Effectively Composing Concurrency Libraries

"KC" Sivaramakrishnan

Joint work with Deepali Ande & Sudha Parimala

IIT MADRAS

Tarides

OCaml 5.0 is out

- First industrial-strength language to support effect handlers!
- Effects in OCaml 5 are *unchecked*
  - ✦ Structured programming with *one-shot* delimited continuations
- Implemented with *runtime-managed, dynamically-growing stack segments*
- Deep and Sheep handlers are supported as *library* functions

5.0

Concurrent Programming

- Primary motivation is *direct-style concurrency as a library*
- *Direct-style concurrency*
  - As opposed to monadic concurrency — Lwt and Async
  - Pros — fewer closures, backtraces, exceptions, no function colours
- *As a library*
  - As opposed to primitive concurrency — GHC Haskell and Go
  - Pros — Specialising schedulers for problems, smaller compiler

Many libraries!

- IO — round-robin scheduling, work-sharing
    - Eio — asynchronous & parallel IO, structured concurrency, multiple backends (io_uring, epoll, *iocp*, *GCD*)
        - Heading towards 1.0 around ICFP
    - Oslo — parallel IO
    - Miou — parallel IO
    - Affect — "composable" concurrent IO
- Parallelism
    - Domainslib — Nested parallel programming, work-stealing
    - Moonpool — Parallelism over thread pools

Great, but…

Monolithic libraries

Each library ends up being
a non-composable monolith

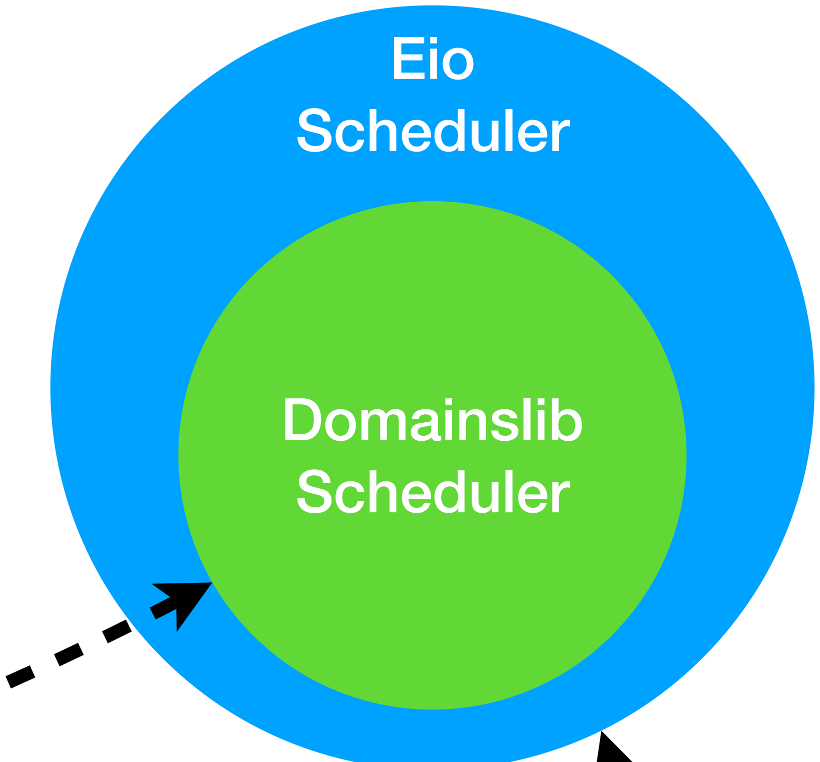
- Each library implements its own incompatible notion *tasks*
  - ✦ *Tasks* = User-level lightweight threads
  - ✦ Domains = A unit of parallel execution (~= system/OS thread)
- Crux of the problem
  - ✦ Each library has its own notion of *blocking* and *unblocking* tasks

Why compose?

- High-performance job processor app
  - ✦ Requests from remote clients, parallelised over multiple domains, results sent back
- Recursive fibonacci compute server
  - ✦ Compute `fib(n)` where `n` is from the client
- Libraries
  - ✦ Eio — high-performance, safe networking
  - ✦ Domainslib — nested parallel programming
- *Compose these two to build the app?*

Recursive Fib server

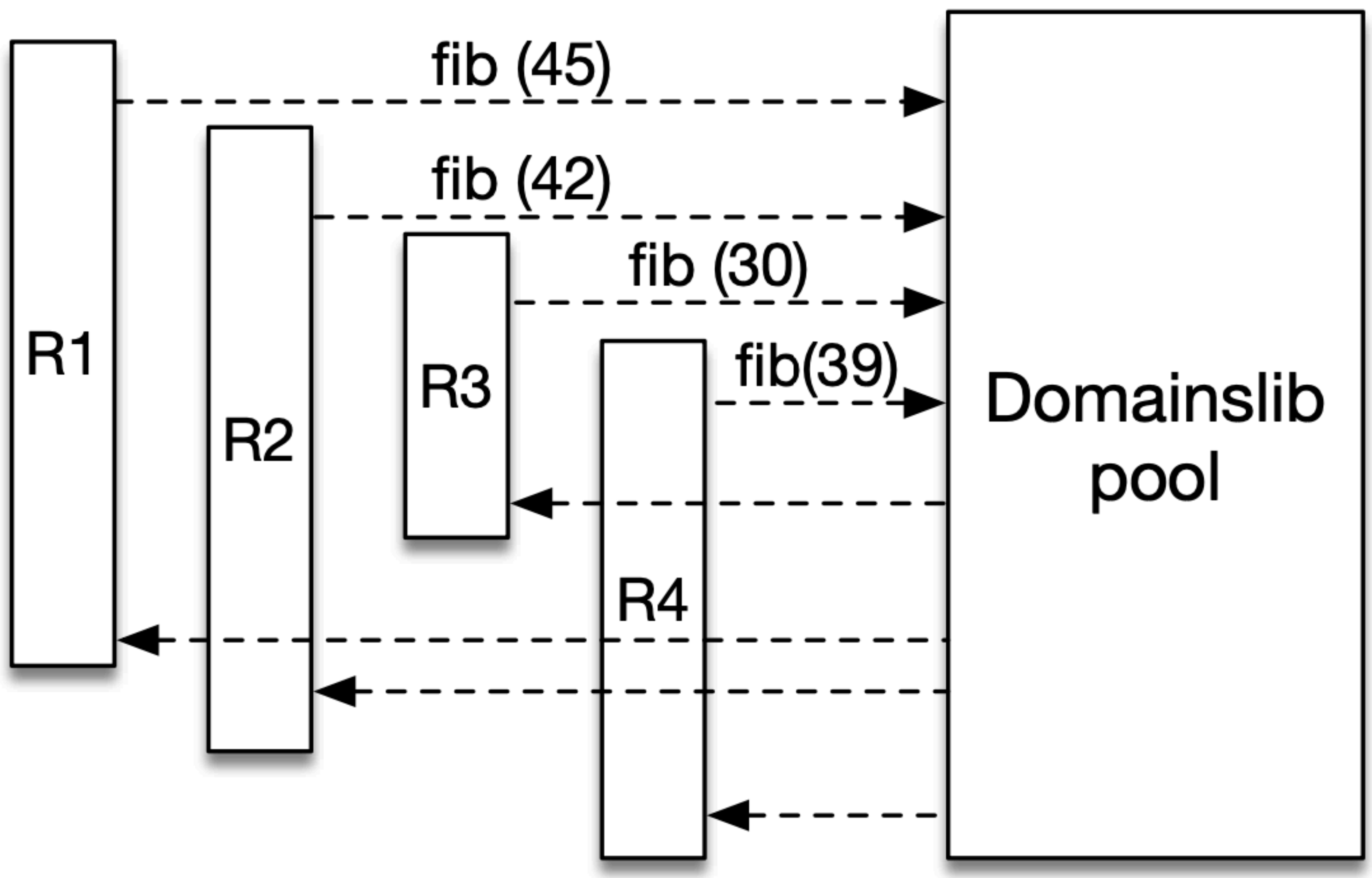
```
module T = Domainslib.Task
(* set up a pool of [num_domains] domains for
   parallel computation *)
let pool = T.setup_pool ~num_domains ()

let main () =
  let sock = Eio.Net.listen ... in
  (* Runs once per request in an Eio task *)
  let request_handler n =
   T.run pool (fun _ -> fib_par n)
  in
  while true do
    (* spawn an Eio task to run [request_handler] per request *)
    Eio.Net.accept_fork sock ... request_handler ...
  done

let () = Eio_main.run main
```



Recursive Fib server

```
module T = Domainslib.Task
(* set up a pool of [num_domains] domains for
   parallel computation *)
let pool = T.setup_pool ~num_domains ()

(* Parallel Fibonacci computation *)
let rec fib_par n =
  let rec fib n =
    if n < 2 then 1
    else fib (n - 1) + fib (n - 2)
  in
  if n > 20 then begin
    let a = T.async pool (fun _ -> fib_par (n-1)) in
    let b = T.async pool (fun _ -> fib_par (n-2)) in
    T.await pool a + T.await pool b
  end else
    fib n
```

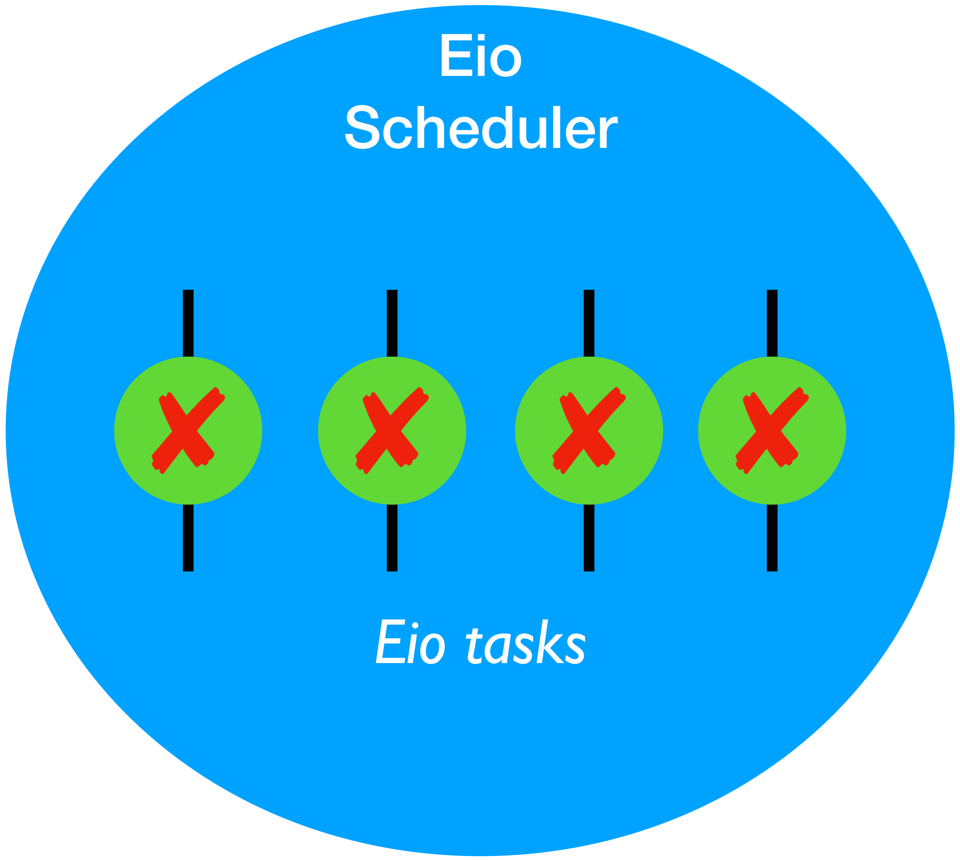
Intended Behaviour



Concurrent client requests are pipelined to the domainslib pool

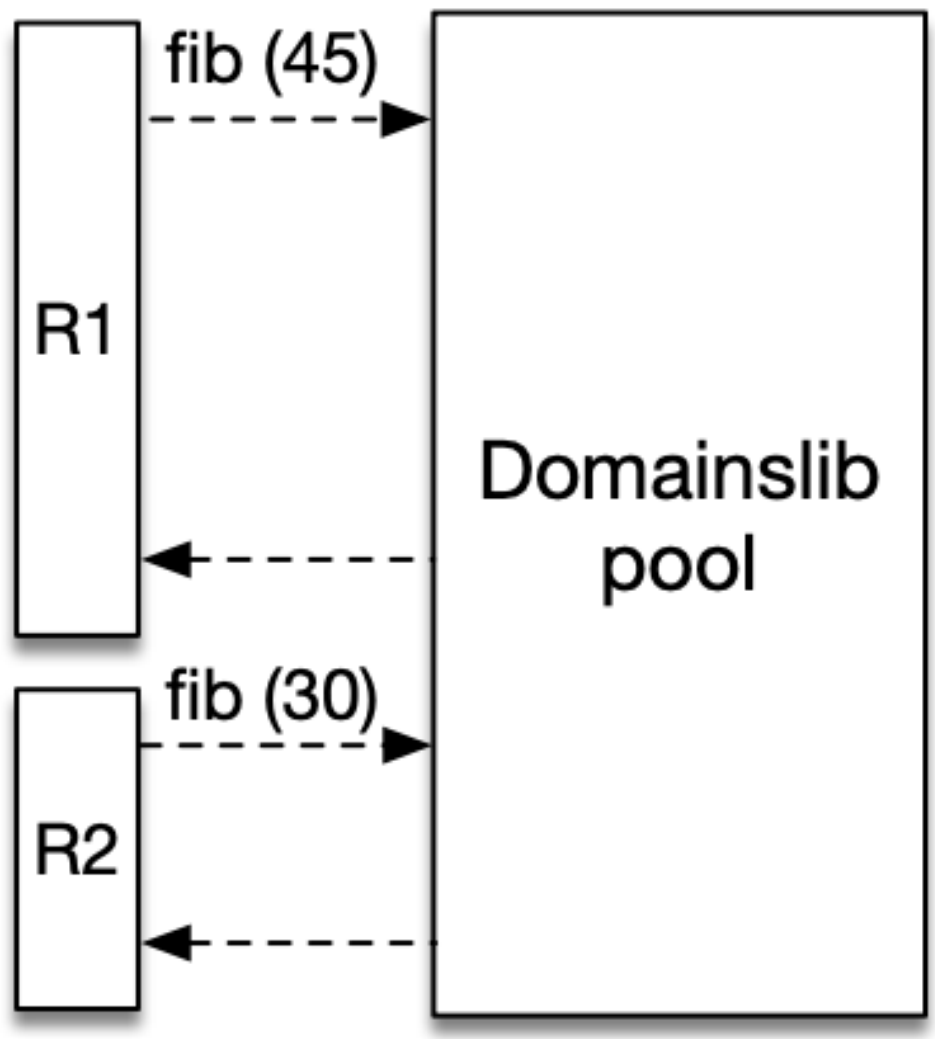
The trouble

```
module T = Domainslib.Task
(* set up a pool of [num_domains] domains for
   parallel computation *)
let pool = T.setup_pool ~num_domains ()

let main () =
  let sock = Eio.Net.listen ... in
  (* Runs once per request in an Eio task *)
  let request_handler n =
    T.run pool (fun _ -> fib_par n)
  in
  while true do
    (* spawn an Eio task to run [request_handler] per request *)
    Eio.Net.accept_fork sock ... request_handler ...
  done

let () = Eio_main.run main
```



Blocks the entire domain (Eio Scheduler)

Observed Behaviour



While the client network requests are handled concurrently, domainslib processing is serial

What's going wrong?

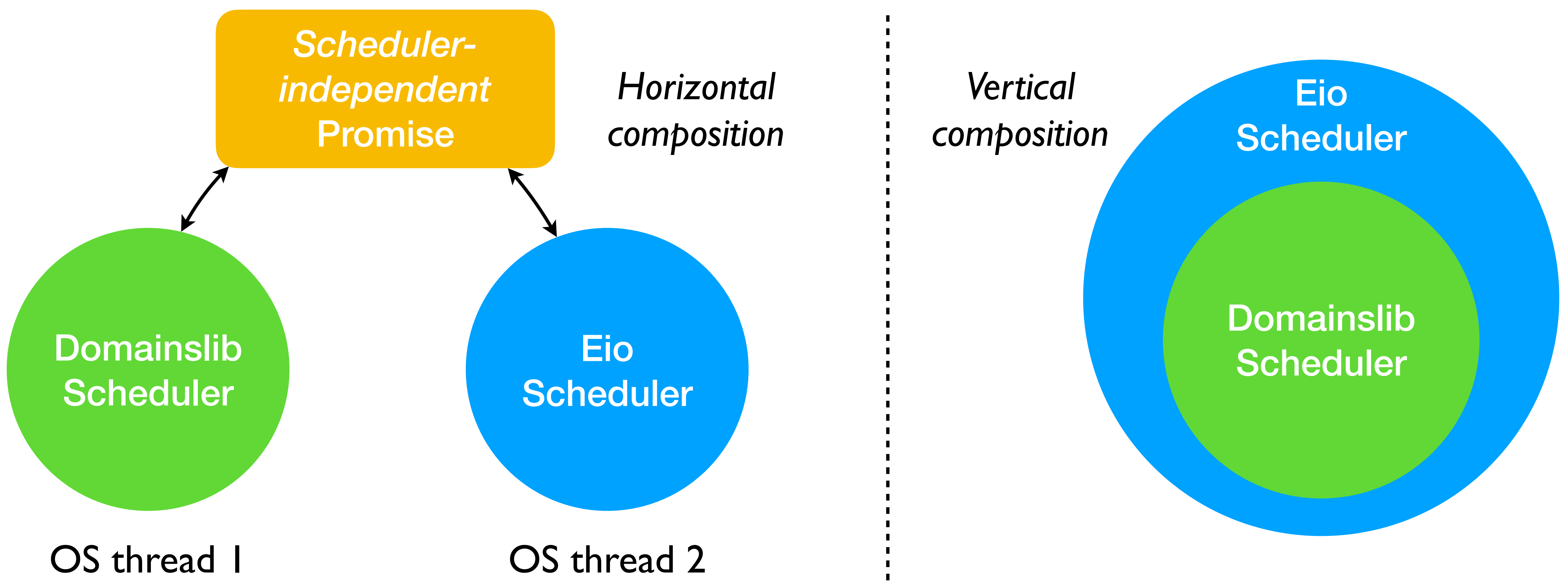
- What we needed
  - Eio task must wait for domainslib task completion
  - We used `Domainslib.Task.run` is a *domainslib-specific* blocking operation
    - Had to use it since there is no *scheduler-independent way of blocking*
- Deeper trouble
  - Every concurrency library implements its own set of *blocking data structures* — promises, channels / streams, MVars, mutex, condition, work-stealing queues, …
  - Often tricky (buggy) lock-free implementations
  - *All implementations are the same modulo the blocking behaviour!*

Why is it important?

- OCaml 4 IO ecosystem already split between Async & Lwt
  - Users must often pick one and stick to it
- OCaml 5 ecosystem may split between incompatible effect-based concurrency libraries
  - though purposes may be different — eio + domainslib
- Similar challenge in Rust
  - Tokio (Eio) for IO & Rayon (Domainslib) for data-parallelism
  - Bespoke tokio_rayon crate for *safely-mixing* the two
    - Bespoke composition not scalable!
- Need to solve this for all general purpose languages using effect handlers for concurrency — Wasm

Solution: Scheduler Effect

- A single Suspend effect to describe how to suspend and resume tasks
  - Schedulers handle Suspend effect
  - *Scheduler-independent blocking* concurrency libraries perform Suspend effect in order to block and unblock tasks



Suspend Effect

```
type 'a resumer = 'a -> unit
type _ Effect.t += Suspend : {block: (‘a resumer -> 'a option)} -> 'a t
```

- To *block* the current task, perform `Suspend {block}`
  - `block` is defined by the blocking data structure
  - `block` is applied to the `resumer` function
- To *unblock* the blocked task, apply `resumer` to a value
  - `resumer` defined by the scheduler
  - `resumer` closes over the delimited continuation
- Due to parallelism, the condition to block may no longer be true
  - `block` must return `None` to the scheduler to indicate successful blocking
  - `block` must return `Some v` to the scheduler to indicate immediate resumption with `v`

Handling Suspend

```
| Suspend {block} -> Some (fun (k: (a,_) continuation) ->
  let resumer v = (* closes over continuation [k] *)
    let wakeup = Queue.is_empty run_q in
    enqueue k v;
    if wakeup then begin
      (* Wake up this sleeping domain *)
      Mutex.lock m; Condition.signal c; Mutex.unlock m
    end
  in
  match block resumer with
  | None -> resume_next ()      (* Resume another task *)
  | Some v -> continue k v) (* Resume immediately *)
```

Scheduler-independent Promise

```
module type Promise = sig
  type 'a t
  val create : unit -> 'a t
  exception Already_filled
  val fill   : 'a t -> 'a -> unit
  val await  : 'a t -> 'a
end
```

```
type 'a state = Full of 'a
              | Empty of 'a resumer list
type 'a t = 'a state Atomic.t

let create () = Atomic.make (Empty [])

exception Already_filled
```

```
let rec fill pv =
  let old = Atomic.get p in
  match old with
  | Full _ -> raise Already_filled
  | Empty l ->
      if Atomic.compare_and_set p old (Full v)
      then List.iter (fun r -> r v) l (* resume waiters *)
      else fill p v (* CAS failure; retry *)
```

Scheduler-independent Promise

```
let await p =
  let rec block r =
    let old = Atomic.get p in
    match old with
    | Full v -> Some v (* Resume immediately *)
    | Empty l ->
      if Atomic.compare_and_set p old (Empty (r::l))
      then None    (* Blocked successfully *)
      else block r (* CAS failure; retry *)
  in
  let old = Atomic.get p in
  match old with
  | Full v -> v
  | _ -> perform (Suspend {block})
```

Synchronisation structures

- Able to implement all *blocking* data structures in scheduler-independent manner
  - Promises, Channels, Mutex, Condition, …
- Different concurrency libraries are able to dynamically use the same structure to *communicate* & *synchronise*
  - Better than functorising the data structure for a specific scheduler

Cancellation

- When tasks are cheap, cancellation becomes prominent
  - ✦ Parallel DFS — cancel parallel search tasks on finding the first match
  - ✦ Async IO — issue concurrent requests; cancel all when one fails
- Cancellation is varied
  - ✦ Structured concurrency — tree-structured hierarchy of tasks that are cancelled together
  - ✦ p2p cancellation — kill an individual task à la `pthread_kill`
- Suspend should be cancellation aware

Do not transfer lock to a cancelled task!

```
module Mutex = struct
  type state = Unlocked
             | Locked of unit resumer list
  type t = state Atomic.t
  ...
```

Cancellation — Scheduler

- Say our aim is to support a pthread_kill style API

```
type handle
val fork   : (unit -> unit) -> handle
val cancel : handle -> unit
```

```
type handle = {mutable cancelled : bool}
let cancel task = task.cancelled <- true
```

```
(* Scheduler maintains a queue of [task]s *)
type task = Task: handle * ('a,unit) continuation * 'a -> task
```

```
let rec resume_next () =
  match Queue.pop run_q with
  | Some (Task (handle, k, v)) -> (* resume the next task *)
      if handle.cancelled then discontinue k Exit else continue k v
  ...
```

Cancellation — Scheduler

```
type 'a resumer = 'a -> bool (* instead of [unit]; is task alive? *)
type _ Effect.t += Suspend: {block: (‘a resumer -> 'a option)} -> 'a t
```

```
| Suspend {block} -> Some (fun (k: (a,_) continuation) ->
  let resumer v =
    let wakeup = Queue.is_empty run_q in
    enqueue k v;
    if wakeup then begin
      Mutex.lock m; Condition.signal c; Mutex.unlock m
    end;
    not handle.cancelled
  in
  match block resumer with
  | None -> dequeue () (* Resume next task *)
  | Some v ->          (* Resume immediately *)
      if handle.cancelled then discontinue k Exit
      else continue k v)
```

Cancellation — Mutex

```
let rec unlock m =
  let old = Atomic.get m in
  match old with
  | Unlocked -> failwith “impossible"
  | Locked [] ->
      if Atomic.compare_and_set m old Unlocked
      then ()       (∗ Unlocked successfully ∗)
      else unlock m (∗ failed CAS; retry ∗)
  | Locked (r::rs) ->
      if Atomic.compare_and_set m old (Locked rs)
      then begin
        if r () then () (∗ Unlocked successfully & transferred control ∗)
        else unlock m   (∗ cancelled; wake up next task ∗)
      end else unlock m (∗ failed CAS; retry ∗)
```

Concurrency-aware Lazy

- OCaml deeply supports lazy computations
  - ✦ Syntax, lazy pattern matches, short-circuiting by the GC
- Not concurrency aware
  - ✦ Raises Undefined exception on recursive or concurrent forcing
- Lazy computations may have side effects
  - ✦ Concurrent tasks forcing a lazy need to be *blocked* and *unblocked*
  - ✦ Suspend effect to the rescue!
- Needs a change in lazy value representation (1 word larger) and Suspend type

```
type 'a resumer = ('a,exn) Result.t (* instead of ['a] *) -> bool
type _ Effect.t += Suspend: {block: (‘a resumer -> 'a option)} -> 'a t
```

Status

- Current solution does not use effect handlers but uses domain-local state — Domain-local await
  - ✦ Pragmatic decision — works with OCaml 4, which doesn't support effects; Use of domains & systhreads directly
- Data structure libraries build on domain-local await
  - ✦ Saturn — parallel data structures (lockfree & lock-based, composable & non-composable)
  - ✦ kcas — lock-free STM based on multi-word compare-and-swap
- Concurrency libraries build on domain-local await
  - ✦ Eio, Domainslib and Moonpool now use domain-local await

Q: Are effect handlers necessary for this?

Specification

The effect signature is hard to comprehend

```
type 'a resumer = ('a,exn) Result.t -> bool
type _ Effect.t += Suspend: {block: (‘a resumer -> 'a option)} -> 'a t
```

Q: How to better specify expectations on the scheduler and the data structures?

Equations?
Refinements?
...

Conclusion

- The ability to define own concurrency libraries using effect handlers may lead to monolithic and incompatible libraries
- Suspend effect to define blocking and unblocking semantics
  - ✦ Permits concurrency library composition
  - ✦ Permits scheduler-independent blocking data structures
- Working draft
  - ✦ Deepali et al, "Effectively Composing Concurrency Libraries", https://kcsrk.info/papers/composable_concurrency.pdf
  - ✦ Includes
    - ✤ composing monadic libraries
    - ✤ Details of changes to lazy blocks