Effect Handlers in Multicore OCaml

Daniel Hillerström, Daan Leijen, Sam Lindley, Matija Pretnar, Andreas Rossberg, **KC Sivaramakrishnan**

Effect Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*
  - Concurrency expressed through *effect handlers*
  - Will land upstream in Q2 2021

Effect Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*
  - ✦ Concurrency expressed through *effect handlers*
  - ✦ Will land upstream in Q2 2021

```
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 Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*
  - ✦ Concurrency expressed through *effect handlers*
  - ✦ Will land upstream in Q2 2021

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 "
```

Effect Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*
  - ✦ Concurrency expressed through *effect handlers*
  - ✦ Will land upstream in Q2 2021

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 "
```

computation

Effect Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*
  - ✦ Concurrency expressed through *effect handlers*
  - ✦ Will land upstream in Q2 2021

```
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 declaration

computation

handler

Effect Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*
  - ✦ Concurrency expressed through *effect handlers*
  - ✦ Will land upstream in Q2 2021

```
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 declaration

suspends current computation

computation

handler

Effect Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*
  - ✦ Concurrency expressed through *effect handlers*
  - ✦ Will land upstream in Q2 2021

```
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 declaration

suspends current computation

computation

delimited continuation

handler

Effect Handlers

- Multicore OCaml is an OCaml extension with native support for *concurrency* and shared-memory *parallelism*
  - ✦ Concurrency expressed through *effect handlers*
  - ✦ Will land upstream in Q2 2021

```
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 declaration

suspends current computation

computation

delimited continuation

handler

resume suspended computation

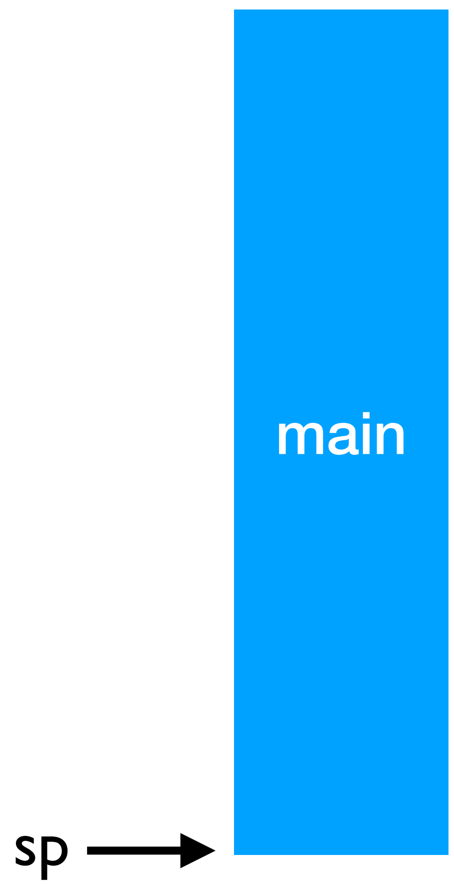
Compilation

```
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



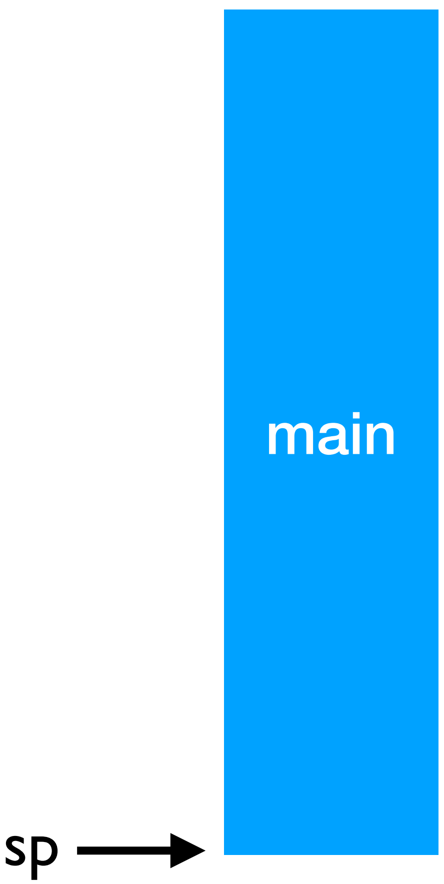
Compilation

```
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

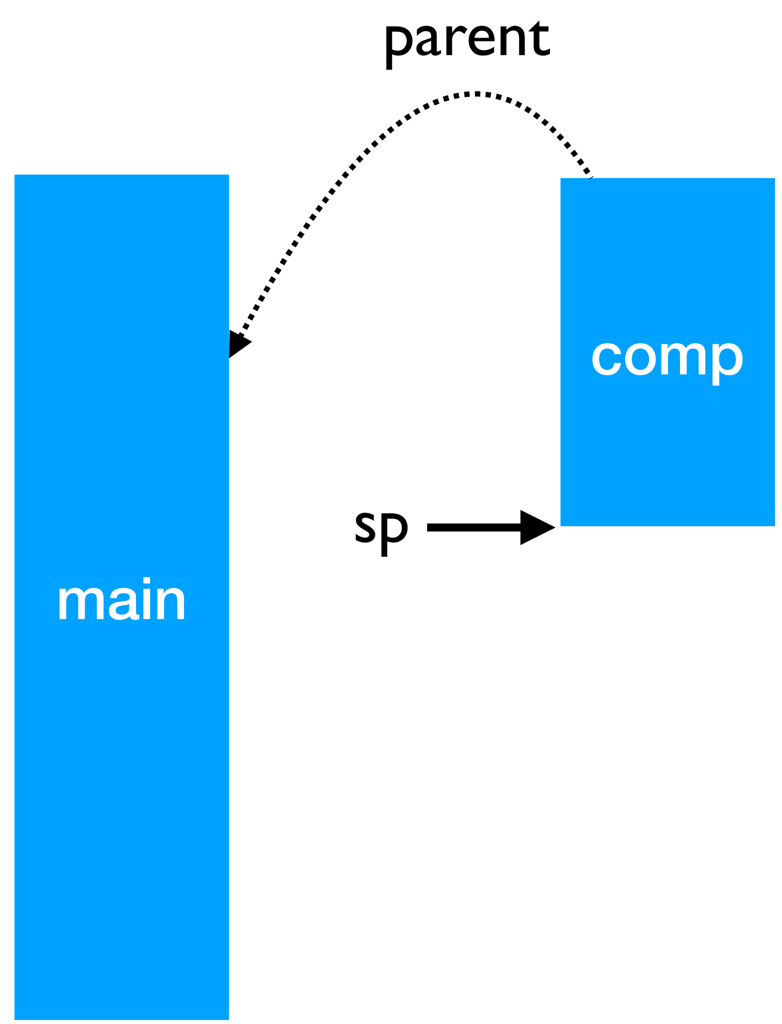


Compilation

```
effect E : string

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

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

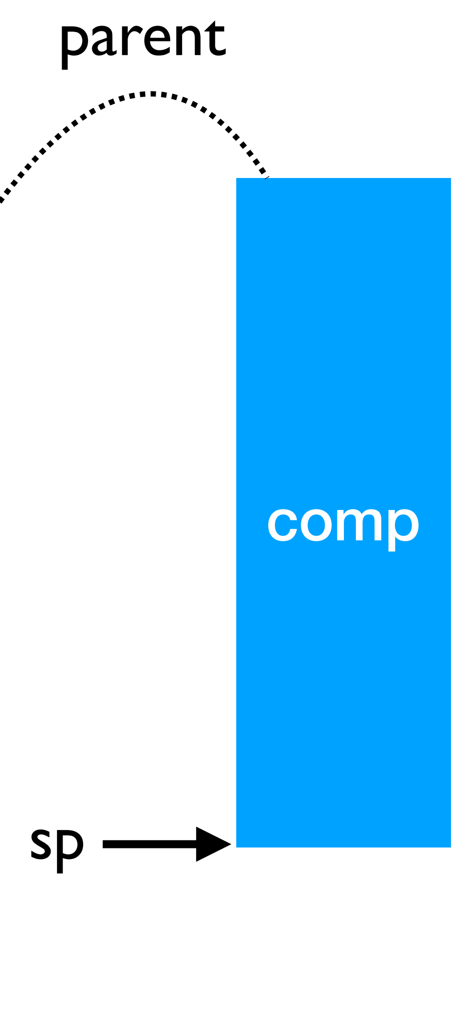


Compilation

```
    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 "
```



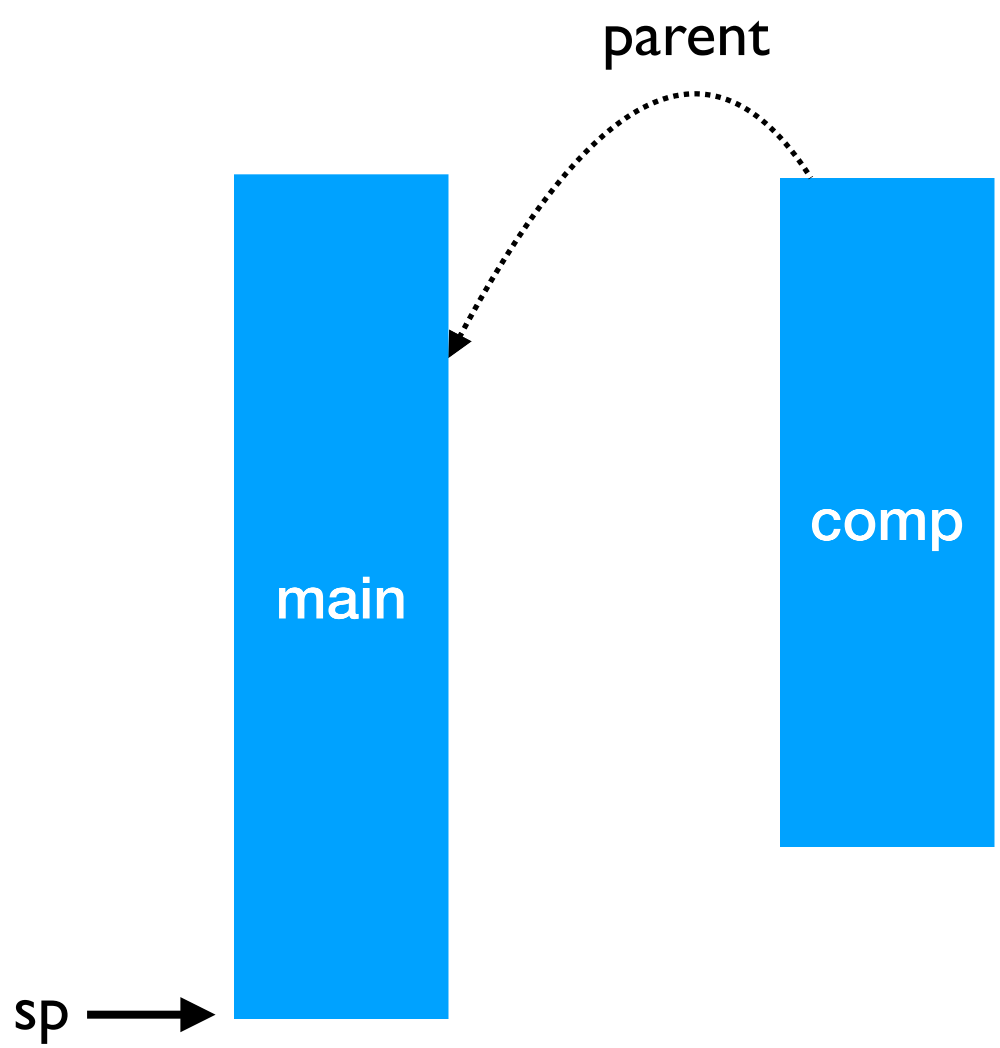
0

Compilation

```
     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 "
```



0

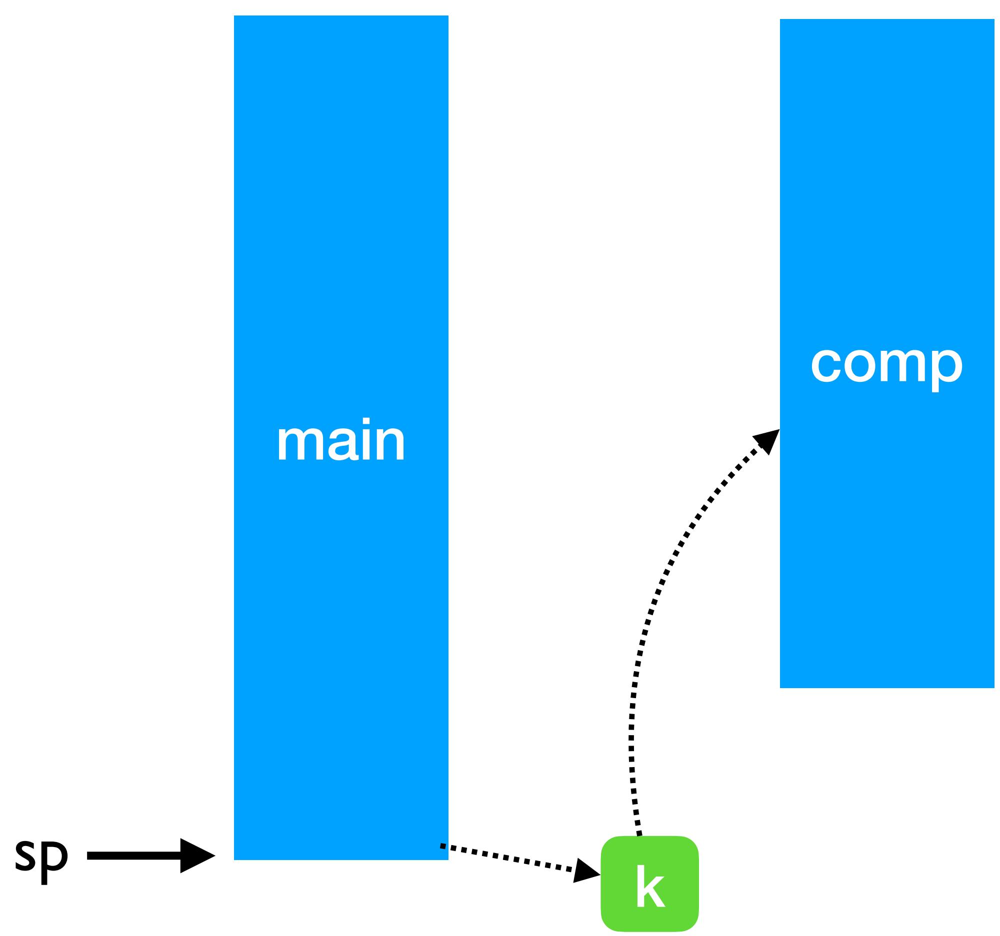
Compilation

```
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

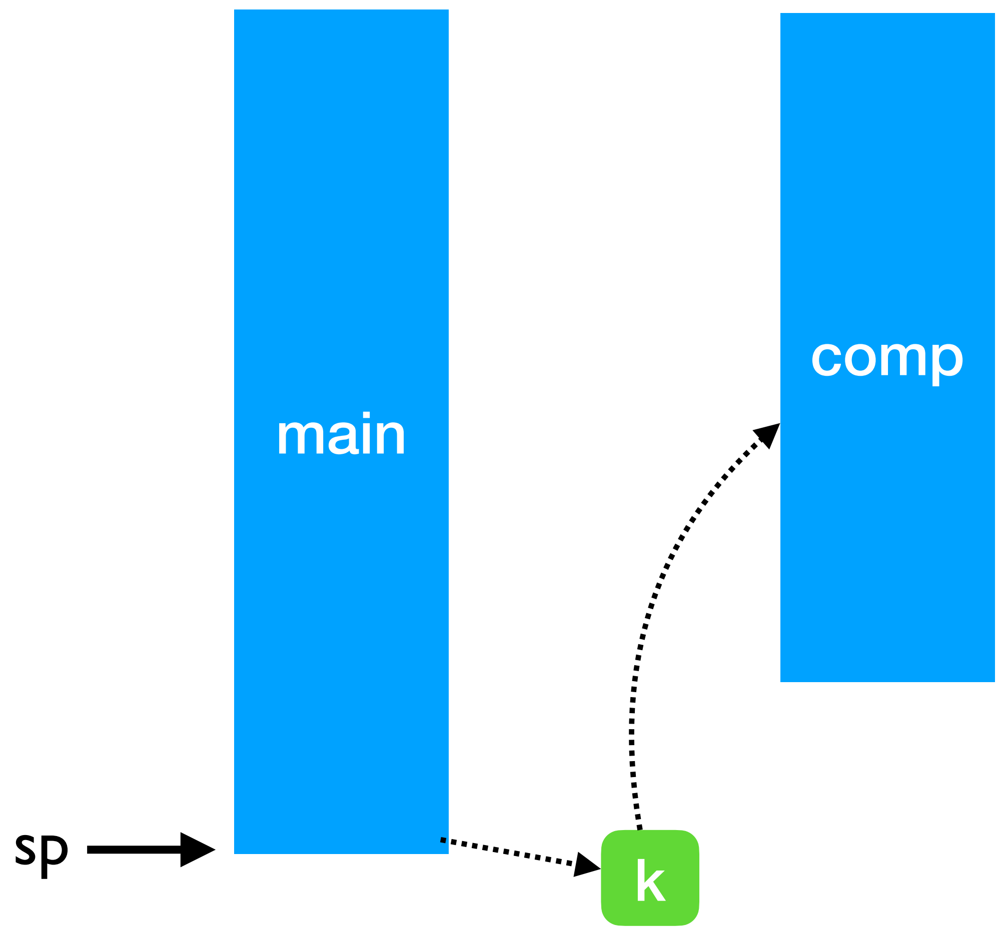
Compilation

```
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

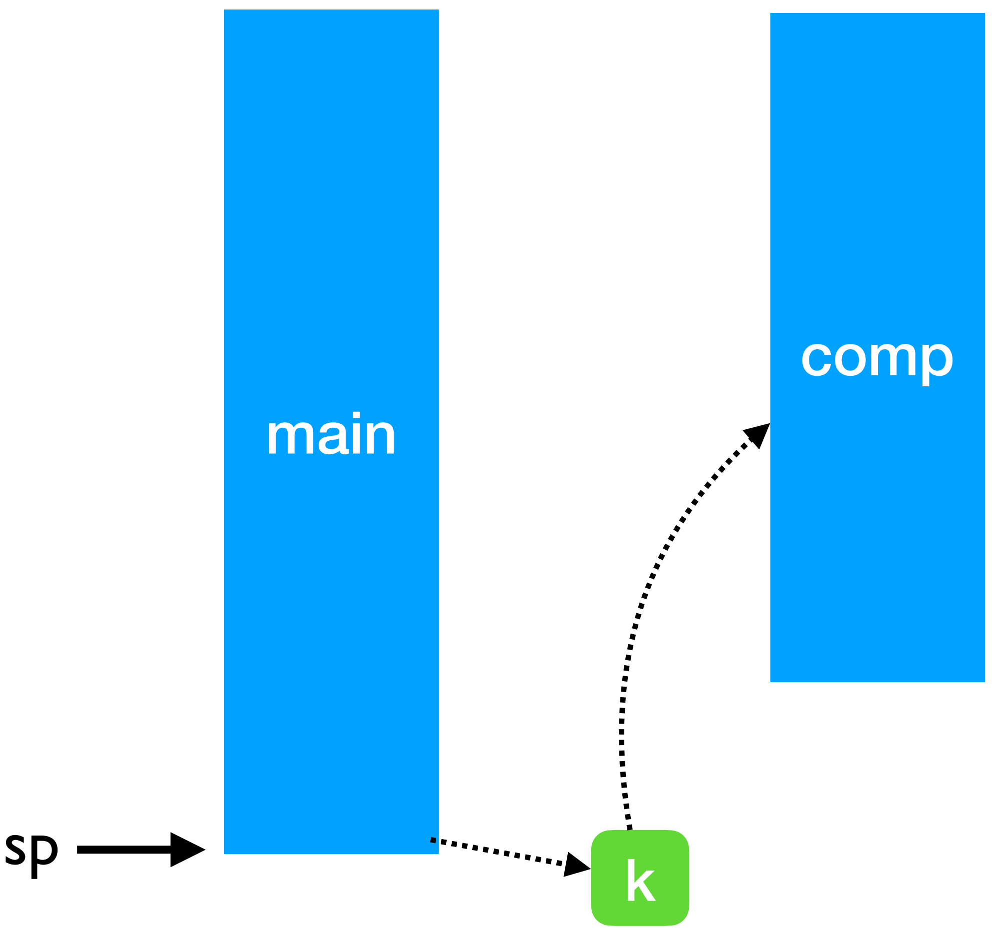
Compilation

```
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

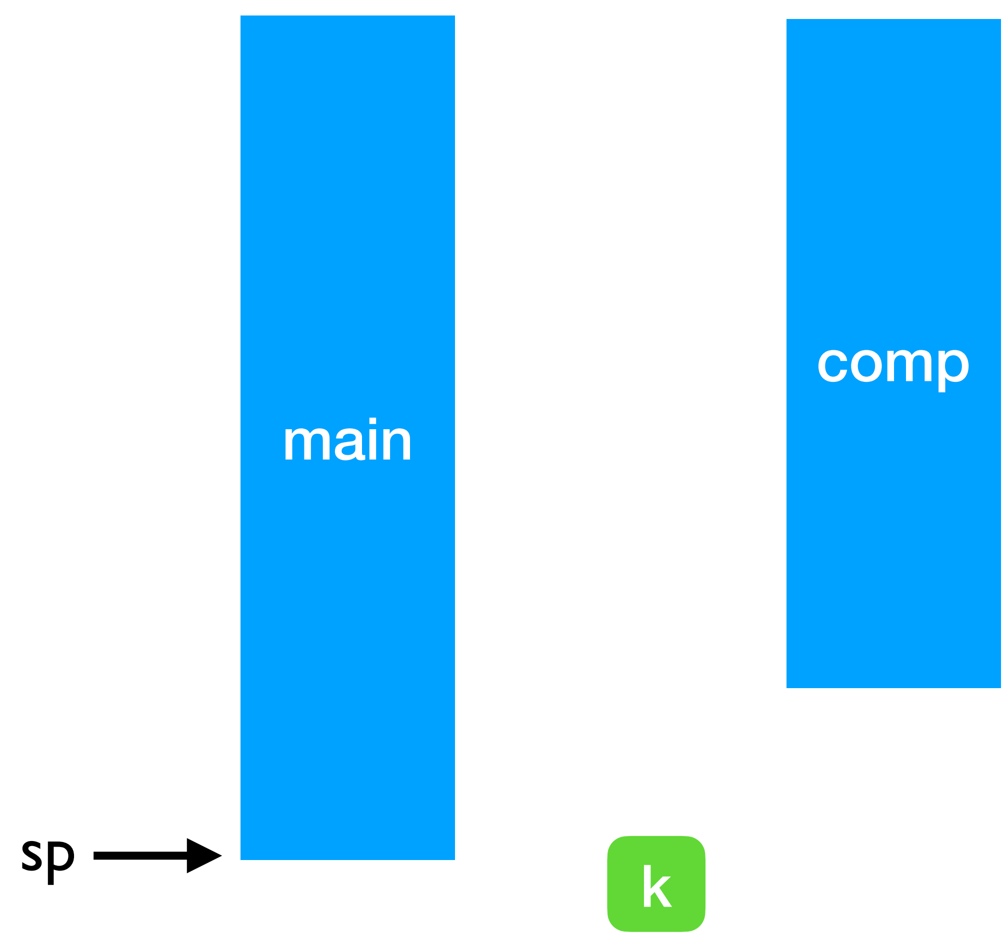
Compilation

```
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

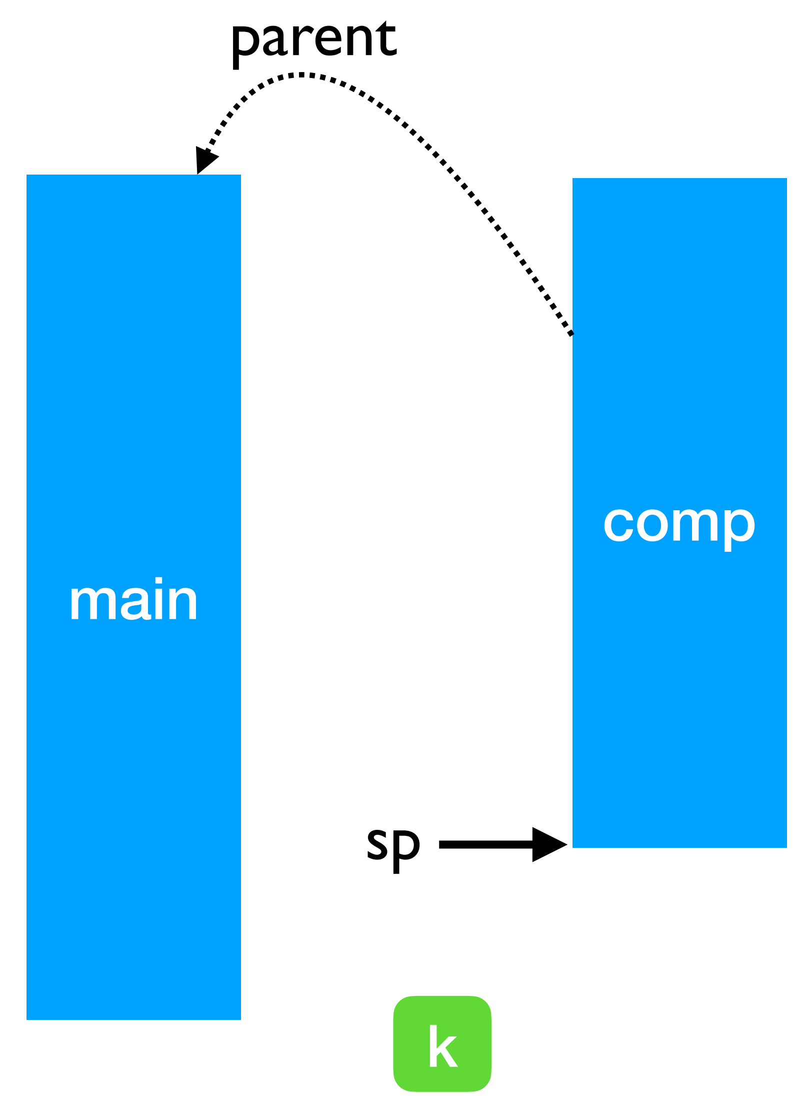
Compilation

```
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

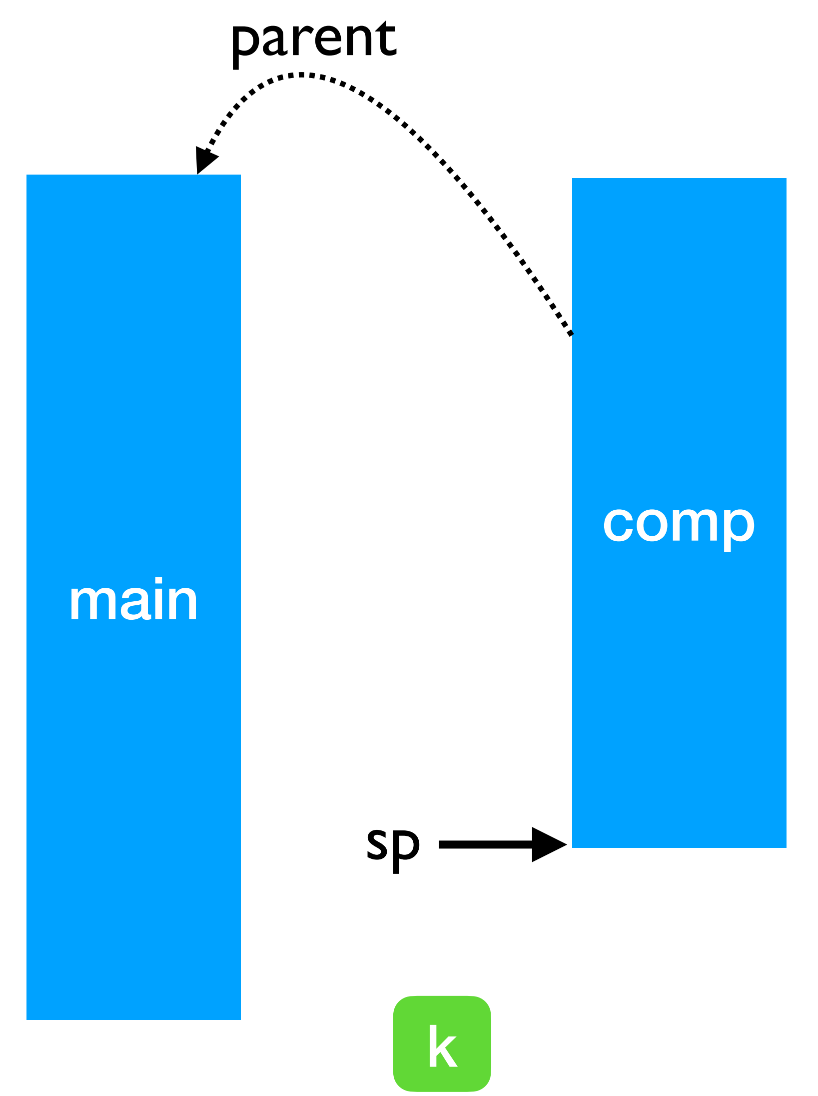
Compilation

pc

```
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 "
```



0 1 2

Compilation

```
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

Compilation

```
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

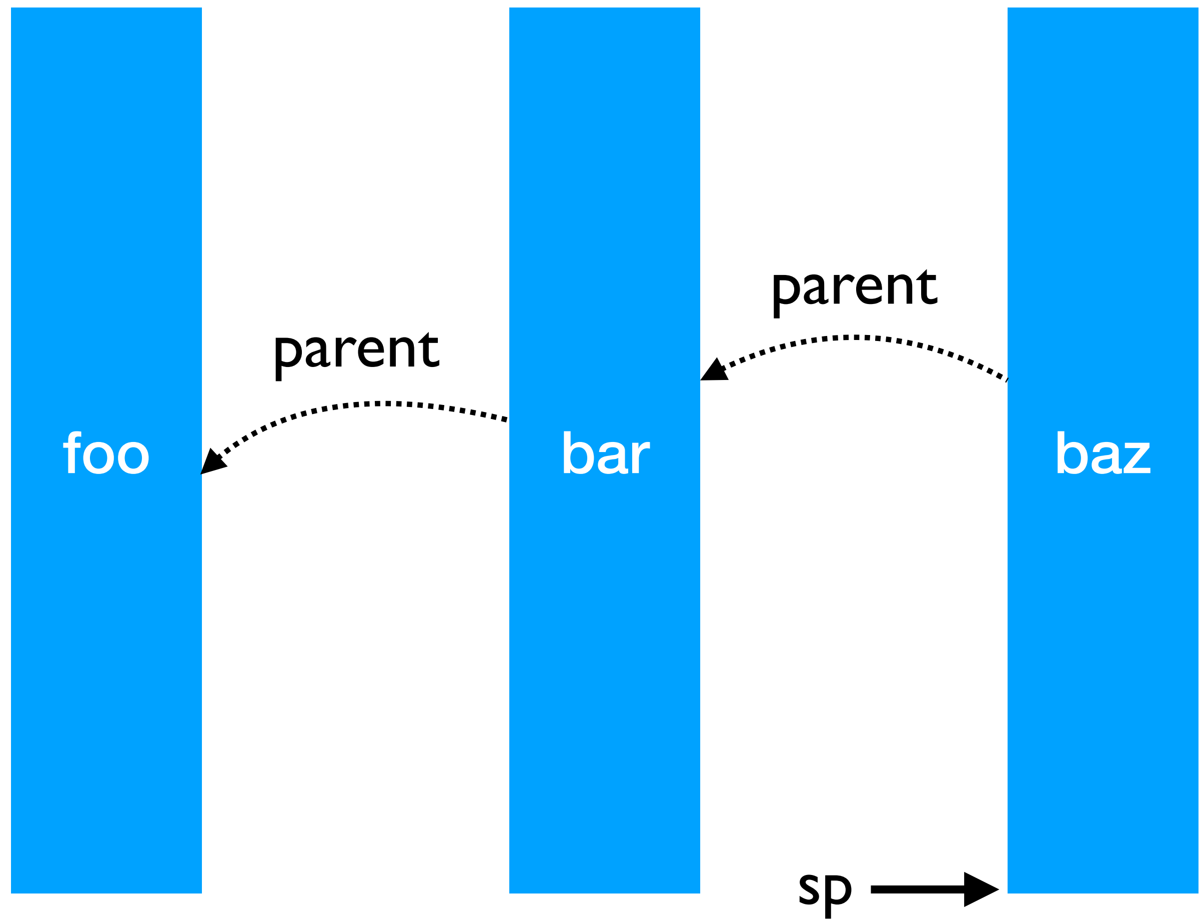
Handlers can be nested

```
effect A : unit
effect B : unit

let baz () =
  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

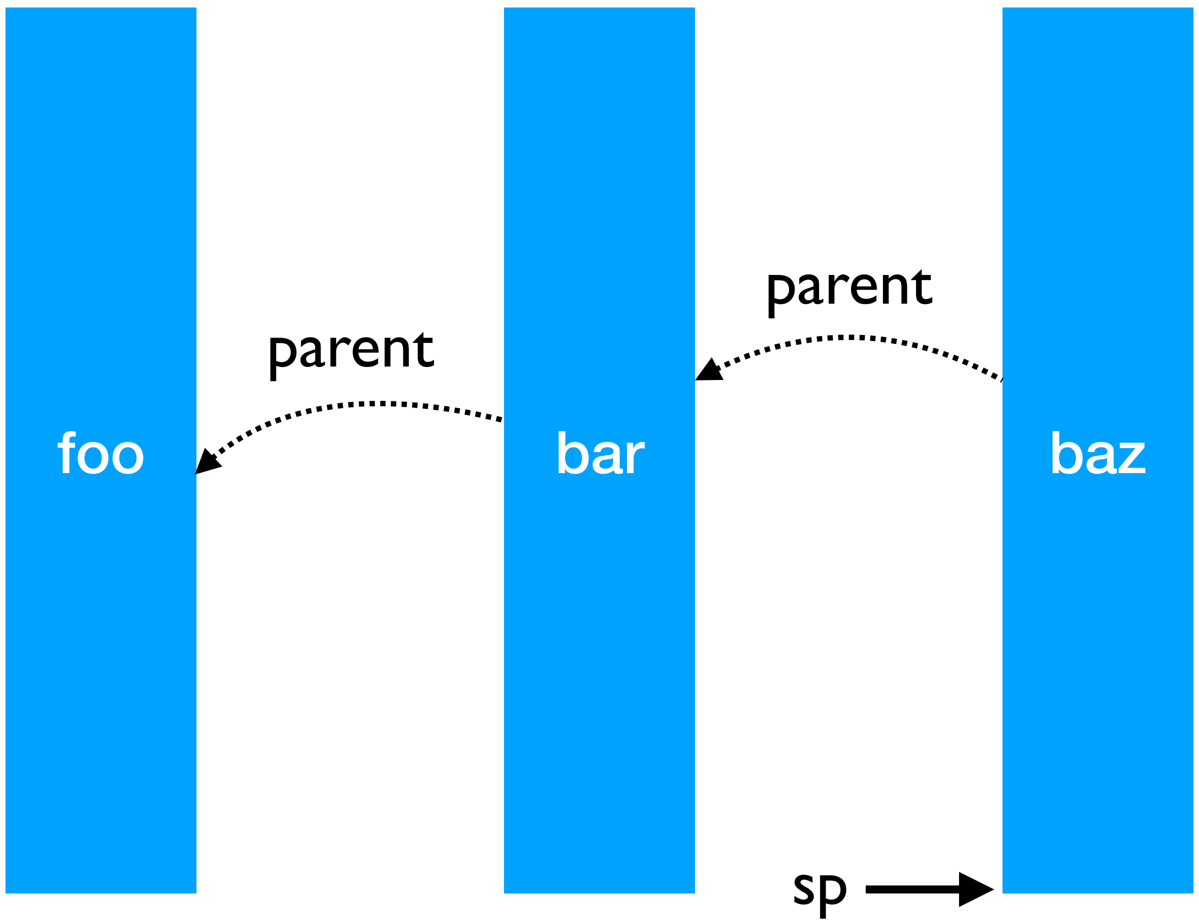
```
effect A : unit
effect B : unit

let baz () =
  perform A

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

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

pc



Handlers can be nested

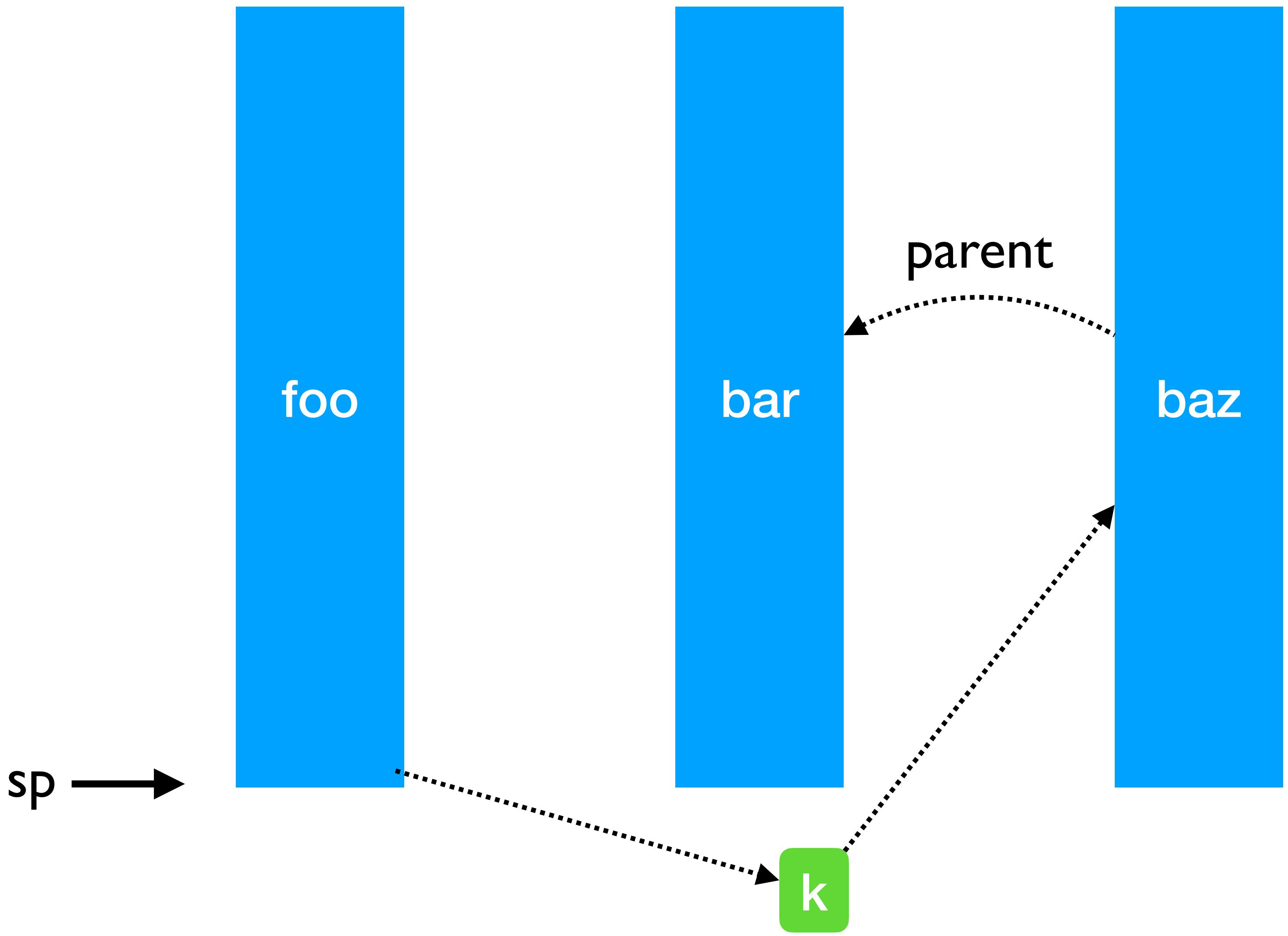
```
effect A : unit
effect B : unit

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



Handlers can be nested

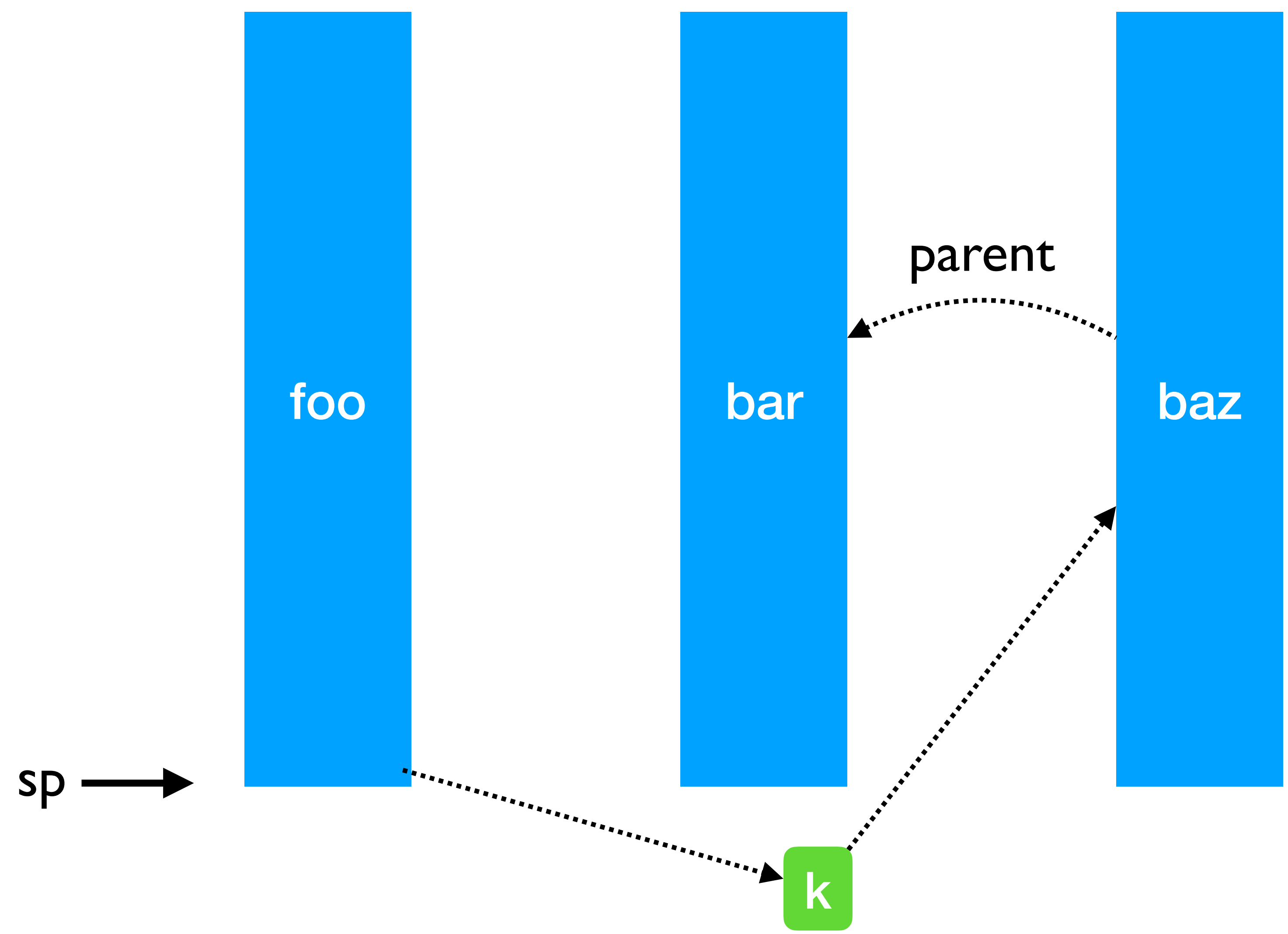
```
effect A : unit
effect B : unit

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



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

Deep-dive into `perform`

Deep-dive into `perform`

- Full power of pattern matching for matching effects
  - Tag test + branching is compiled to a function

Deep-dive into `perform`

- Full power of pattern matching for matching effects
  - ✦ Tag test + branching is compiled to a function

https://github.com/ocaml-multicore/ocaml-multicore/blob/parallel_minor_gc/runtime/amd64.S#L865

Performance

- 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)

Performance

- 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)

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

Performance

- 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)

```
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

- 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)

```
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	2479
b to c	Performing & handling an effect	122
c to d	Resuming a continuation	189
d to e	Returning from a computation & free the stack	155

- Each of the instruction sequences involves a stack switch

Performance: Generators

Performance: Generators

- Traverse a complete binary-tree of depth 25

Performance: Generators

- Traverse a complete binary-tree of depth 25
- Iterator — idiomatic recursive traversal

Performance: Generators

- Traverse a complete binary-tree of depth 25
- Iterator — idiomatic recursive traversal
- Generator — `next()` function to consume elements on-demand
  - Hand-written generator (hw-generator)
    - CPS translation + defunctionalization to remove intermediate closure allocation
  - Generator using effect handlers (eh-generator)
    - $2 * (2^{25} - 1) + 2 = 2^{26}$ stack switches

Performance: Generators

- Traverse a complete binary-tree of depth 25
- Iterator — idiomatic recursive traversal
- Generator — `next()` function to consume elements on-demand
  - ✦ Hand-written generator (hw-generator)
    - ✤ CPS translation + defunctionalization to remove intermediate closure allocation
  - ✦ Generator using effect handlers (eh-generator)
    - ✤ $2 * (2^{25} - 1) + 2 = 2^{26}$ stack switches

Multicore OCaml

Variant	Time (milliseconds)
Iterator (baseline)	202
hw-generator	761 (**3.76x**)
eh-generator	1879 (**9.30x**)

Performance: Generators

- Traverse a complete binary-tree of depth 25
- Iterator — idiomatic recursive traversal
- Generator — `next()` function to consume elements on-demand
    - Hand-written generator (hw-generator)
        - CPS translation + defunctionalization to remove intermediate closure allocation
    - Generator using effect handlers (eh-generator)
        - $2 * (2^{25} - 1) + 2 = 2^{26}$ stack switches

Multicore OCaml

Variant	Time (milliseconds)
Iterator (baseline)	202
hw-generator	761 (**3.76x**)
eh-generator	1879 (**9.30x**)

nodejs 14.07

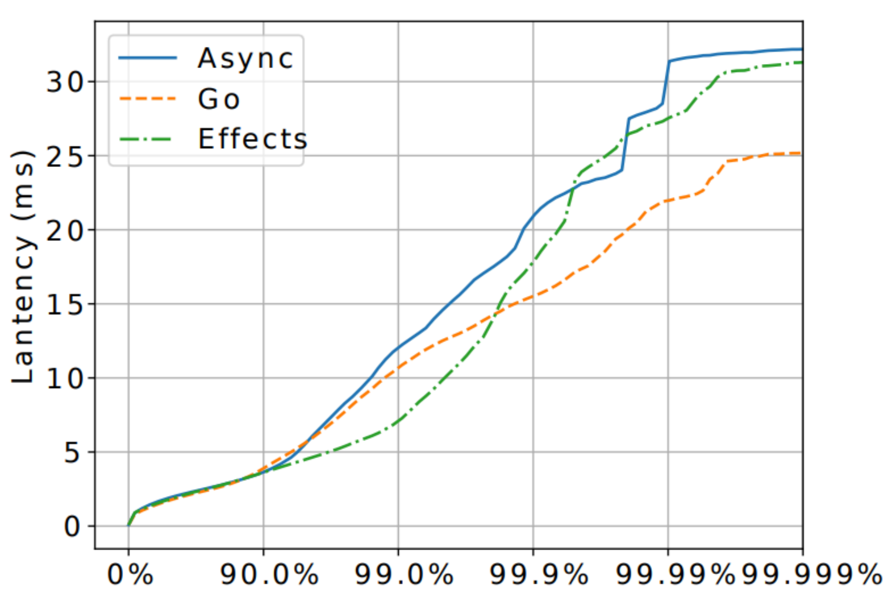
Variant	Time (milliseconds)
Iterator (baseline)	492
generator	43842 (**89.1x**)

Performance: WebServer

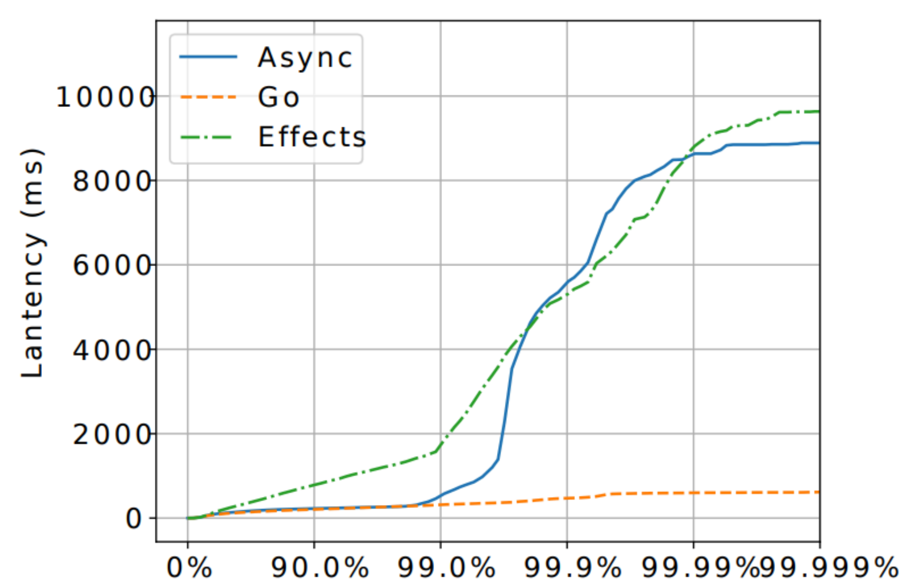
- Effect handlers for asynchronous I/O
- Variants
  - **Go** + net/http
  - OCaml + http/af + **Async** (explicit callbacks)
  - OCaml + http/af + **Effect** handlers
- Latency measured using wrk2

Performance: WebServer

- Effect handlers for asynchronous I/O
- Variants
  - ✦ **Go** + net/http
  - ✦ OCaml + http/af + **Async** (explicit callbacks)
  - ✦ OCaml + http/af + **Effect** handlers
- Latency measured using wrk2



(a) Medium contention: 1k connections, 10k requests/sec



(b) High contention: 10k connections, 30k requests/sec

Thank you!

- Multicore OCaml
  - https://github.com/ocaml-multicore/ocaml-multicore
- A collection of effect handlers examples
  - https://github.com/ocaml-multicore/effects-examples
- JS generator example
  - https://github.com/kayceesrk/wasmfx/tree/master/cg_4_aug_20