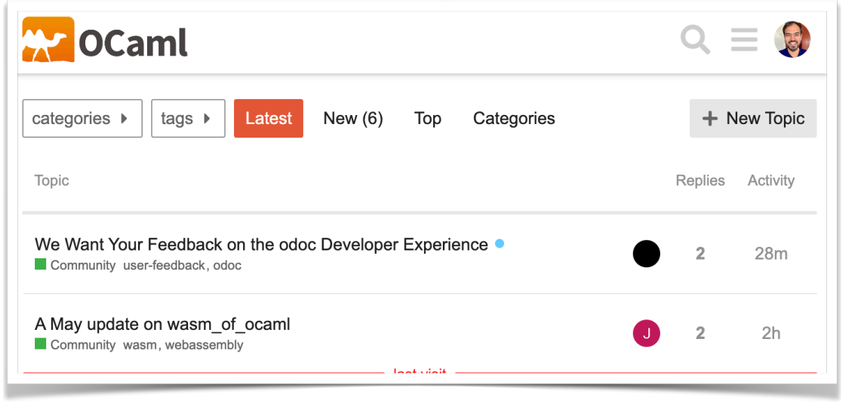
Riot Stack



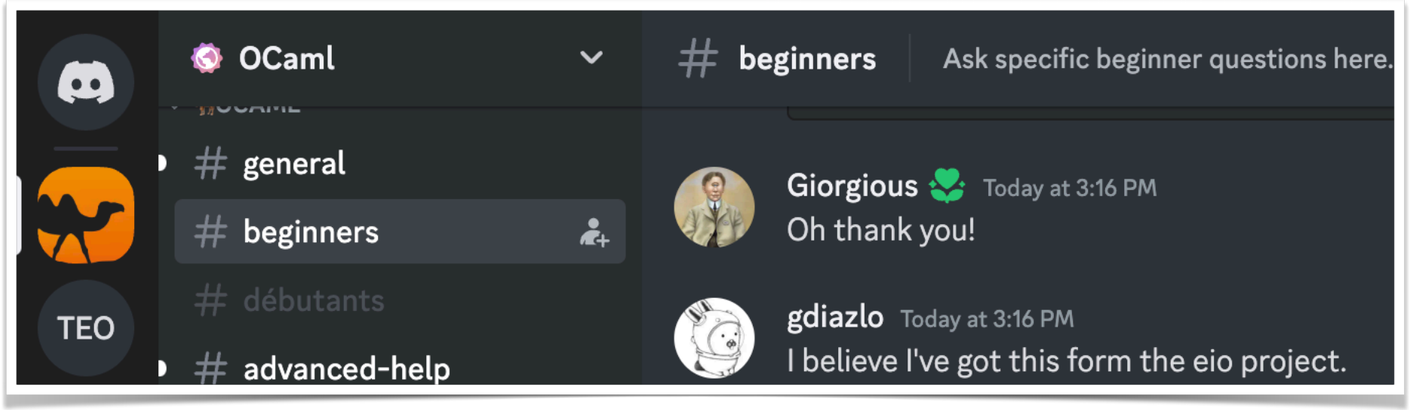
Eio



Miou



discuss.ocaml.org



OCaml Discord