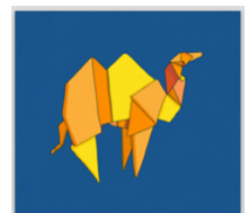


Retrofitting *Effect handlers* onto OCaml

“KC” Sivaramakrishnan

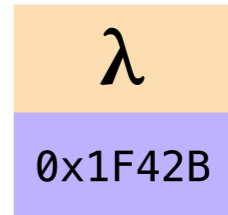
IIT
MADRAS



OCaml
Labs



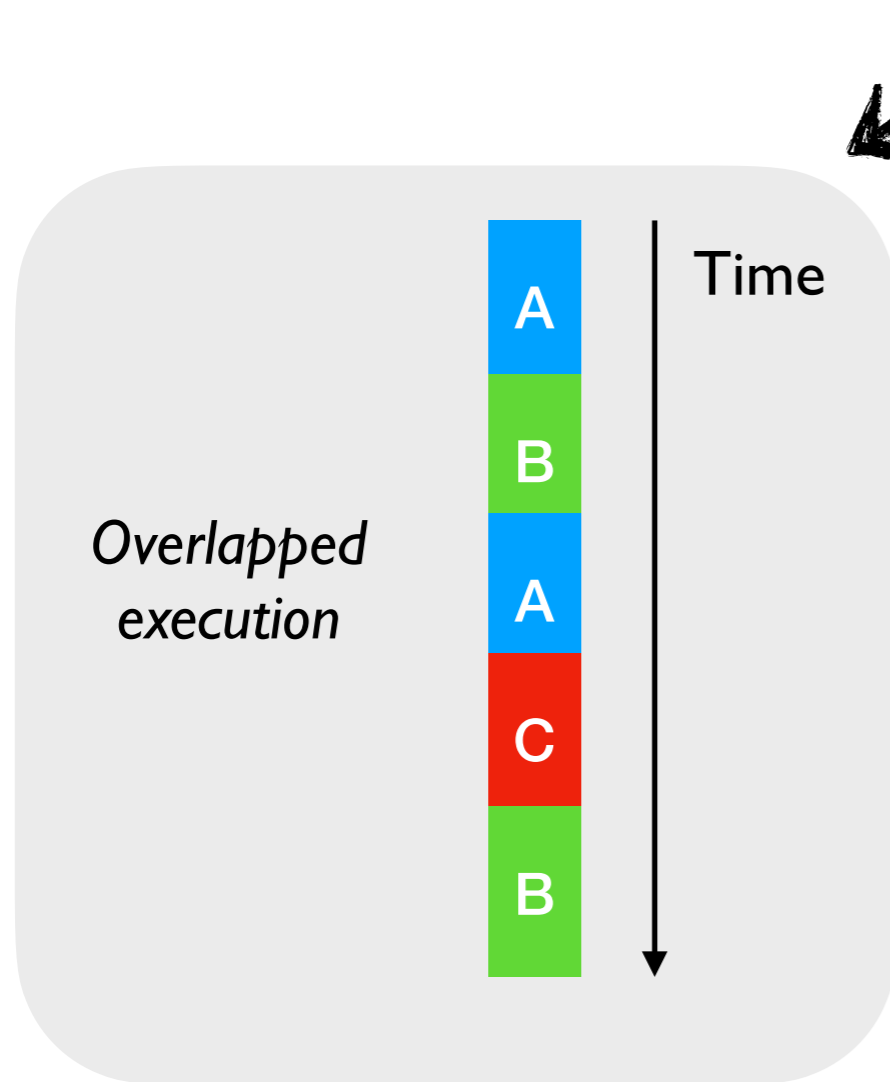
Tarides



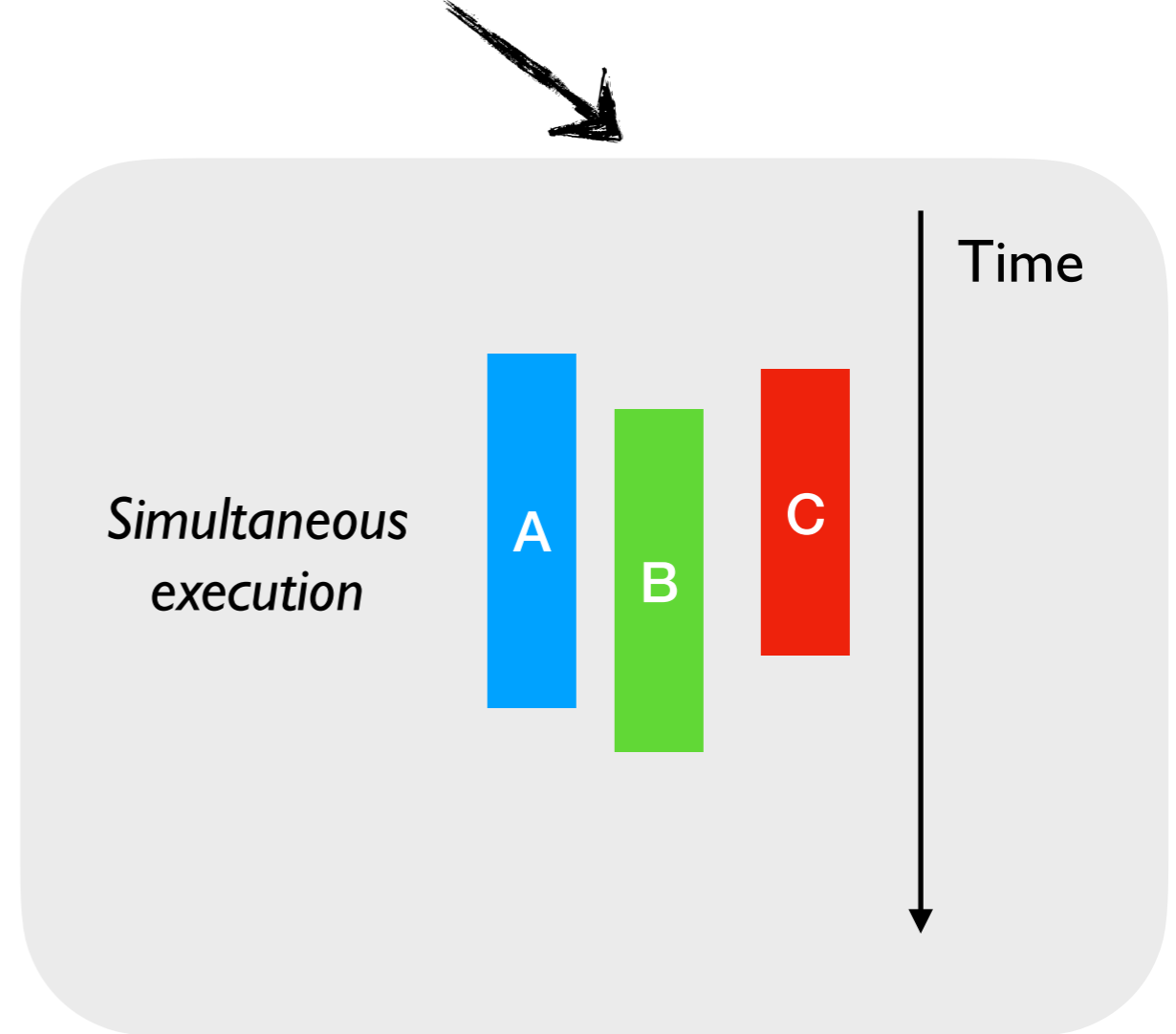
Segfault
Systems

Multicore OCaml

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



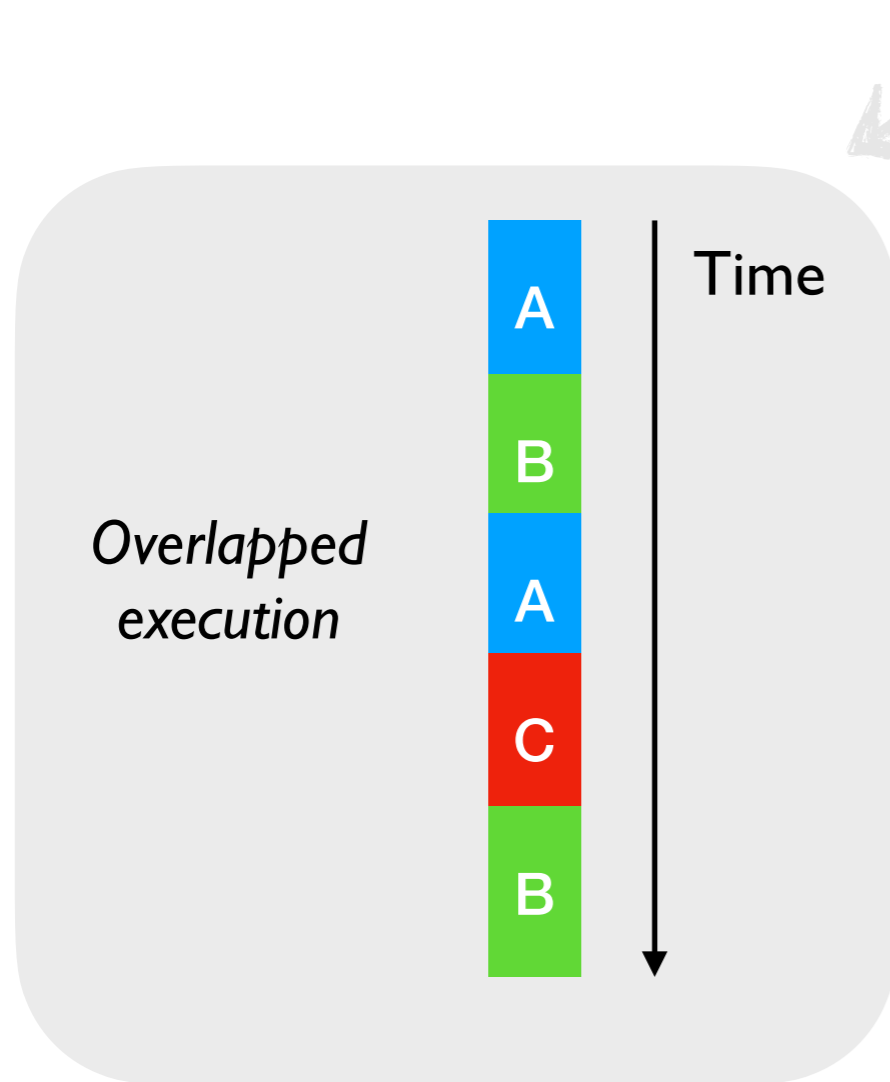
Effect Handlers



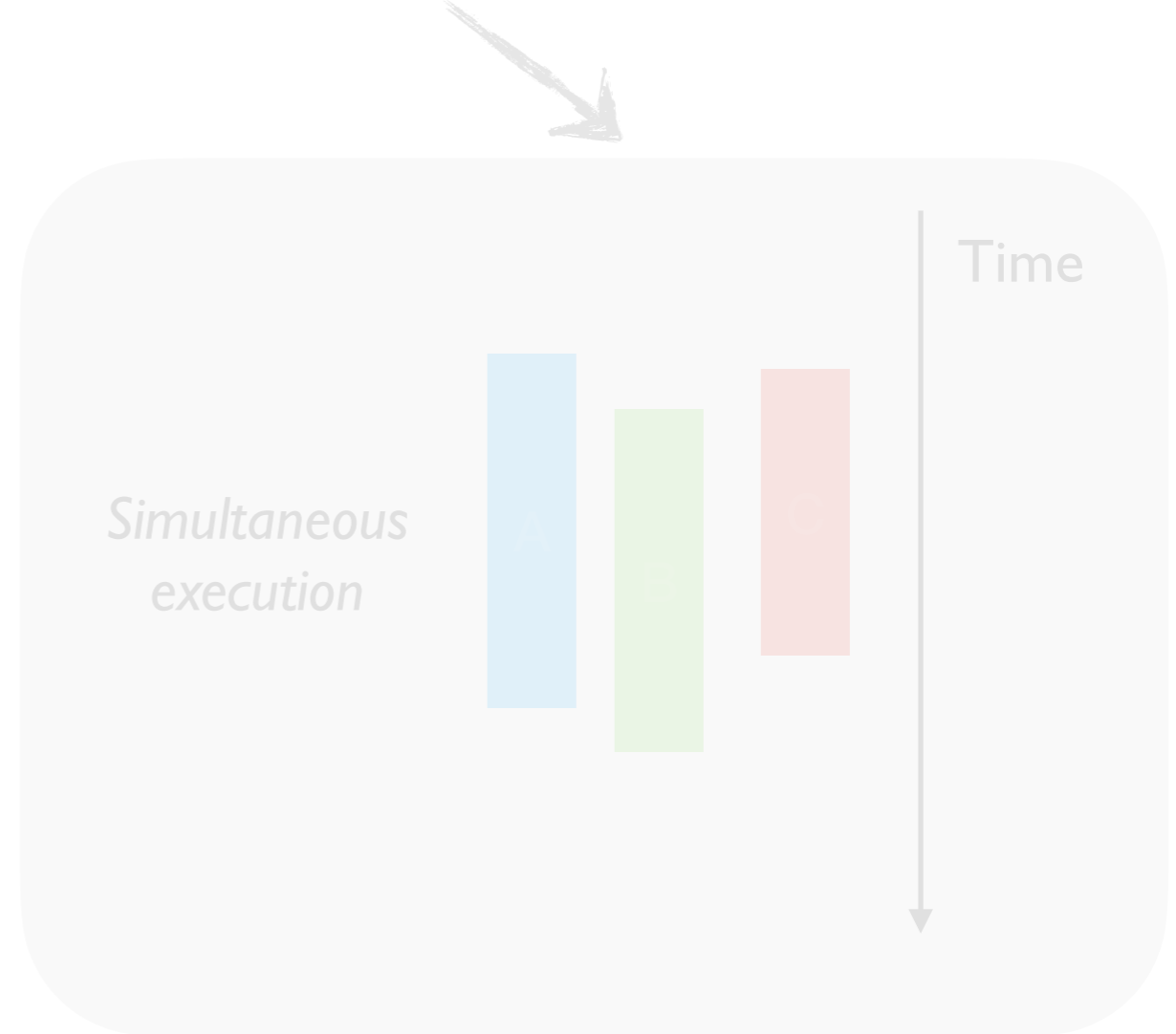
Domains

Multicore OCaml

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



Effect Handlers



Domains

Concurrent Programming

- Computations may be *suspended* and *resumed* later

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, ...

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, ...
- Often include different primitives for concurrent programming
 - ✦ JavaScript has *async/await*, *generators*, *promises*, and *callbacks*!!

Concurrent Programming in OCaml

- No primitive support for concurrent programming in OCaml
 - ✦ **Lwt** and **Async** - concurrent programming libraries
 - ✦ Callback-oriented programming with monadic syntax $>>=$

Concurrent Programming in OCaml

- No primitive support for concurrent programming in OCaml
 - ✦ **Lwt** and **Async** - concurrent programming libraries
 - ✦ Callback-oriented programming with monadic syntax $>>=$
- Suffers many pitfalls of *callback-oriented programming*
 - ✦ No backtraces, no exceptions, more closures

Concurrent Programming in OCaml

- No primitive support for concurrent programming in OCaml
 - ✦ **Lwt** and **Async** - concurrent programming libraries
 - ✦ Callback-oriented programming with monadic syntax $>>=$
- Suffers many pitfalls of *callback-oriented programming*
 - ✦ No backtraces, no exceptions, more closures
- Monads split the ecosystem into *Asynchronous* and *Synchronous*
 - ✦ Bob Nystrom, “*What colour is your function?*”

Concurrent Programming in OCaml

- No primitive support for concurrent programming in OCaml
 - ✦ **Lwt** and **Async** - concurrent programming libraries
 - ✦ Callback-oriented programming with monadic syntax $>>=$
- Suffers many pitfalls of *callback-oriented programming*
 - ✦ No backtraces, no exceptions, more closures
- Monads split the ecosystem into *Asynchronous* and *Synchronous*
 - ✦ Bob Nystrom, “*What colour is your function?*”
- Go (goroutines) and GHC Haskell (threads) have better abstractions — lightweight threads
 - ✦ *Should we add lightweight threads to OCaml?*

Solution

Effect Handlers

- A mechanism for programming with *user-defined effects*

Solution

Effect Handlers

- A mechanism for programming with *user-defined effects*
- *Modular* and *composable* basis of non-local control-flow mechanisms
 - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines as *libraries*

Solution

Effect Handlers

- A mechanism for programming with *user-defined effects*
- *Modular* and *composable* basis of non-local control-flow mechanisms
 - ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines as *libraries*
- Effect handlers \sim *first-class, restartable exceptions*
 - ✦ Structured programming with *delimited continuations*

Solution

Effect Handlers

- A mechanism for programming with *user-defined effects*
 - *Modular* and *composable* basis of non-local control flow mechanisms
 - ✦ Exceptions, generators, lightweight threads, IO, coroutines as *libraries*
 - Effect handlers \sim *first-class, restartable*
 - ✦ Structured programming with *delimited*
- Direct-style asynchronous I/O
 - Generators
 - Resumable parsers
 - Probabilistic Programming
 - Reactive UIs
 -

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

An example

```
effect E : string
```

```
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 "
```


An example

effect declaration

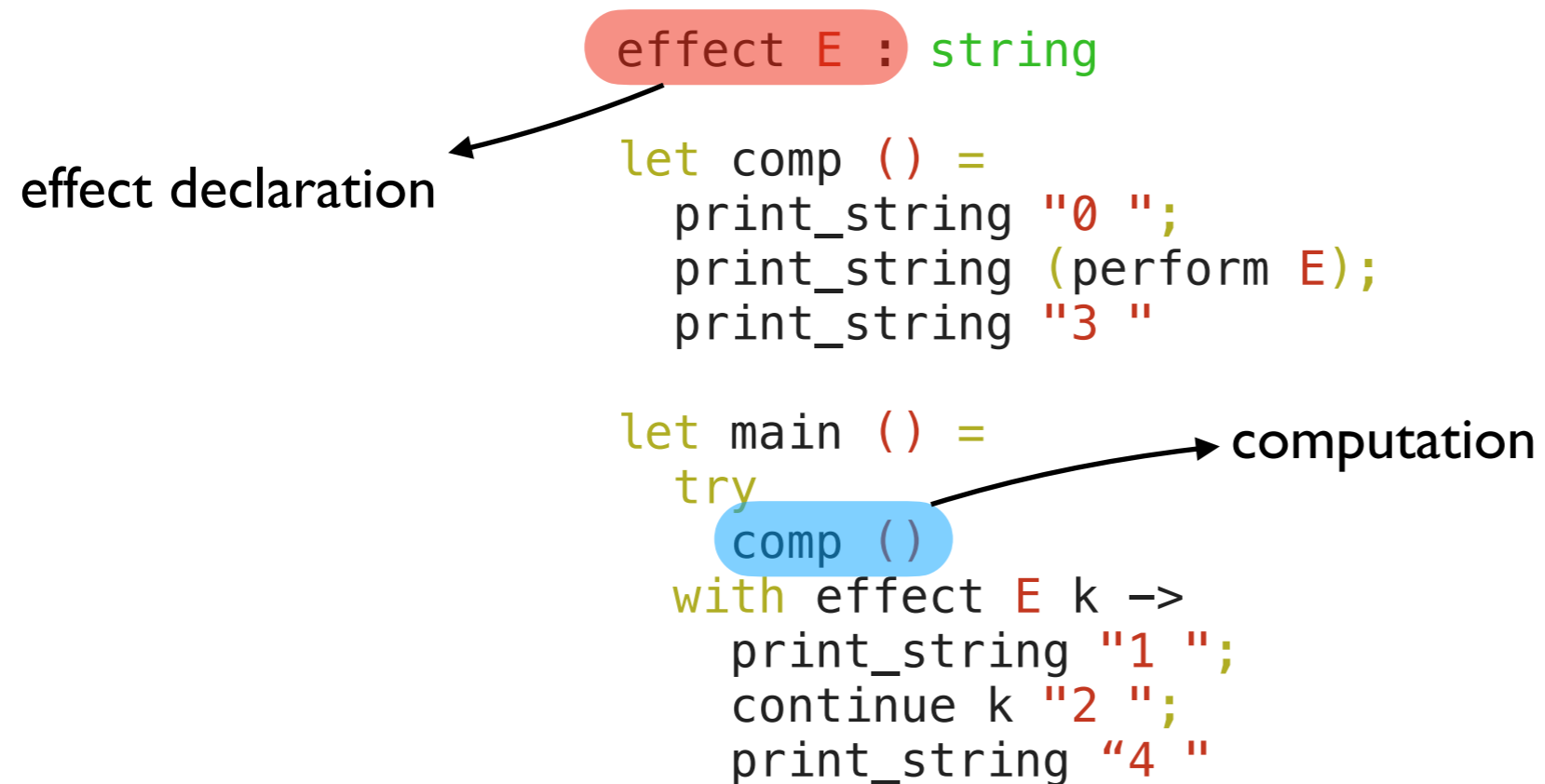
```
effect E : string

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

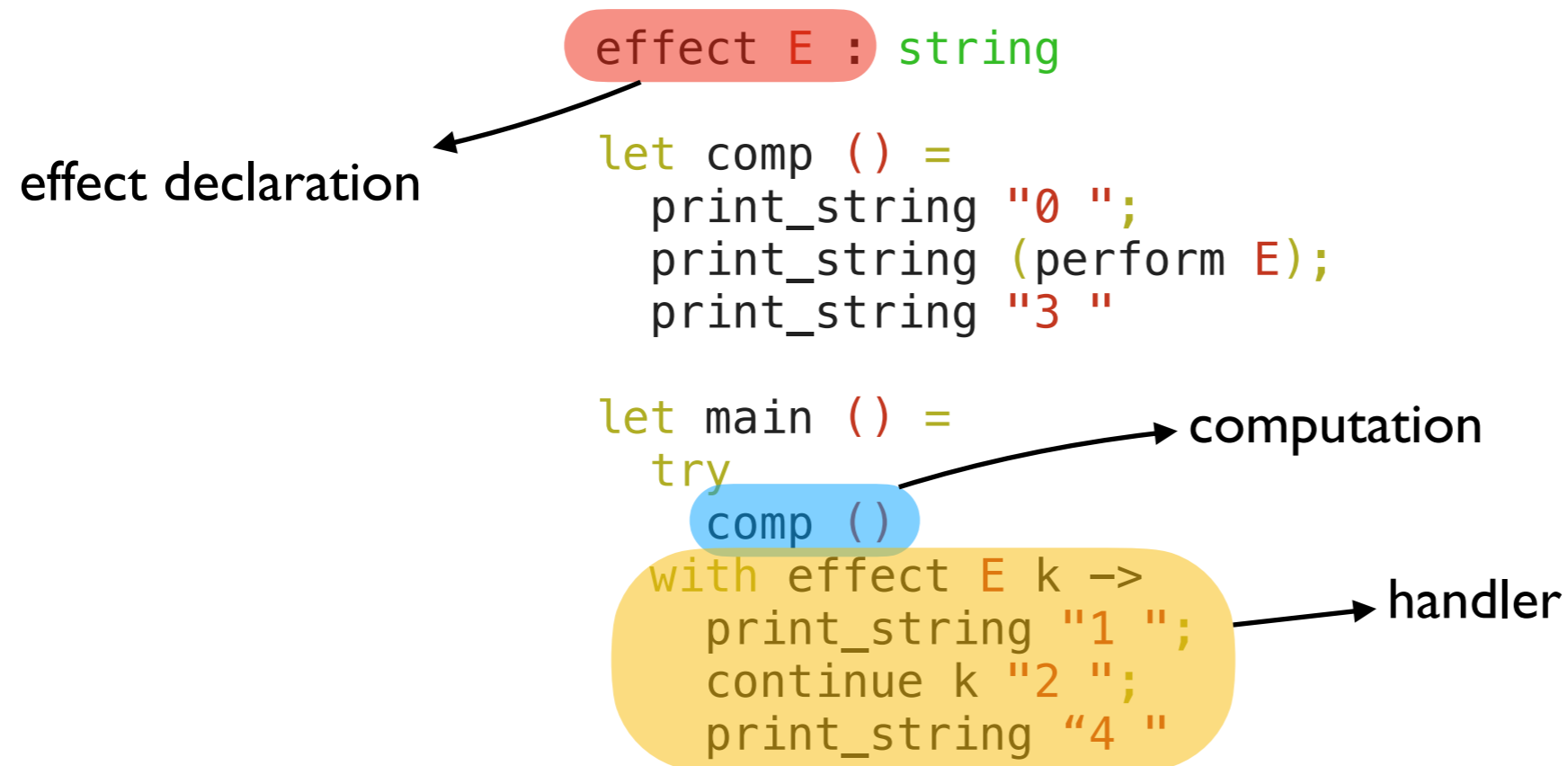
let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```



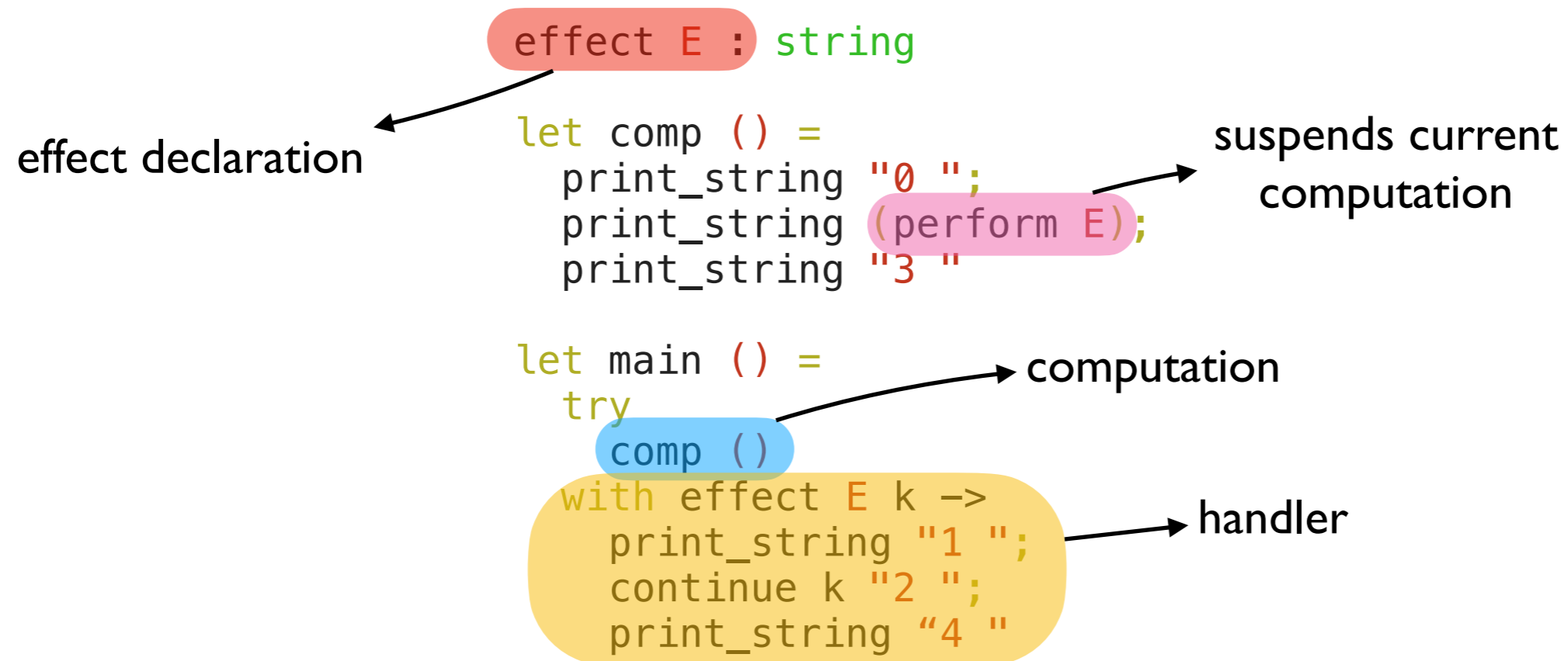
An example



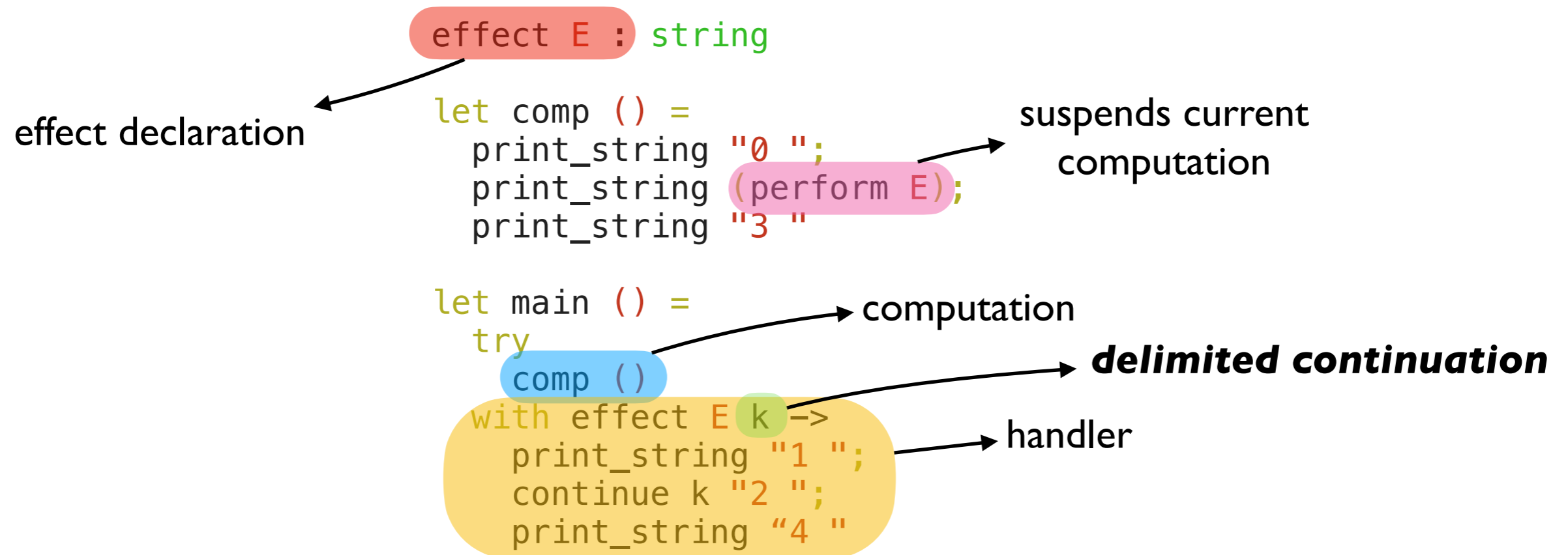
An example



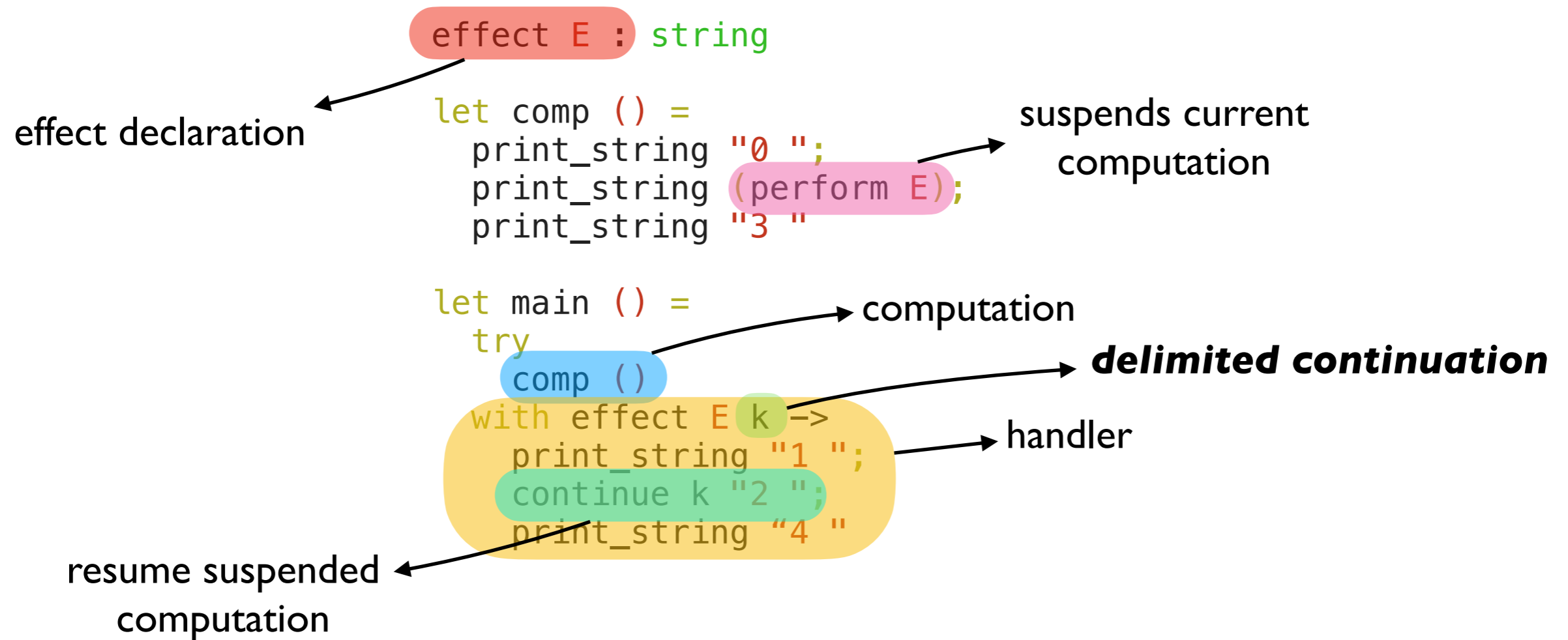
An example



An example



An example



Stepping through the example

```
effect E : string
```

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

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



Stepping through the example

```
effect E : string
```

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

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

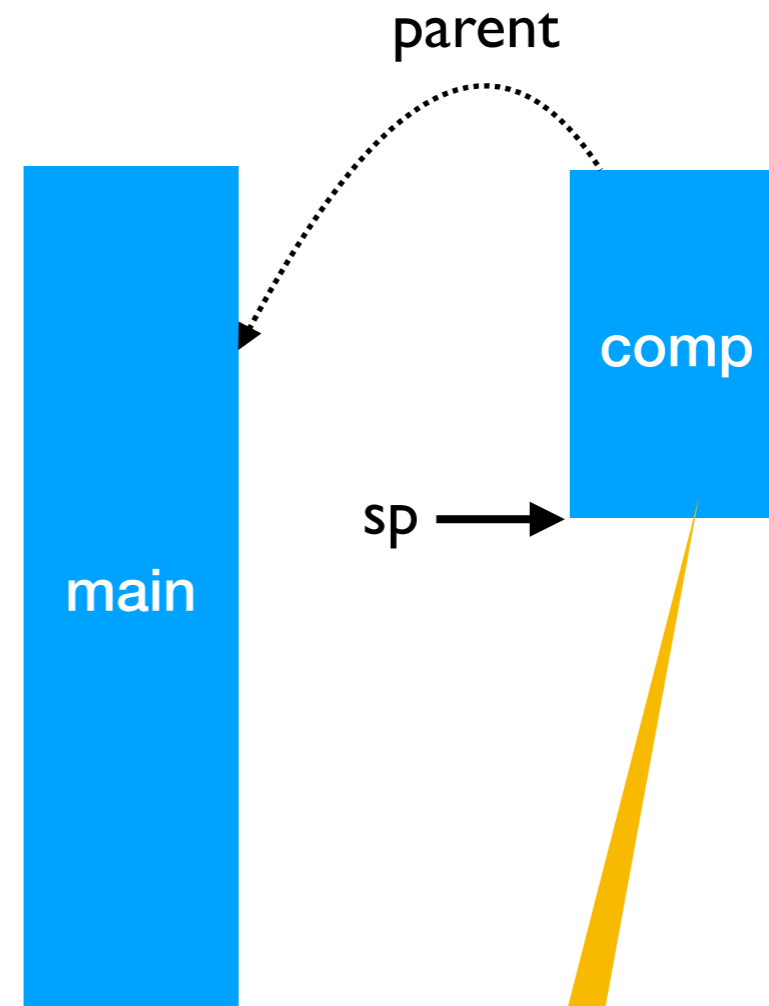


Stepping through the example

```
effect E : string
```

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

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



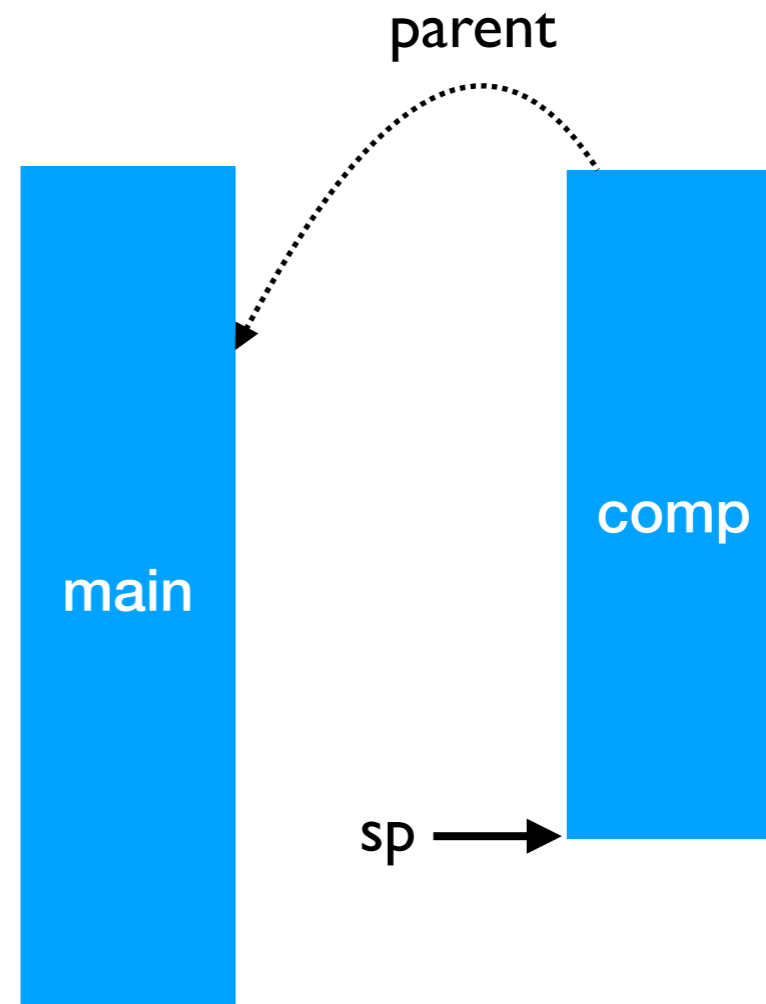
Fiber: A piece of stack
+ effect handler

Stepping through the example

```
effect E : string

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

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

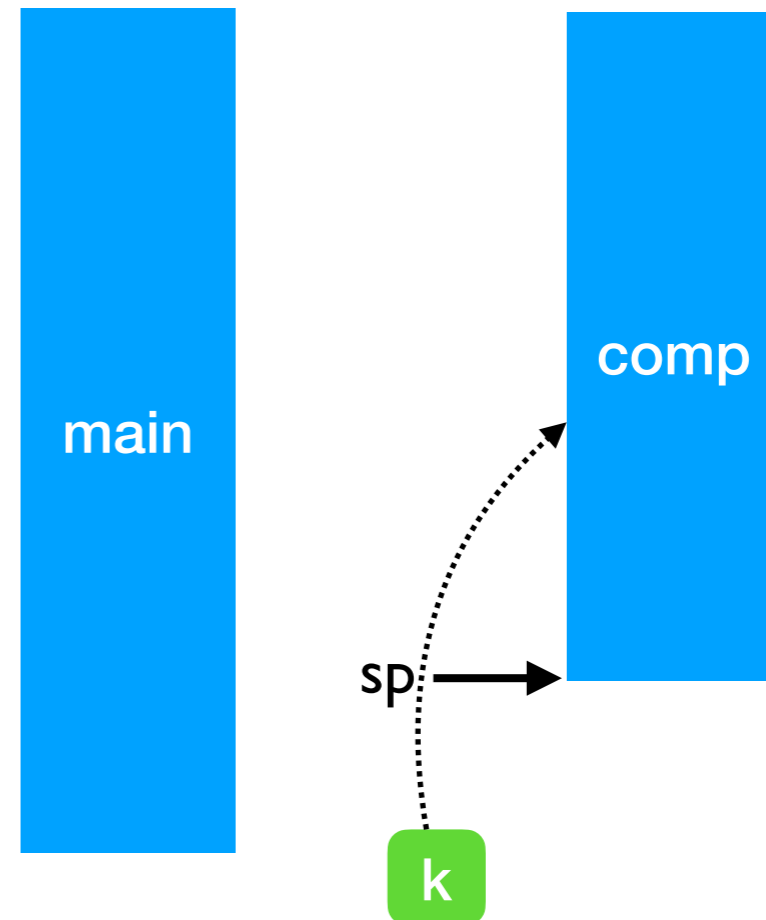


Stepping through the example

```
effect E : string

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

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

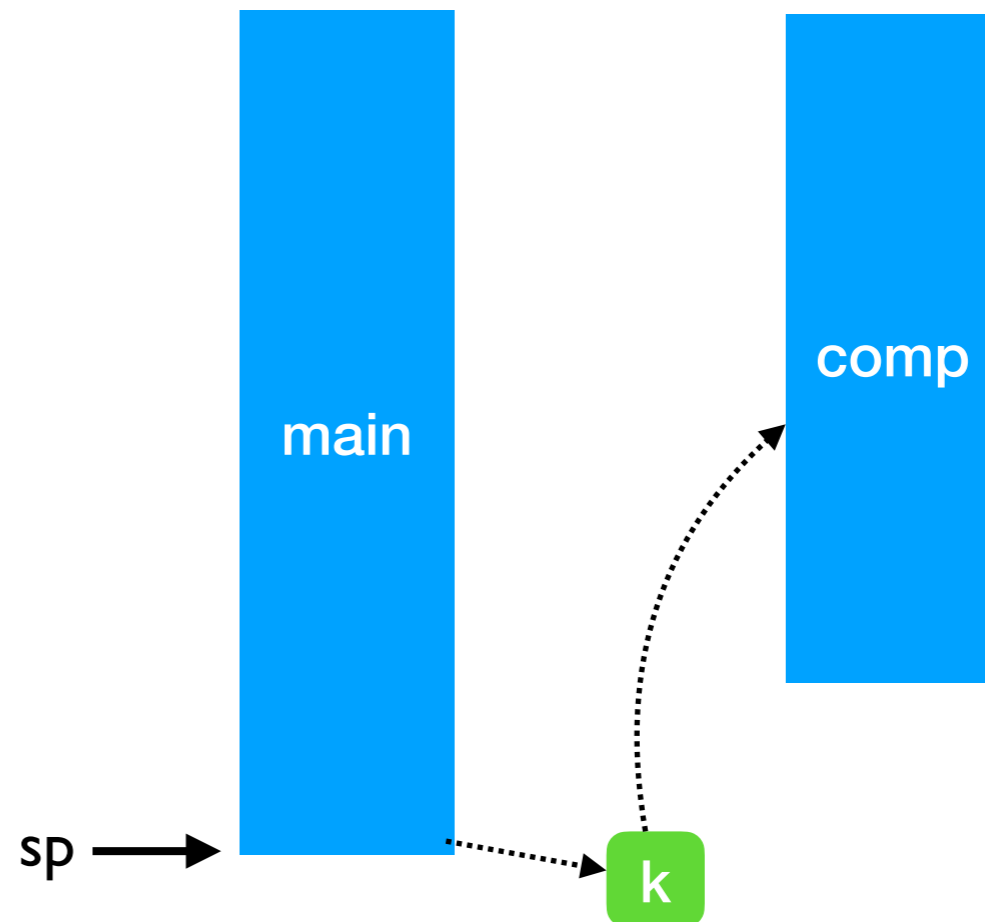


Stepping through the example

```
effect E : string

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

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



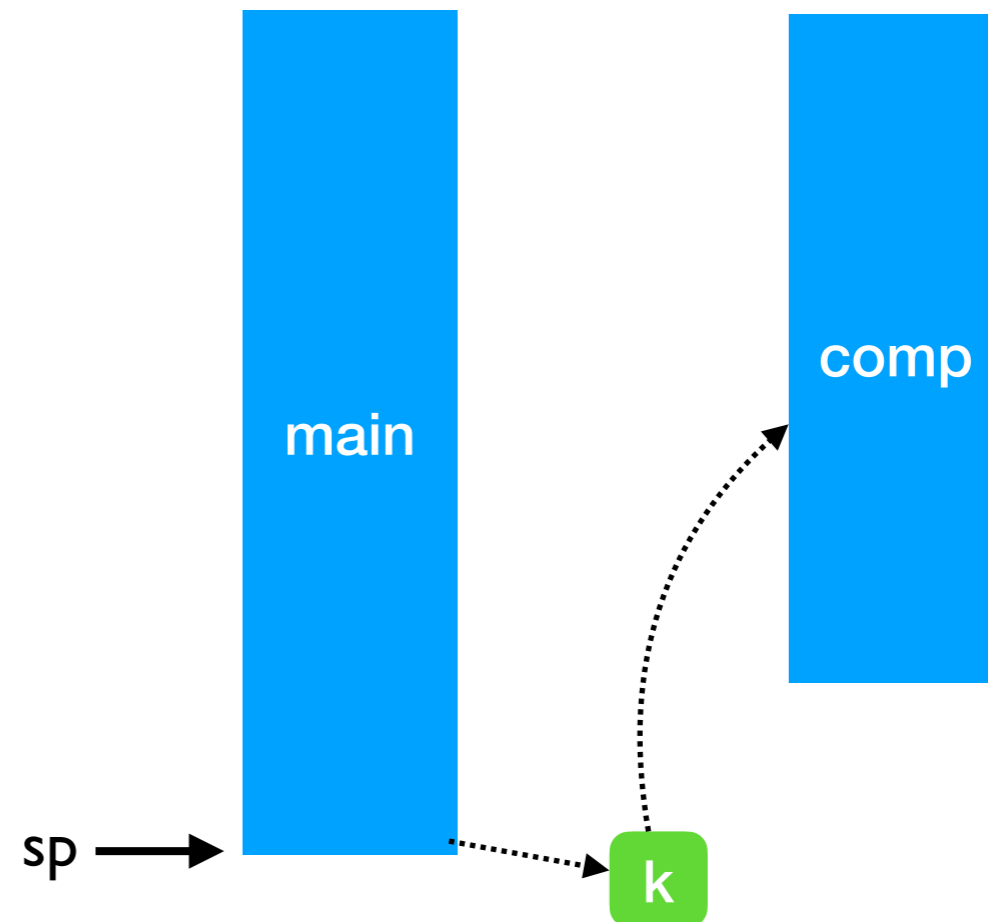
Stepping through the example

```
effect E : string
```

```
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 →



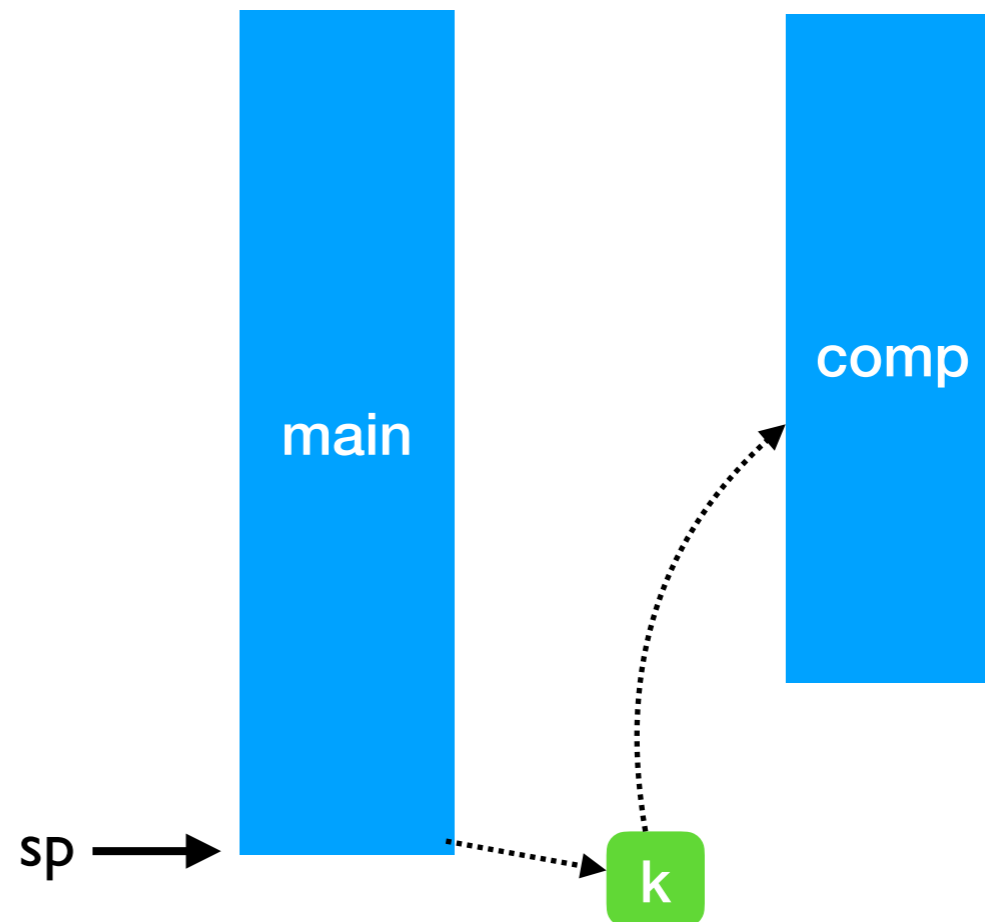
Stepping through the example

```
effect E : string
```

```
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 |

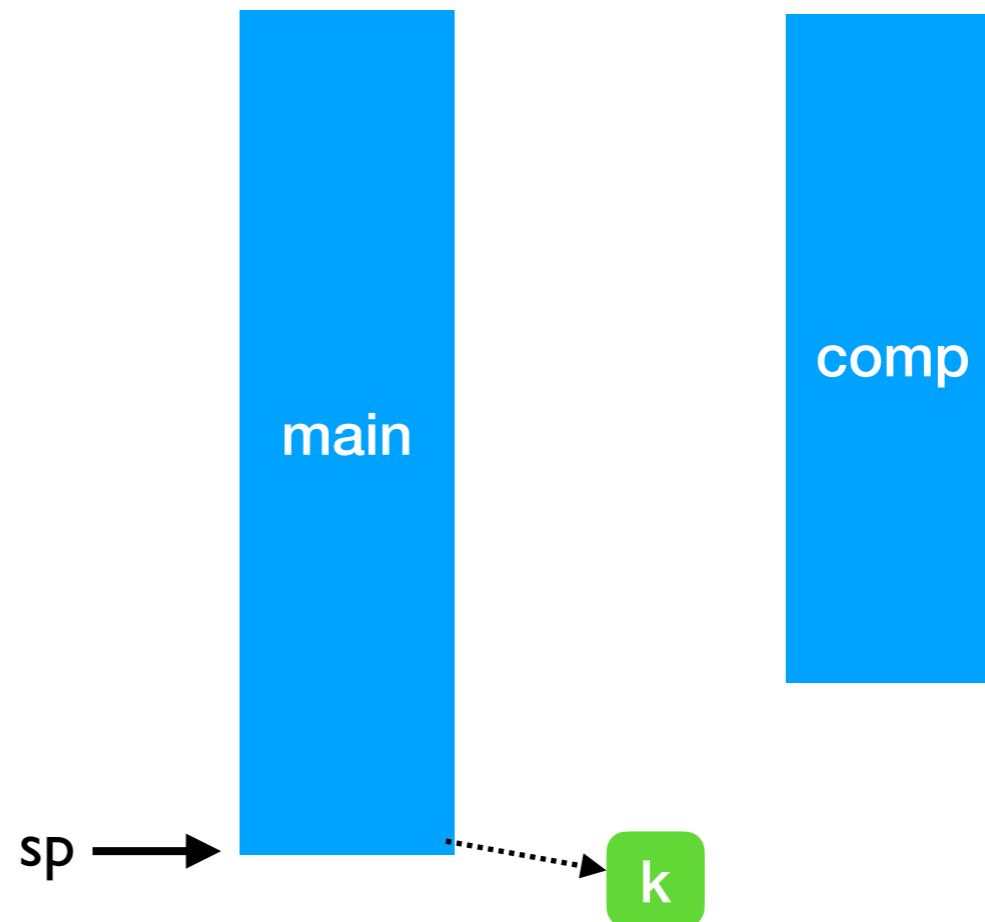
Stepping through the example

```
effect E : string
```

```
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 |

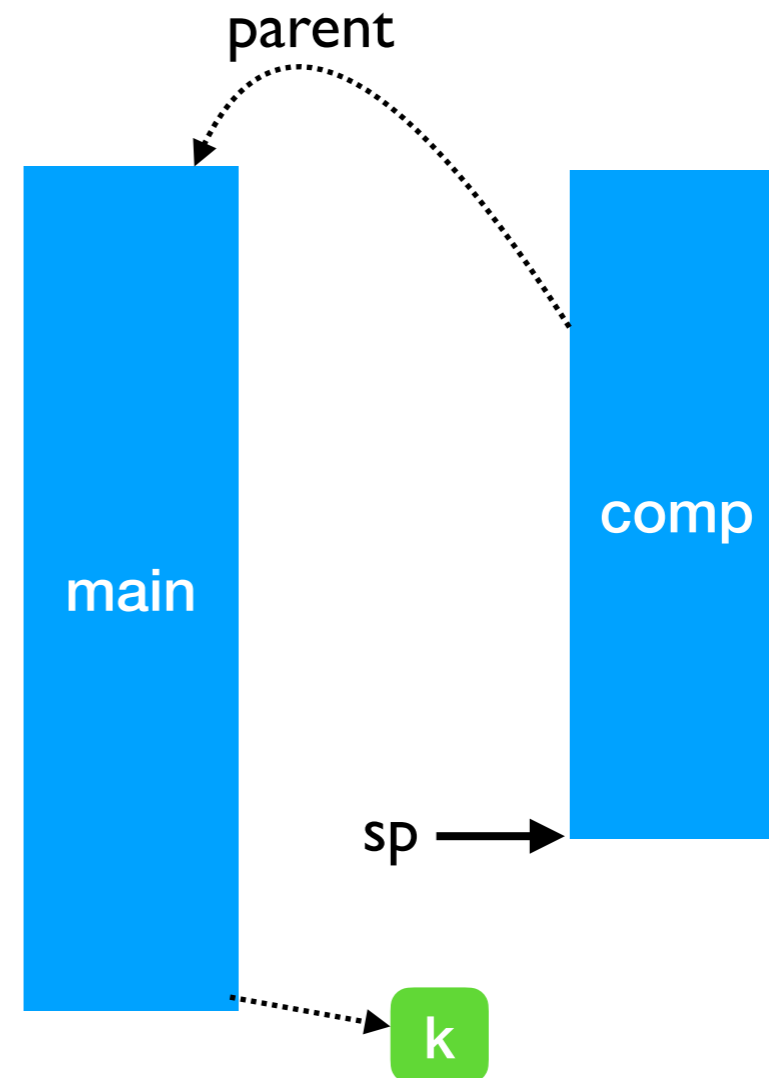
Stepping through the example

```
effect E : string
```

```
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 →



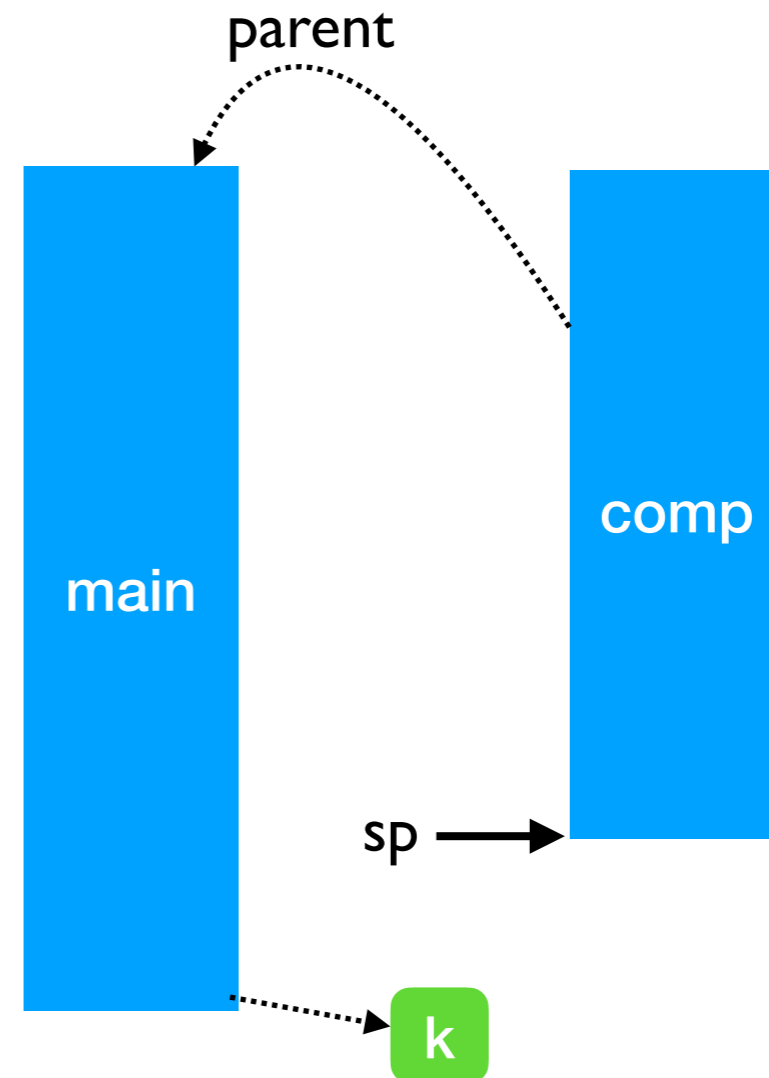
0 |

Stepping through the example

```
effect E : string

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

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



0 | 2

Stepping through the example

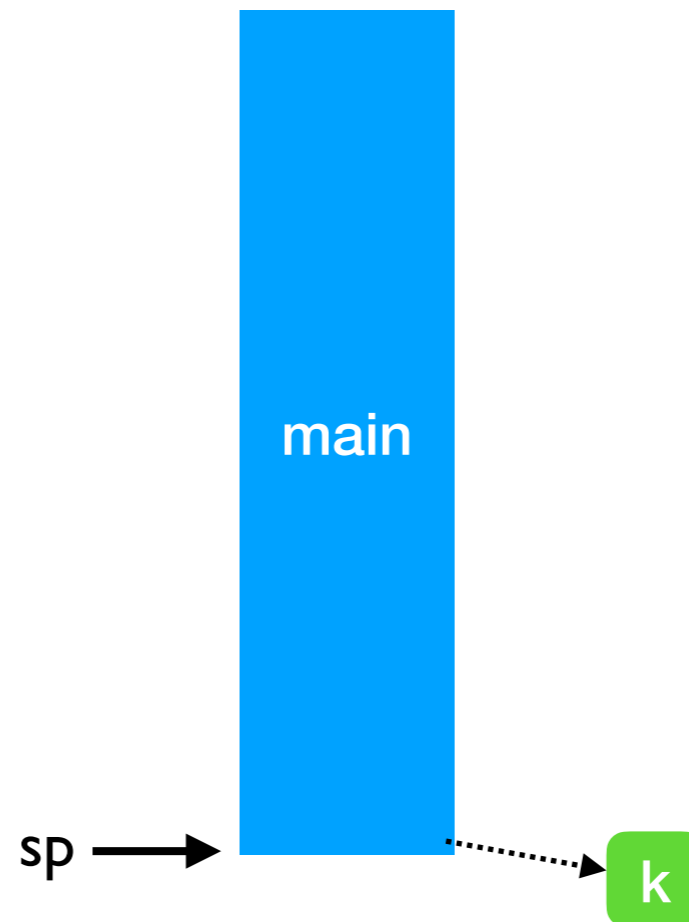
```
effect E : string
```

```
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



Stepping through the example

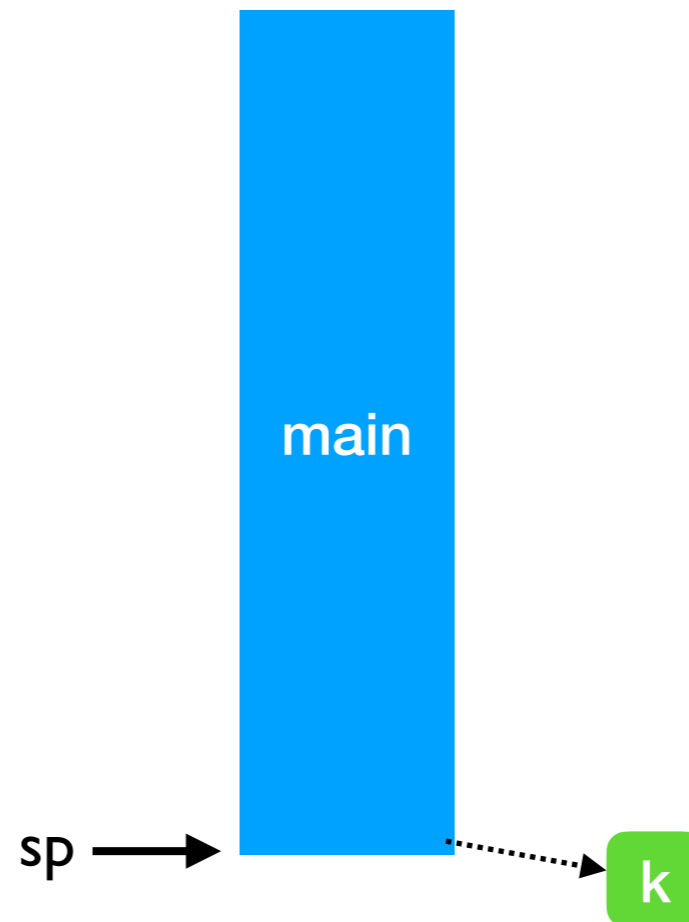
```
effect E : string
```

```
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



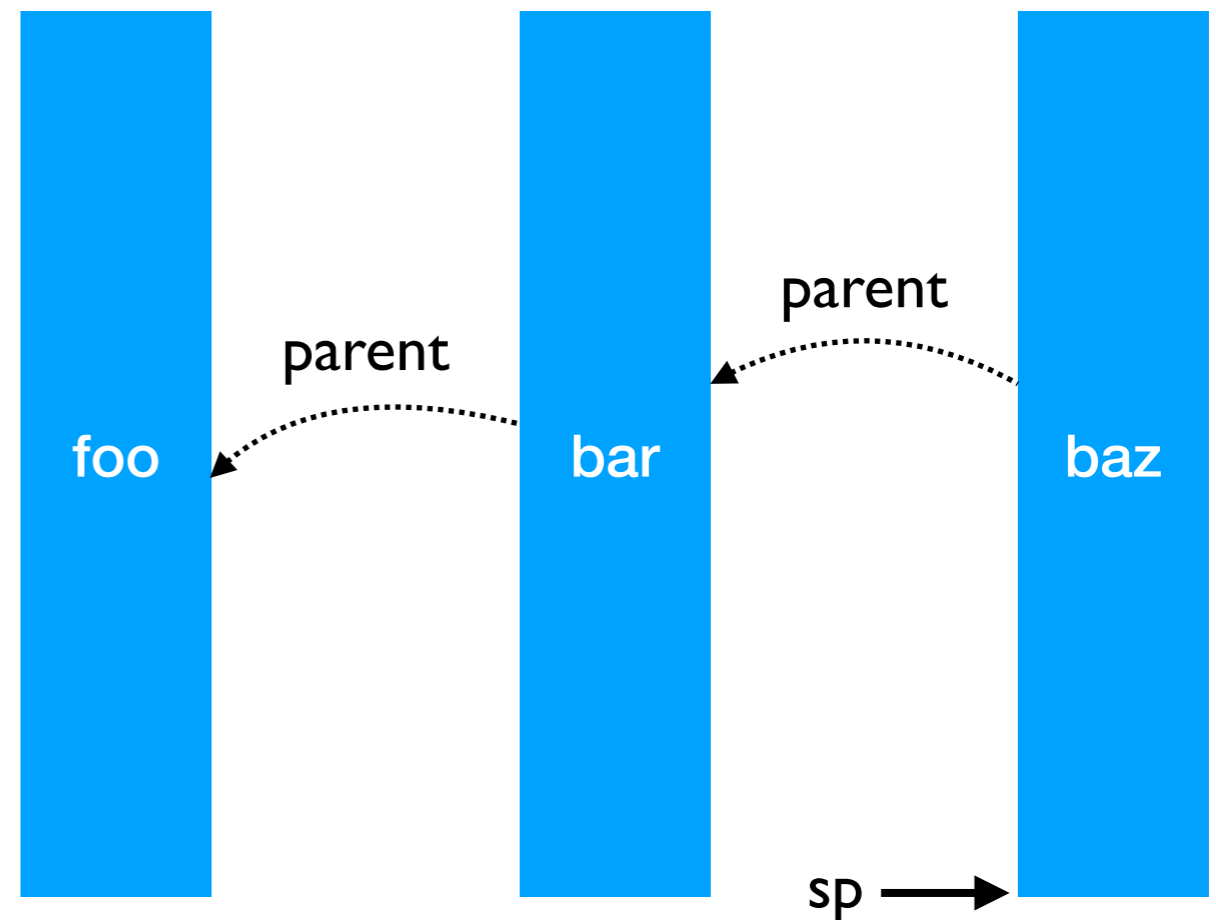
Handlers can be nested

```
effect A : unit  
effect B : unit
```

```
let baz () =  
pc → perform A
```

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

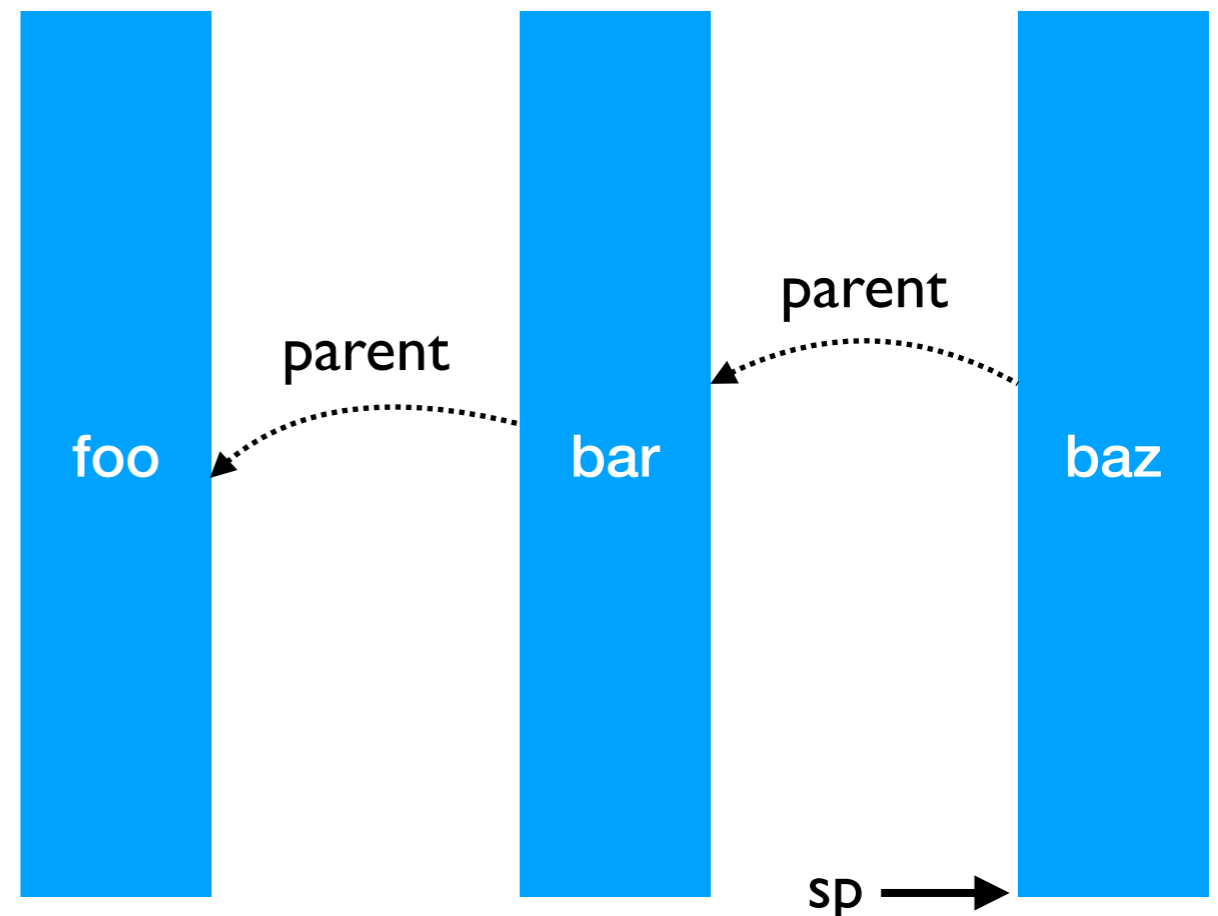
```
let foo () =  
  try  
    bar ()  
  with effect A k ->  
    continue k ()
```



Handlers can be nested

```
effect A : unit  
effect B : unit
```

```
let baz () =  
pc → perform A  
  
let bar () =  
  try  
    baz ()  
  with effect B k ->  
    continue k ()  
  
let foo () =  
  try  
    bar ()  
  with effect A k ->  
    continue k ()
```

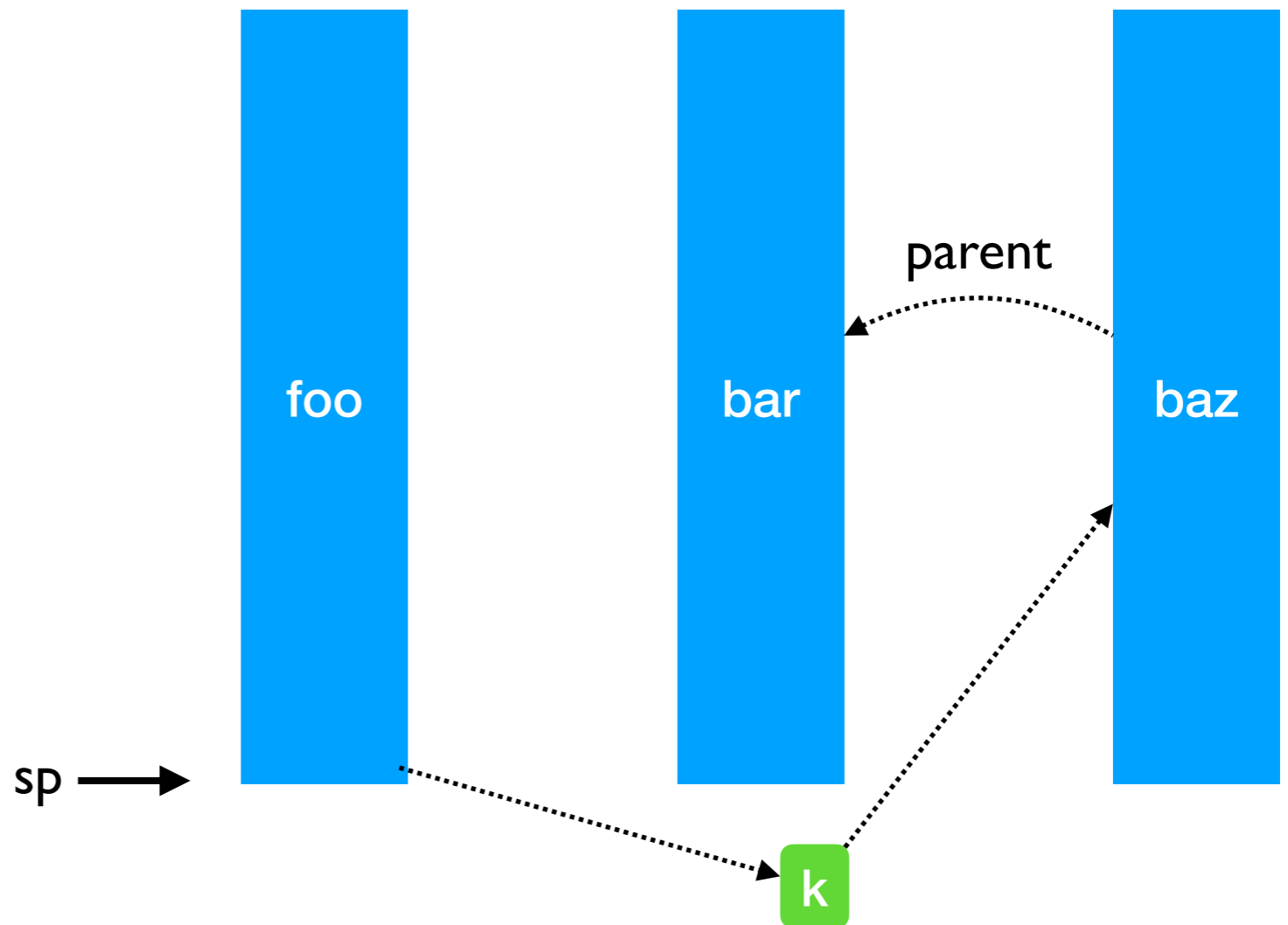


- Linear search through handlers
 - *Handler stacks shallow in practice*

Handlers can be nested

```
effect A : unit  
effect B : unit
```

```
pc → let baz () =  
      perform A  
  
      let bar () =  
        try  
          baz ()  
        with effect B k ->  
          continue k ()  
  
      let foo () =  
        try  
          bar ()  
        with effect A k ->  
          continue k ()
```



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

Handlers can be nested

```
effect A : unit  
effect B : unit
```

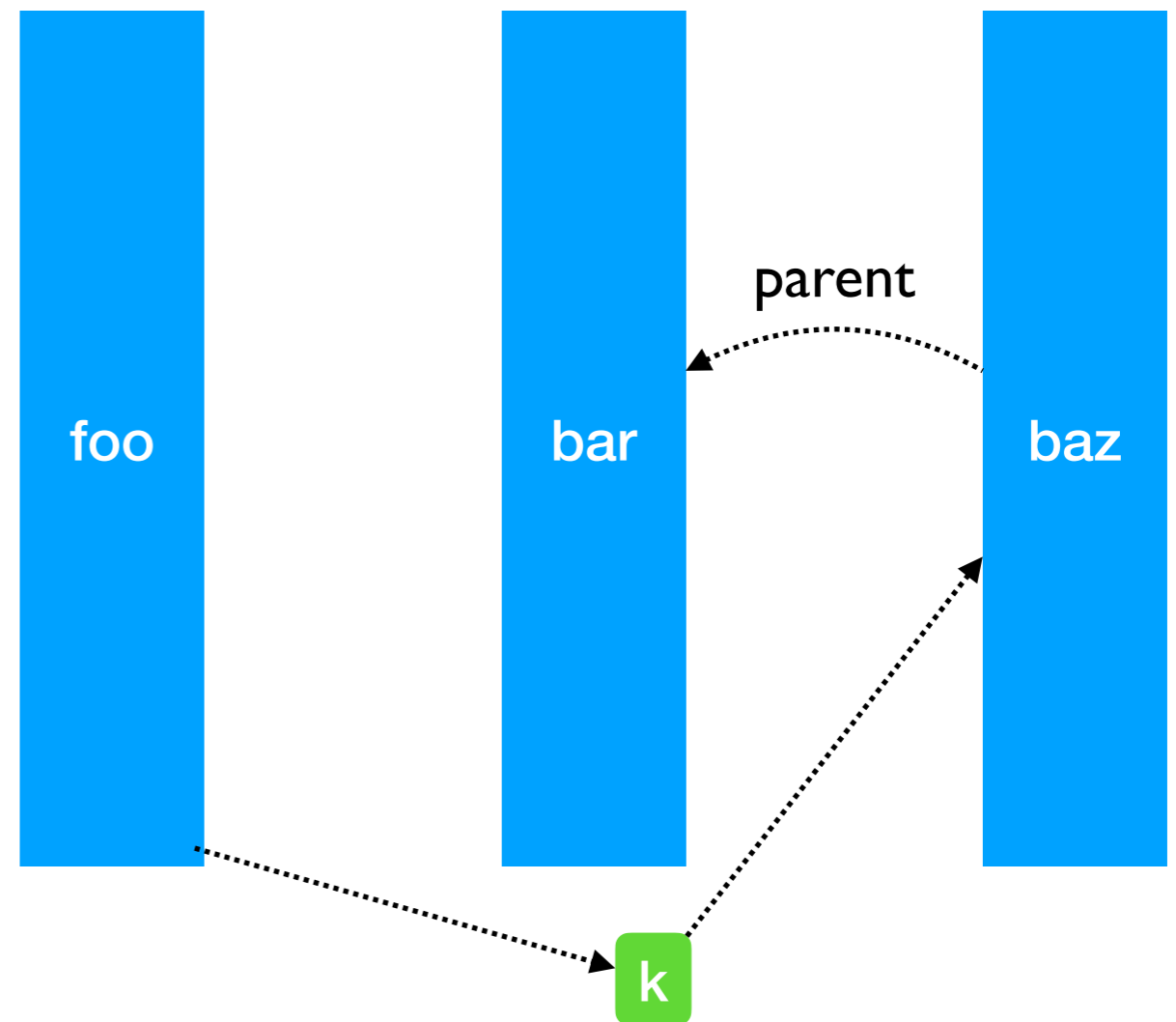
```
let baz () =  
  perform A
```

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

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

pc →

sp →



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

Lightweight Threading

```
effect Fork  : (unit -> unit) -> unit  
effect Yield : unit
```

Lightweight Threading

```
effect Fork   : (unit -> unit) -> unit
effect Yield : unit

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
```

Lightweight Threading

```
effect Fork   : (unit -> unit) -> unit
effect Yield  : unit
```

```
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
```

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
```

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

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

Generators

Generators

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

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
```

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
```

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 effect Yield : a -> unit 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 = MGen (struct
  type 'a t = 'a list
  let iter = List.iter
end)
```

Generators: List

```
module L = MGen (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)
```

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)
```

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
```

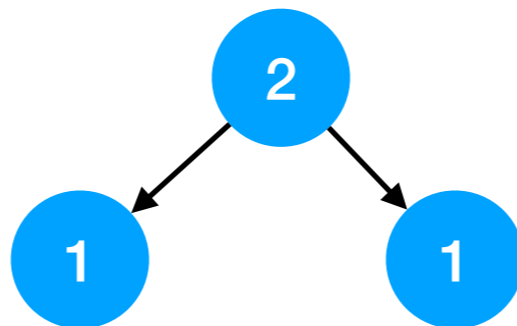
```
module T = MkGen(struct  
  type 'a t = 'a tree  
  let iter = iter  
end)
```

(* 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 *)
```



Retrofitting Challenges

- Millions of lines of legacy code
 - ✦ Written without *non-local control-flow* in mind
 - ✦ Cost of refactoring sequential code itself is *prohibitive*

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

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**

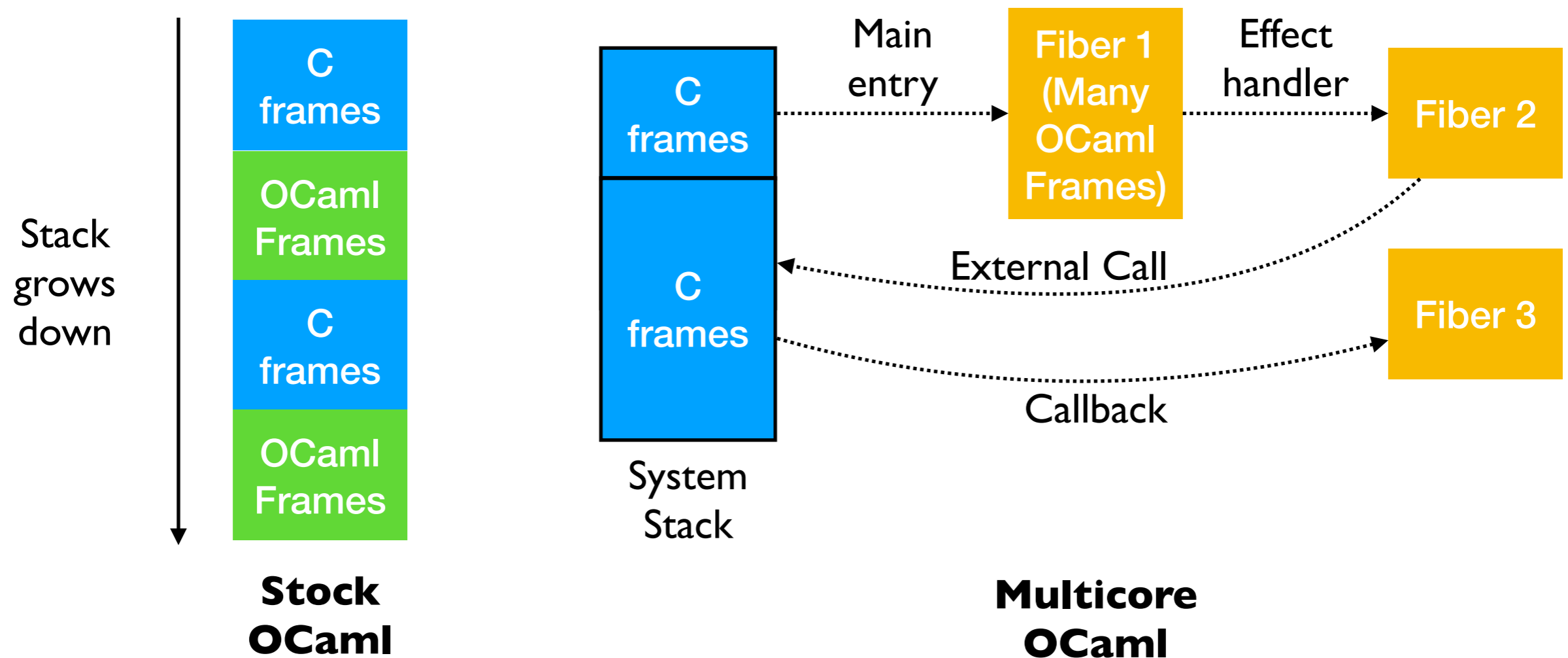
Representing Stacks & Continuations

Representing Stacks & Continuations

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

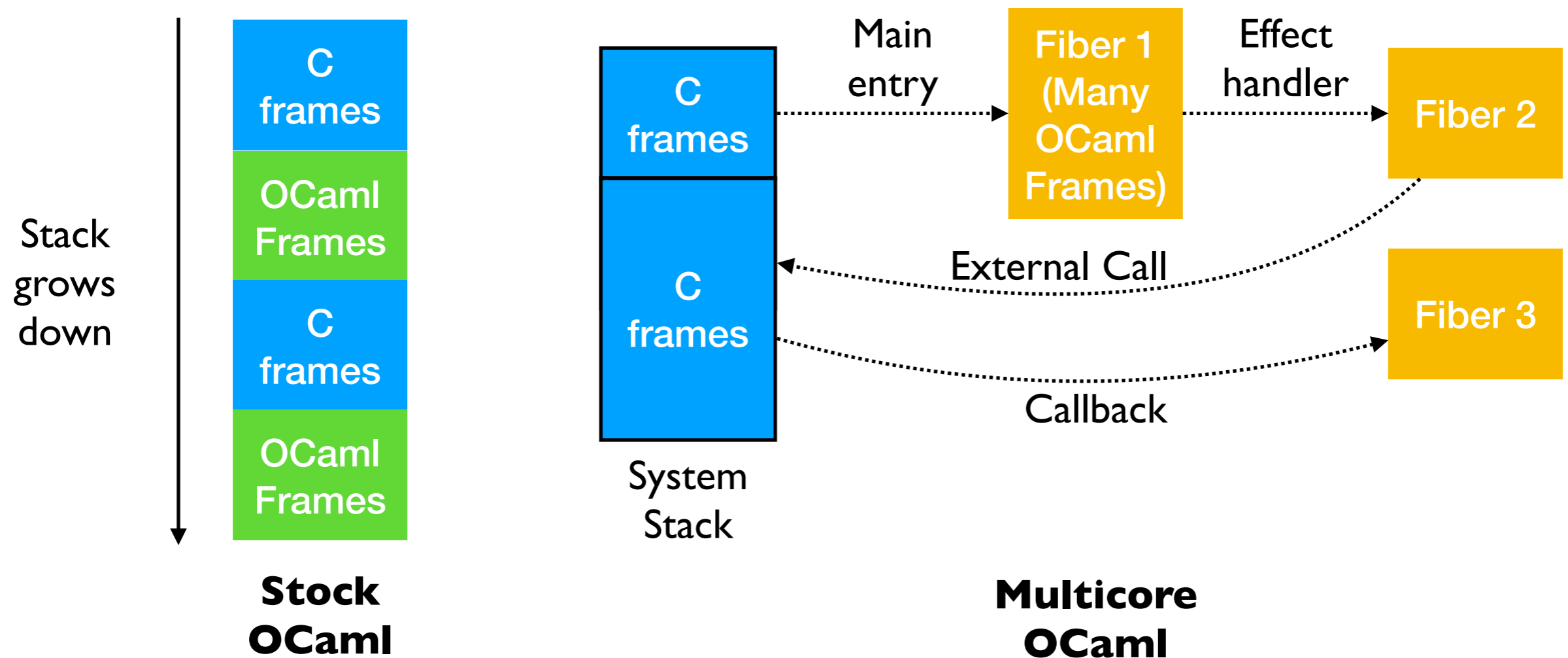
Representing Stacks & Continuations

- 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 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



Switching stacks fast

- *One-shot* — capture and resumption does not involve copying frames

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

Static Semantics

Static Semantics

- No *effect safety*
 - ✦ No static guarantee that all the effects performed are handled (c.f. exceptions)
 - ✦ perform **E** at the top-level raises **Unhandled** exception

Static Semantics

- No *effect safety*
 - ✦ No static guarantee that all the effects performed are handled (c.f. exceptions)
 - ✦ perform `E` at the top-level raises `Unhandled` exception
- Effect system in the works
 - ✦ See also Eff, Koka, Links, Helium
 - ✦ Track both user-defined and built-in (ref, io, exceptions) effects
 - ✦ *OCaml becomes a pure language* (in the Haskell sense — divergence)

Static Semantics

- No *effect safety*
 - ✦ No static guarantee that all the effects performed are handled (c.f. exceptions)
 - ✦ perform *E* at the top-level raises *Unhandled* exception
- Effect system in the works
 - ✦ See also Eff, Koka, Links, Helium
 - ✦ Track both user-defined and built-in (ref, io, exceptions) effects
 - ✦ *OCaml becomes a pure language* (in the Haskell sense — divergence)

```
let foo () = print_string "hello, world"  
val foo : unit -[ io ]-> unit
```

*Syntax &
Semantics in the
works*

Effects without Syntax

- OCaml 5.0 will not feature the syntax presented so far
 - ✦ Do not want a effect handler implementation without effect safety

Effects without Syntax

- OCaml 5.0 will not feature the syntax presented so far
 - ✦ Do not want a effect handler implementation without effect safety
- Expose functions from Obj module to program with effects

Effects without Syntax

- OCaml 5.0 will not feature the syntax presented so far
 - ✦ Do not want a effect handler implementation without effect safety
- Expose functions from Obj module to program with effects

```
effect E : string
```

```
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 "
```

```
effect E : string
```

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

```
let main () =  
  try_with comp ()  
  { effc = fun e ->  
    match e with  
    | E -> Some (fun k ->  
      print_string "1 ";  
      continue k "2 ";  
      print_string "4 ")  
    | e -> None }
```

Performance

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

Performance

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

Instruction Sequence	Significance
a to b	Create a new stack & run the computation
b to c	Performing & handling an effect
c to d	Resuming a continuation
d to e	Returning from a computation & free the stack

- Each of the instruction sequences involves a stack switch

Performance

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

Instruction Sequence	Significance
a to b	Create a new stack & run the computation
b to c	Performing & handling an effect
c to d	Resuming a continuation
d to e	Returning from a computation & free the stack

- Each of the instruction sequences involves a stack switch
- Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
 - ✦ Cost measured using Intel PT's cycle accurate tracing
 - ✦ For calibration, memory read latency is **90 ns** (local NUMA node) and **145 ns** (remote NUMA node)

Performance

```
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
 - ✦ Cost measured using Intel PT's cycle accurate tracing
 - ✦ For calibration, memory read latency is **90 ns** (local NUMA node) and **145 ns** (remote NUMA node)

Performance: Generators

- Traverse a complete binary-tree of depth 25
 - ♦ 2^{26} stack switches

Performance: Generators

- Traverse a complete binary-tree of depth 25
 - ✦ 2^{26} stack switches
- *Iterator* — idiomatic recursive traversal

Performance: Generators

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

Performance: Generators

Multicore OCaml

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

Performance: Generators

Multicore OCaml

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)

Performance: WebServer

- Effect handlers for asynchronous I/O in direct-style
 - ✦ <https://github.com/kayceesrk/ocaml-aeio>
- Variants
 - ✦ **Go** + net/http (GOMAXPROCS=1)
 - ✦ OCaml + http/af + **Lwt** (explicit callbacks)
 - ✦ OCaml + http/af + Effect handlers (**MC**)
- Performance measured using wrk2

Performance: WebServer

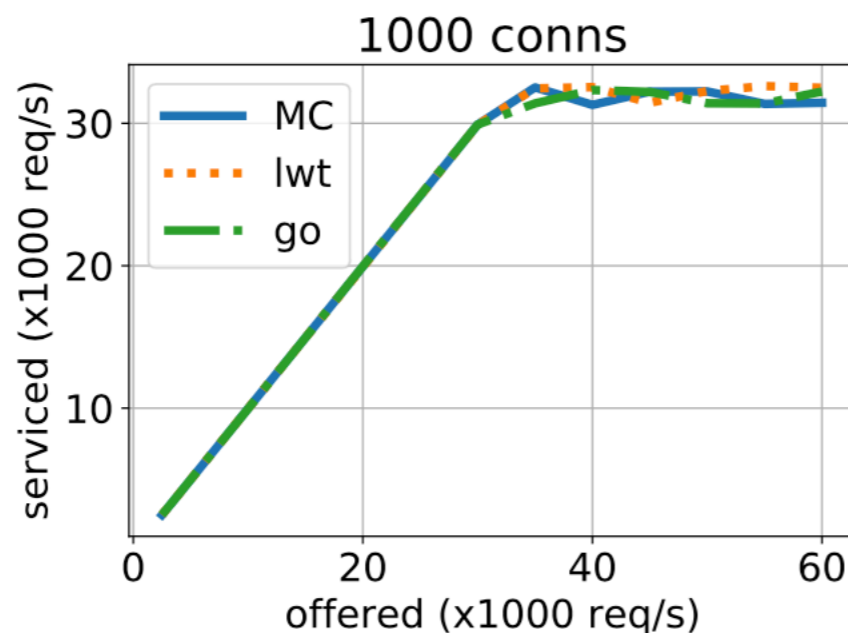
- Effect handlers for asynchronous I/O in direct-style

♦ <https://github.com/kayceesrk/ocaml-aeio>

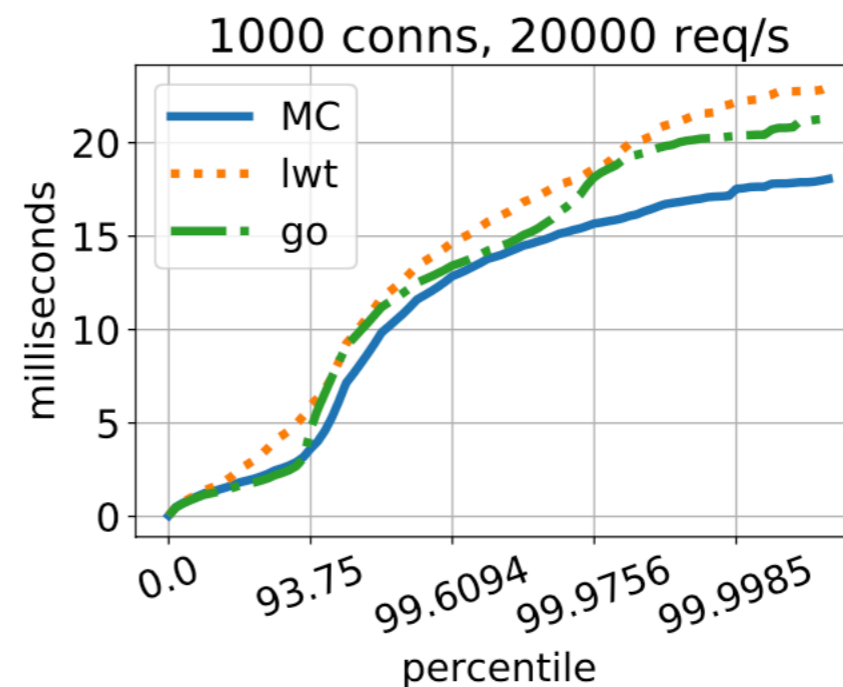
- Variants

- ♦ **Go** + net/http (GOMAXPROCS=1)
- ♦ OCaml + http/af + **Lwt** (explicit callbacks)
- ♦ OCaml + http/af + Effect handlers (**MC**)

- Performance measured using wrk2



(a) Throughput



(b) Tail latency

Performance: WebServer

- Effect handlers for asynchronous I/O in direct-style

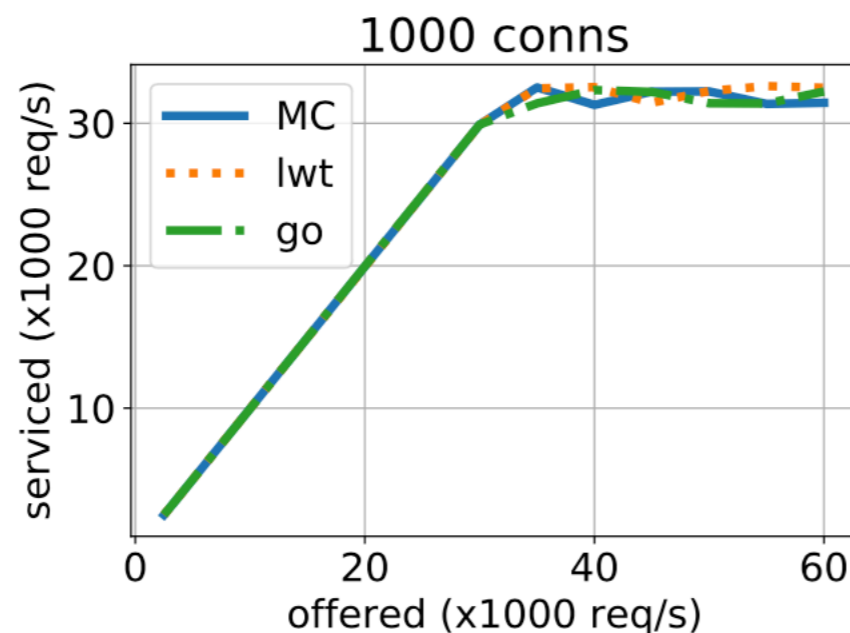
♦ <https://github.com/kayceesrk/ocaml-aeio>

- Variants

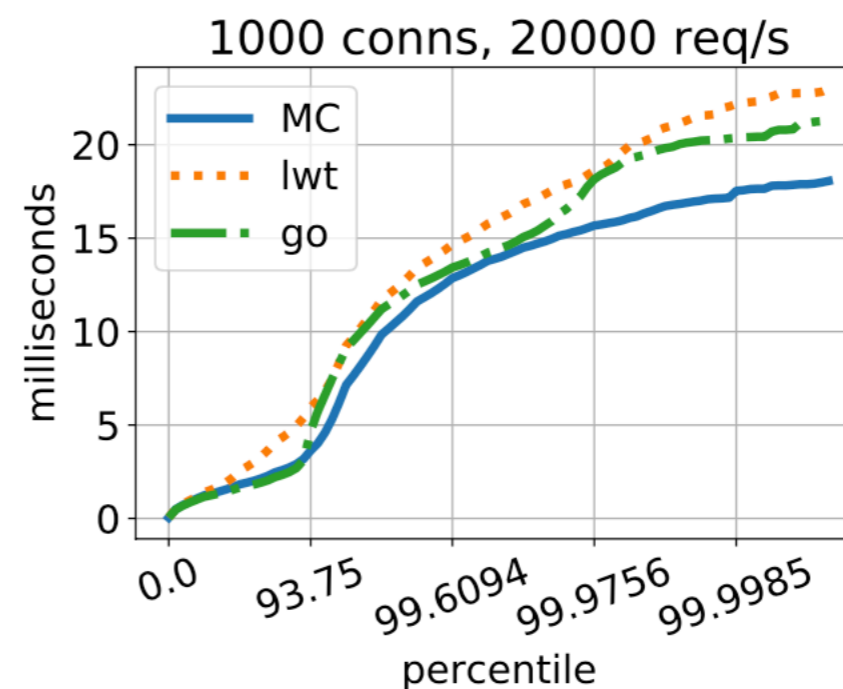
- ♦ **Go** + net/http (GOMAXPROCS=1)
- ♦ OCaml + http/af + **Lwt** (explicit callbacks)
- ♦ OCaml + http/af + Effect handlers (**MC**)

- Direct style (no monadic syntax)
- Can use OCaml exceptions!
- Backtrace per thread (request)
- gdb & perf work!

- Performance measured using wrk2



(a) Throughput



(b) Tail latency

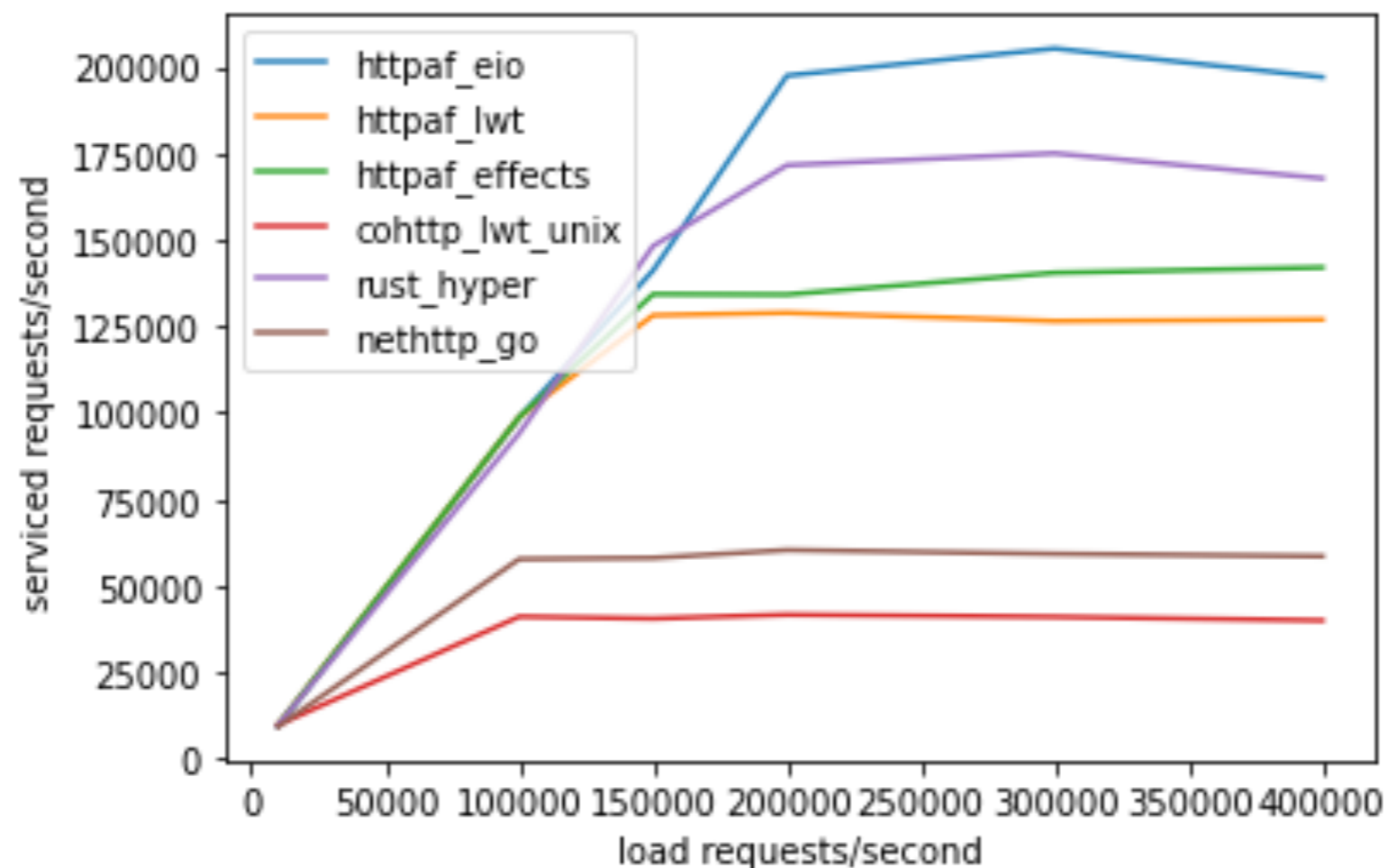
Performance: WebServer

Performance: WebServer

- **eio**: effects-based direct-style I/O
 - ✦ Multiple backends — Linux *io_uring*, *epoll*, MacOS *GCD*, Windows *IOCP*, FreeBSD *kqueue*

Performance: WebServer

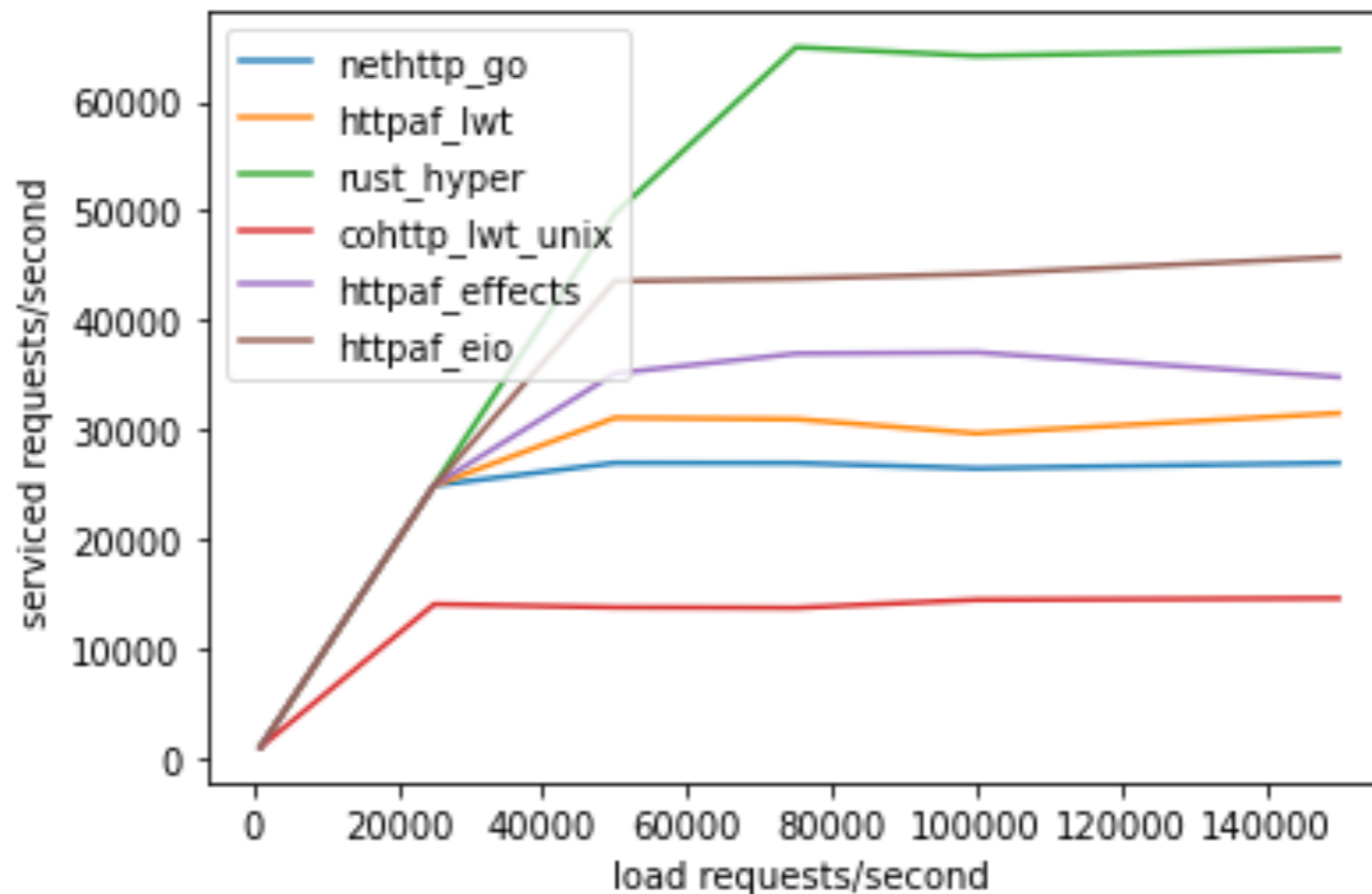
- **eio**: effects-based direct-style I/O
 - ✦ Multiple backends — Linux *io_uring*, *epoll*, MacOS *GCD*, Windows *IOCP*, FreeBSD *kqueue*



100 open connections, 60 seconds w/ io_uring

Performance: WebServer

- **eio**: effects-based direct-style I/O
 - ✦ Multiple backends — Linux *io_uring*, *epoll*, MacOS *GCD*, Windows *IOCP*, FreeBSD *kqueue*



1k open connections, 60 seconds, w/ io_uring

Backwards Compatibility

- OCaml is a systems programming language
 - ✦ Manipulates resources such as files, sockets, buffers, etc.

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

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

```
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
```

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

raises
End_of_file at
the end

```
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
```

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

```
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
```

raises
End_of_file at
the end

raise Sys_error
when channel is
closed

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

```
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
```

raises
End_of_file at
the end

raise Sys_error
when channel is
closed

We would like to make this code transparently asynchronous

Asynchronous IO

```
effect In_line : in_channel -> string  
effect Out_str : out_channel * string -> unit
```

Asynchronous IO

```
effect In_line : in_channel -> string
effect Out_str : out_channel * string -> unit

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

Asynchronous IO

```
effect In_line : in_channel -> string
effect Out_str : out_channel * string -> unit

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

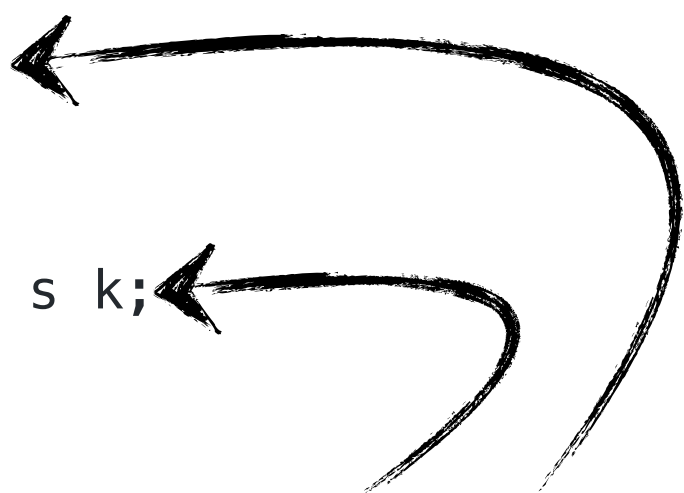
let run_aio f = match f () with
| v -> v
| effect (In_line chan) k ->
    register_async_input_line chan k;
    run_next ()
| effect (Out_str (chan, s)) k ->
    register_async_output_string chan s k;
    run_next ()
```

Asynchronous IO

```
effect In_line : in_channel -> string
effect Out_str : out_channel * string -> unit
```

```
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 (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 ()
```



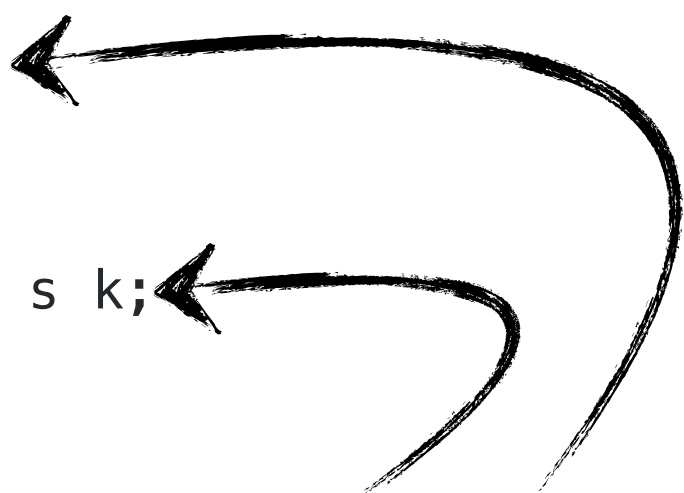
- Continue with appropriate *value* when the asynchronous IO call returns

Asynchronous IO

```
effect In_line : in_channel -> string
effect Out_str : out_channel * string -> unit
```

```
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 (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

Linearity

- Resources such as sockets, file descriptors, channels and buffers are *linear* resources
 - ✦ Created and destroyed *exactly once*

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

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 not resumed
 - ✦ This breaks resource-safe legacy code

Linearity

```
effect E : unit  
let foo () = perform E
```

Linearity

```
effect E : unit
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
```

Linearity

```
effect E : unit
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 *)
```

Linearity

```
effect E : unit
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 debugging and profiling tools — gdb, lldb, perf, libunwind, etc.
 - ✦ DWARF stack unwinding support

Backtraces

- OCaml has excellent compatibility with debugging and profiling tools — gdb, lldb, perf, libunwind, etc.
 - ✦ DWARF stack unwinding support
- *Multicore OCaml supports DWARF stack unwinding across fibers*

Backtraces

- OCaml has excellent compatibility with debugging and profiling tools — gdb, lldb, perf, libunwind, etc.
 - ✦ DWARF stack unwinding support
- *Multicore OCaml supports DWARF stack unwinding across fibers*

```
effect E : unit
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 _ -> () (* leak *)
```

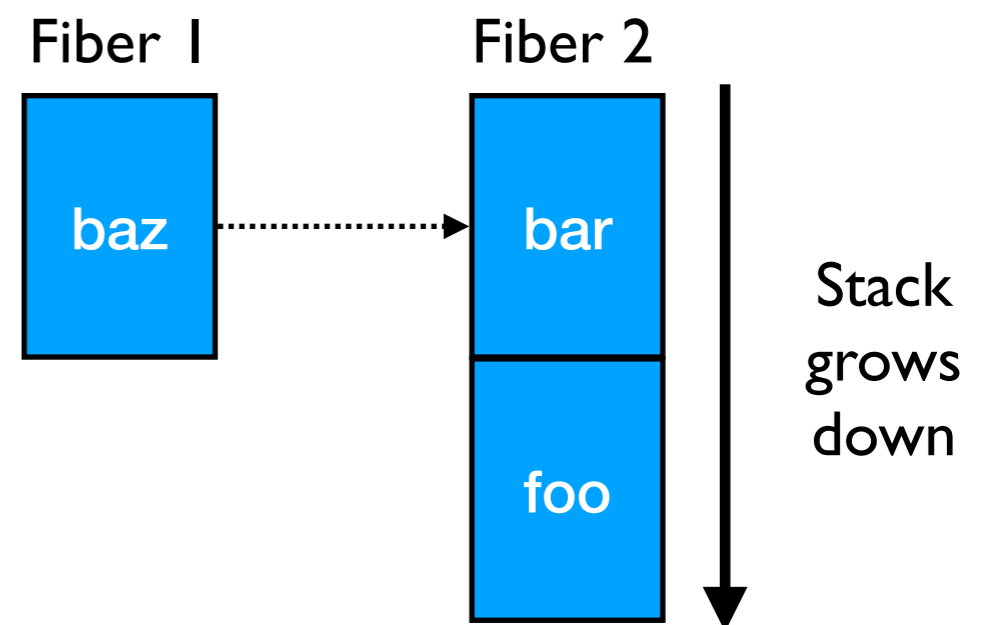
Backtraces

- OCaml has excellent compatibility with debugging and profiling tools — gdb, lldb, perf, libunwind, etc.
 - ✦ DWARF stack unwinding support
- *Multicore OCaml supports DWARF stack unwinding across fibers*

```
effect E : unit
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 _ -> () (* leak *)
```



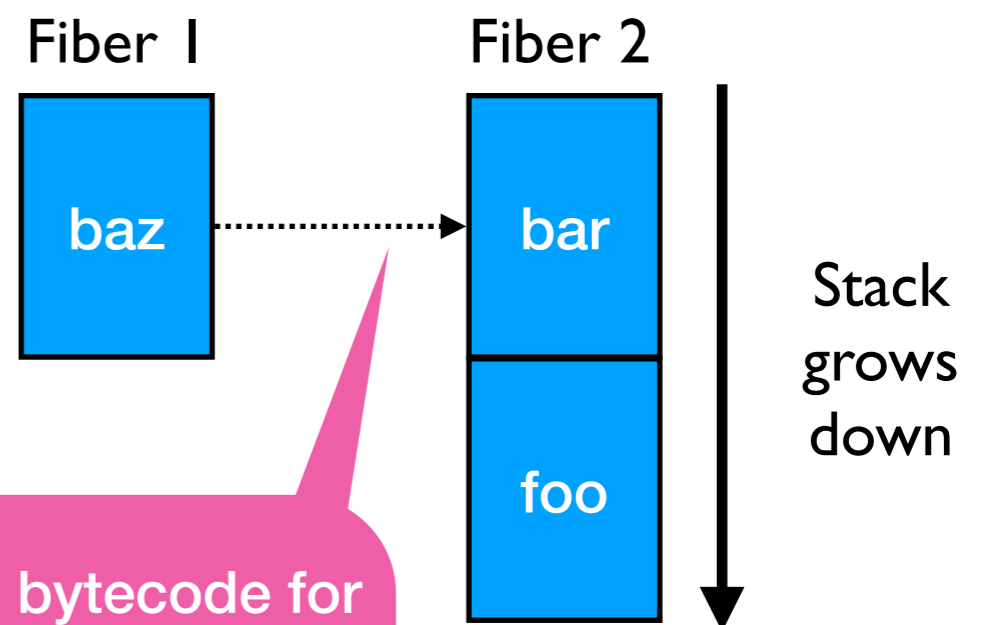
Backtraces

- OCaml has excellent compatibility with debugging and profiling tools — gdb, lldb, perf, libunwind, etc.
 - ✦ DWARF stack unwinding support
- *Multicore OCaml supports DWARF stack unwinding across fibers*

```
effect E : unit
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 _ -> () (* leak *)
```



Bespoke DWARF bytecode for unwinding across fibers

Backtraces

```
effect E : unit
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 _ -> () (* leak *)
```

```
(lldb) bt
* thread #1, name = 'a.out', stop reason = ...
  * #0: 0x58b208 caml_perform
    #1: 0x56aa5d camlTest__foo_83 at test.ml:4
    #2: 0x56aae2 camlTest__bar_85 at test.ml:9
    #3: 0x56a9fc camlTest__fun_199 at test.ml:14
    #4: 0x58b322 caml_runstack + 70
    #5: 0x56ab99 camlTest__baz_91 at test.ml:14
    #6: 0x56ace6 camlTest__entry at test.ml:21
    #7: 0x56a41c caml_program + 60
    #8: 0x58b0b7 caml_start_program + 135
    #9: ...
```

Summary

- Multicore OCaml brings *effect handlers* for writing high-performance concurrent programs
 - ✦ Removes the dichotomy between synchronous and asynchronous code
 - ✦ Better than baking in lightweight threads in the language

Summary

- Multicore OCaml brings *effect handlers* for writing high-performance concurrent programs
 - ✦ Removes the dichotomy between synchronous and asynchronous code
 - ✦ Better than baking in lightweight threads in the language
- Effects Examples
 - ✦ <https://github.com/ocaml-multicore/effects-examples>

Summary

- Multicore OCaml brings *effect handlers* for writing high-performance concurrent programs
 - ✦ Removes the dichotomy between synchronous and asynchronous code
 - ✦ Better than baking in lightweight threads in the language
- Effects Examples
 - ✦ <https://github.com/ocaml-multicore/effects-examples>
- Sivaramakrishnan et al, “Retrofitting Effect Handlers onto OCaml”, PLDI 2021

Nothing to see here...

Fiber Layout

```
match f () with
| v -> ...
| exception X1 -> ...
| exception (X2 v) -> ...
| effect E1 k -> ...
| effect E2 k -> ...
```

