Effective Concurrency with Algebraic Effects

Stephen Dolan[1], Leo White[2], **KC Sivaramakrishnan**[1], Jeremy Yallop[1], Anil Madhavapeddy[1]

1 UNIVERSITY OF CAMBRIDGE

2 Jane Street

Concurrency ≠ Parallelism

- Concurrency
  - Programming technique
  - **Overlapped** execution of processes
- Parallelism
  - Performance hack
  - **Simultaneous** execution of computations

Concurrency ≠ Parallelism

- Concurrency
  - Programming technique
  - **Overlapped** execution of processes
- Parallelism
  - Performance hack
  - **Simultaneous** execution of computations

Concurrency ∩ Parallelism ➔ *Scalable Concurrency*

Concurrency ≠ Parallelism

- Concurrency
  - Programming technique
  - **Overlapped** execution of processes
- Parallelism
  - Performance hack
  - **Simultaneous** execution of computations

Concurrency ∩ Parallelism ➔ *Scalable Concurrency*
(Fibers) (Domains)

Schedulers

- Multiplexing fibers over domain(s)
  - Bake scheduler into the runtime system (GHC)

Schedulers

- Multiplexing fibers over domain(s)
  - Bake scheduler into the runtime system (GHC)
- Allow programmers to describe schedulers!
  - Parallel search —> LIFO work-stealing
  - Web-server —> FIFO runqueue
  - Data parallel —> Gang scheduling

Schedulers

- Multiplexing fibers over domain(s)
  - Bake scheduler into the runtime system (GHC)
- Allow programmers to describe schedulers!
  - Parallel search —> LIFO work-stealing
  - Web-server —> FIFO runqueue
  - Data parallel —> Gang scheduling
- ***Algebraic Effects and Handlers***

Algebraic Effects and Handlers

- Programming and reasoning about computational effects in a pure setting.
  - Cf. Monads
- Effects in practice
  - M Pretnar, A Bauer, “Eff programming language”
    - http://www.eff-lang.org/
  - O Kiselyov, A Sabry, C Swords, B Foppa, “Extensible-effects for Haskell”
    - https://hackage.haskell.org/package/extensible-effects
  - E Brady, “Effects in Idris”
    - http://eb.host.cs.st-andrews.ac.uk/drafts/eff-tutorial.pdf
  - O Kammar, S Lindley, N Oury , “Handlers in Action”, ICFP ’13
    - dl.acm.org/citation.cfm?id=2500590

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

val r : int = 4

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

val r : int = 4

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3))

let r =
  try
    f () (* spawned in a new fiber *)
  with effect (Foo i) k ->
         continue k (i + 1)
```

val r : int = 5

Effects interface

```
type _ eff += Foo : int -> int eff

val perform  : 'a eff -> 'a

type ('a,'b) continuation
val continue : ('a,'b) continuation -> 'a -> 'b
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

val r : int = 4

```
effect Foo : int -> int

let f () = 1 + (perform (Foo 3))

let r =
  try
    f () (* spawned in a new fiber *)
  with effect (Foo i) k ->
           continue k (i + 1)
```

val r : int = 5

Effects interface

```
type _ eff += Foo : int -> int eff

val perform  : 'a eff -> 'a

type ('a,'b) continuation
val continue : ('a,'b) continuation -> 'a -> 'b
```

Algebraic Effects: Example

```
exception Foo of int

let f () = 1 + (raise (Foo 3))

let r =
    try
      f ()
    with Foo i -> i + 1
```

val r : int = 4

```
effect Foo : int -> int
```

let f () = 1 + ~~(perform (Foo 3))~~ 4

```
let r =
  try
    f () (* spawned in a new fiber *)
  with effect (Foo i) k ->
          continue k (i + 1)
```

val r : int = 5

Effects interface

```
type _ eff += Foo : int -> int eff

val perform  : 'a eff -> 'a

type ('a,'b) continuation
val continue : ('a,'b) continuation -> 'a -> 'b
```

Handlers are Deep!

```
effect Foo : int -> int

let f () = (perform (Foo 3)) (* 3 + 1 *)
         + (perform (Foo 3)) (* 3 + 1 *)
let r =
  try
    f () (* spawned in a new fiber *)
  with effect (Foo i) k ->
    (* continuation called outside try/with *)
    continue k (i + 1)
```

val r : int = 8

Handlers are Deep!

```
effect Foo : int -> int

let f () = (perform (Foo 3)) (* 3 + 1 *)
         + (perform (Foo 3)) (* 3 + 1 *)
let r =
  try
    f () (* spawned in a new fiber *)
  with effect (Foo i) k ->
    (* continuation called outside try/with *)
    continue k (i + 1)
```

val r : int = 8

Handlers are Deep!

```
effect Foo : int -> int

let f () = (perform (Foo 3)) (* 3 + 1 *)
         + (perform (Foo 3)) (* 3 + 1 *)
let r =
  try
    f () (* spawned in a new fiber *)
  with effect (Foo i) k ->
    (* continuation called outside try/with *)
    continue k (i + 1)
```

val r : int = 8

Scheduler Demo[1]

[1] https://github.com/kayceesrk/ocaml15-eff/tree/master/chameneos-redux

Implementation

- Fibers: Heap allocated, dynamically resized stacks
    - ~10s of bytes
    - No unnecessary closure allocation costs unlike CPS

Implementation

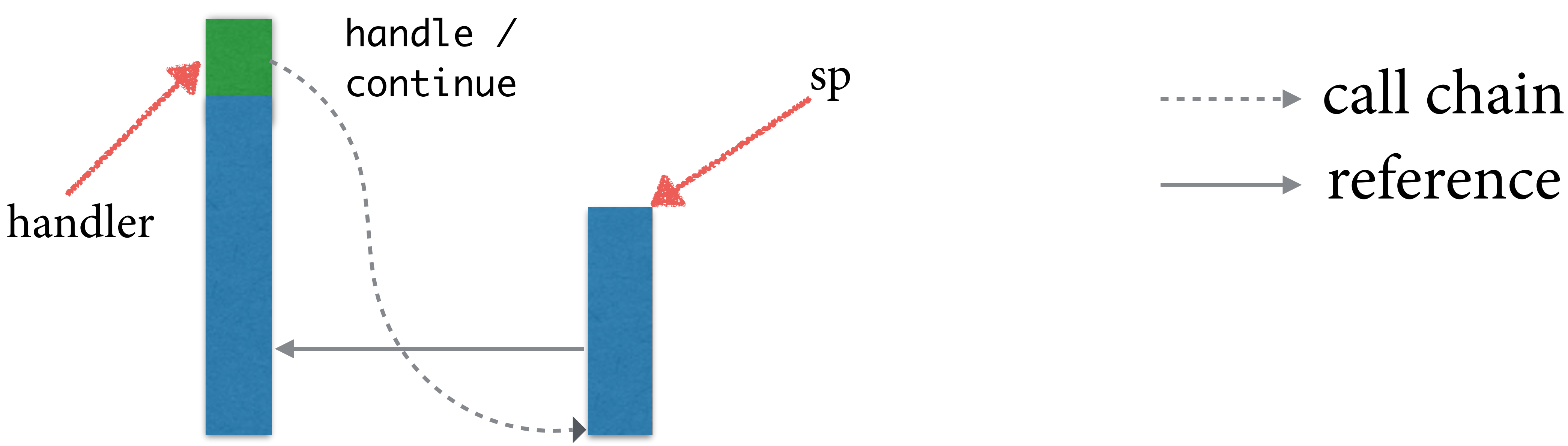
- Fibers: Heap allocated, dynamically resized stacks
  - ~10s of bytes
  - No unnecessary closure allocation costs unlike CPS
- One-shot delimited continuations
  - Simplifies reasoning about resources - sockets, locks, etc.

Implementation

- Fibers: Heap allocated, dynamically resized stacks
  - ~10s of bytes
  - No unnecessary closure allocation costs unlike CPS
- One-shot delimited continuations
  - Simplifies reasoning about resources - sockets, locks, etc.
- Handlers —> Linked-list of fibers

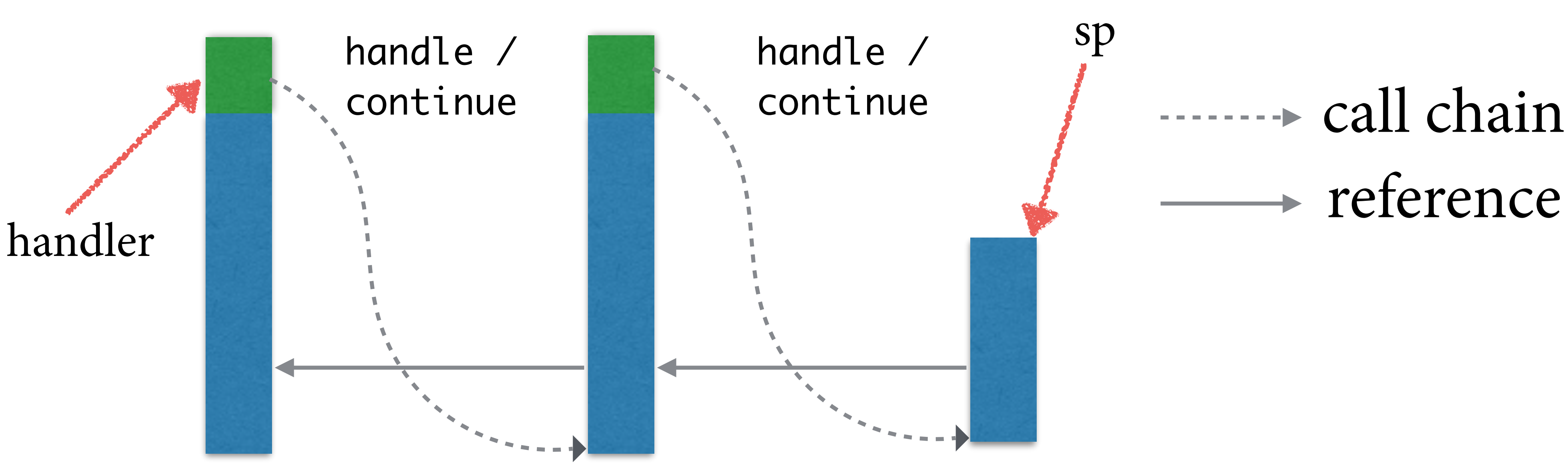
Implementation

- Fibers: Heap allocated, dynamically resized stacks
    - ~10s of bytes
    - No unnecessary closure allocation costs unlike CPS
- One-shot delimited continuations
    - Simplifies reasoning about resources - sockets, locks, etc.
- Handlers —> Linked-list of fibers



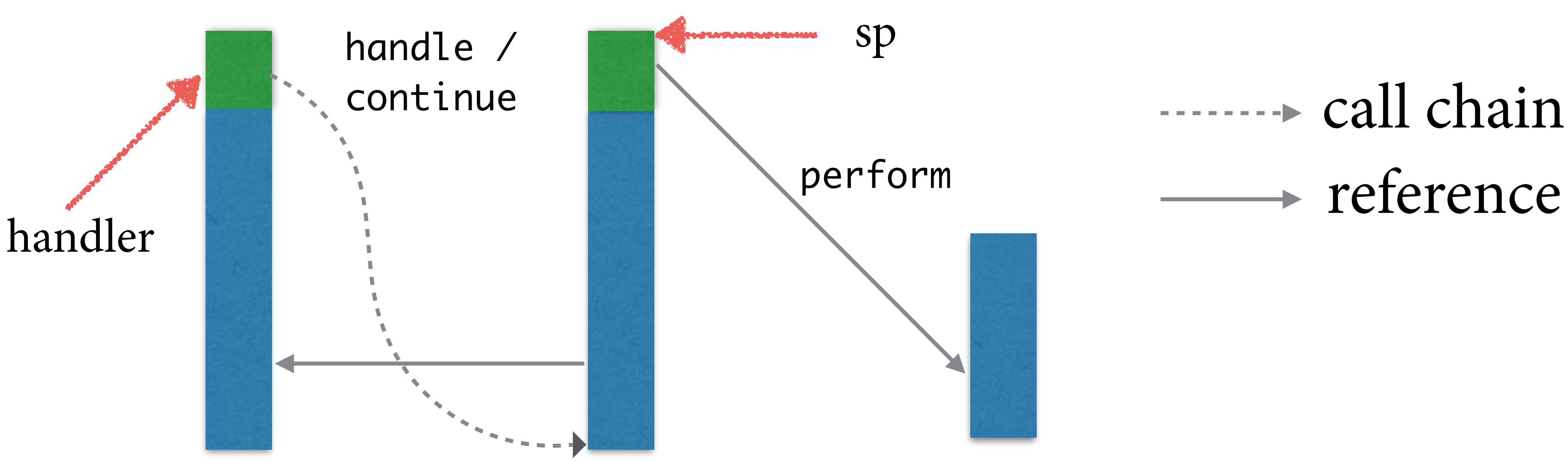
Implementation

- Fibers: Heap allocated, dynamically resized stacks
  - ~10s of bytes
  - No unnecessary closure allocation costs unlike CPS
- One-shot delimited continuations
  - Simplifies reasoning about resources - sockets, locks, etc.
- Handlers —> Linked-list of fibers

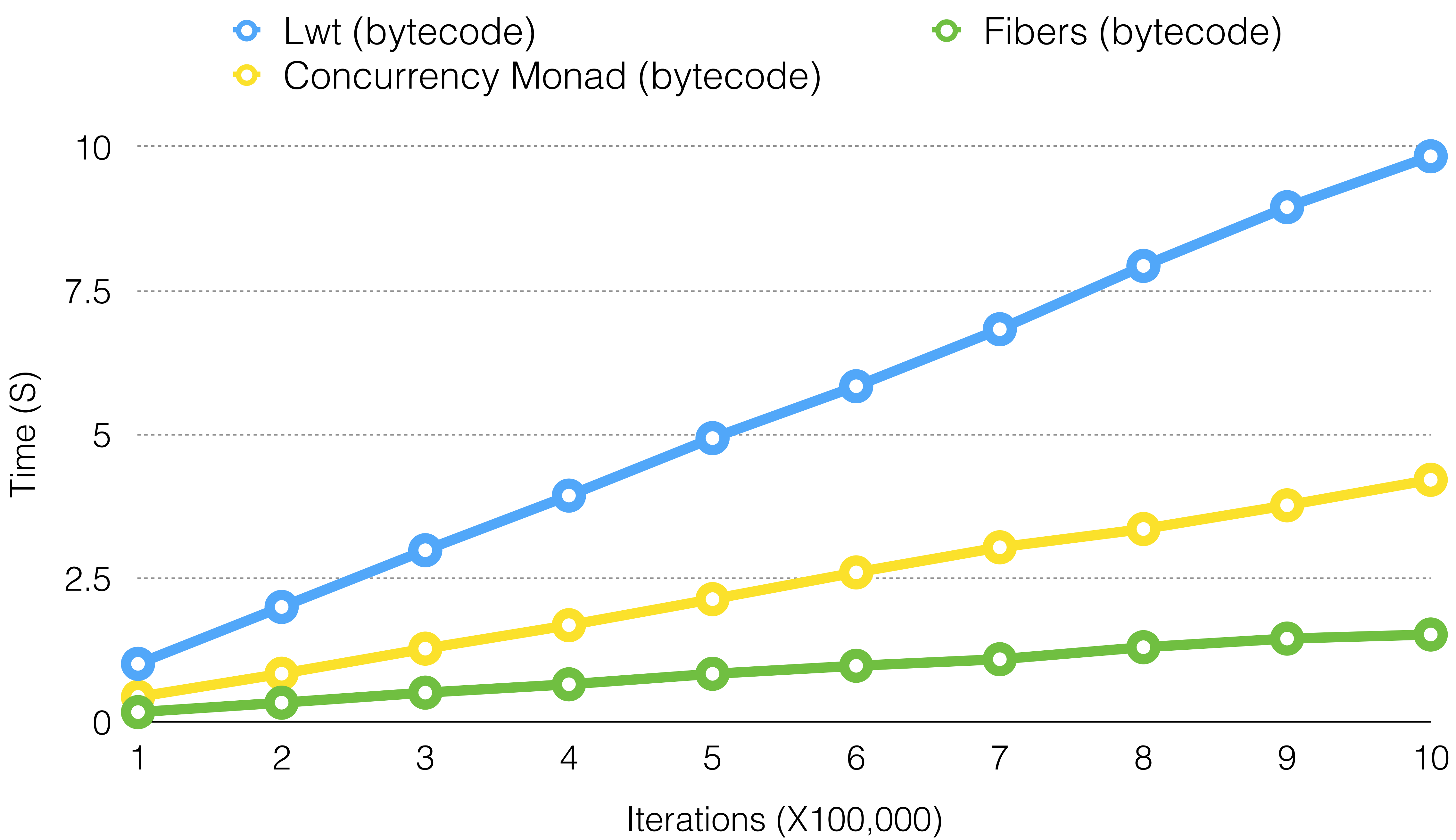


Implementation

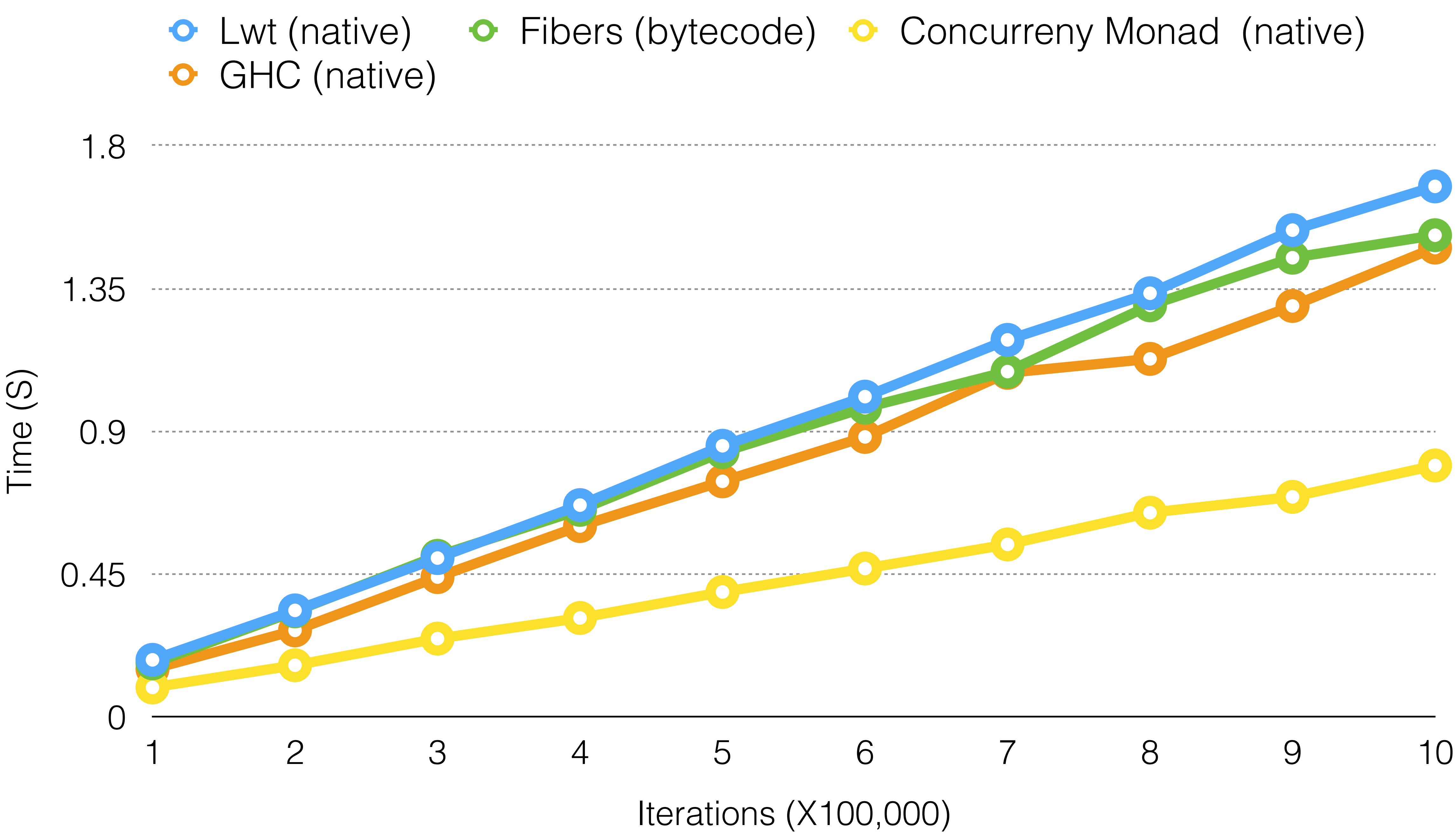
- Fibers: Heap allocated, dynamically resized stacks
  - ~10s of bytes
  - No unnecessary closure allocation costs unlike CPS
- One-shot delimited continuations
  - Simplifies reasoning about resources - sockets, locks, etc.
- Handlers —> Linked-list of fibers



Performance : Chameneos-Redux



Performance : Chameneos-Redux



Generator from Iterator[1]

```
type 'a t =
| Leaf
| Node of 'a t * 'a * 'a t

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

[1] https://github.com/kayceesrk/ocaml15-eff/blob/master/generator.ml

Generator from Iterator[1]

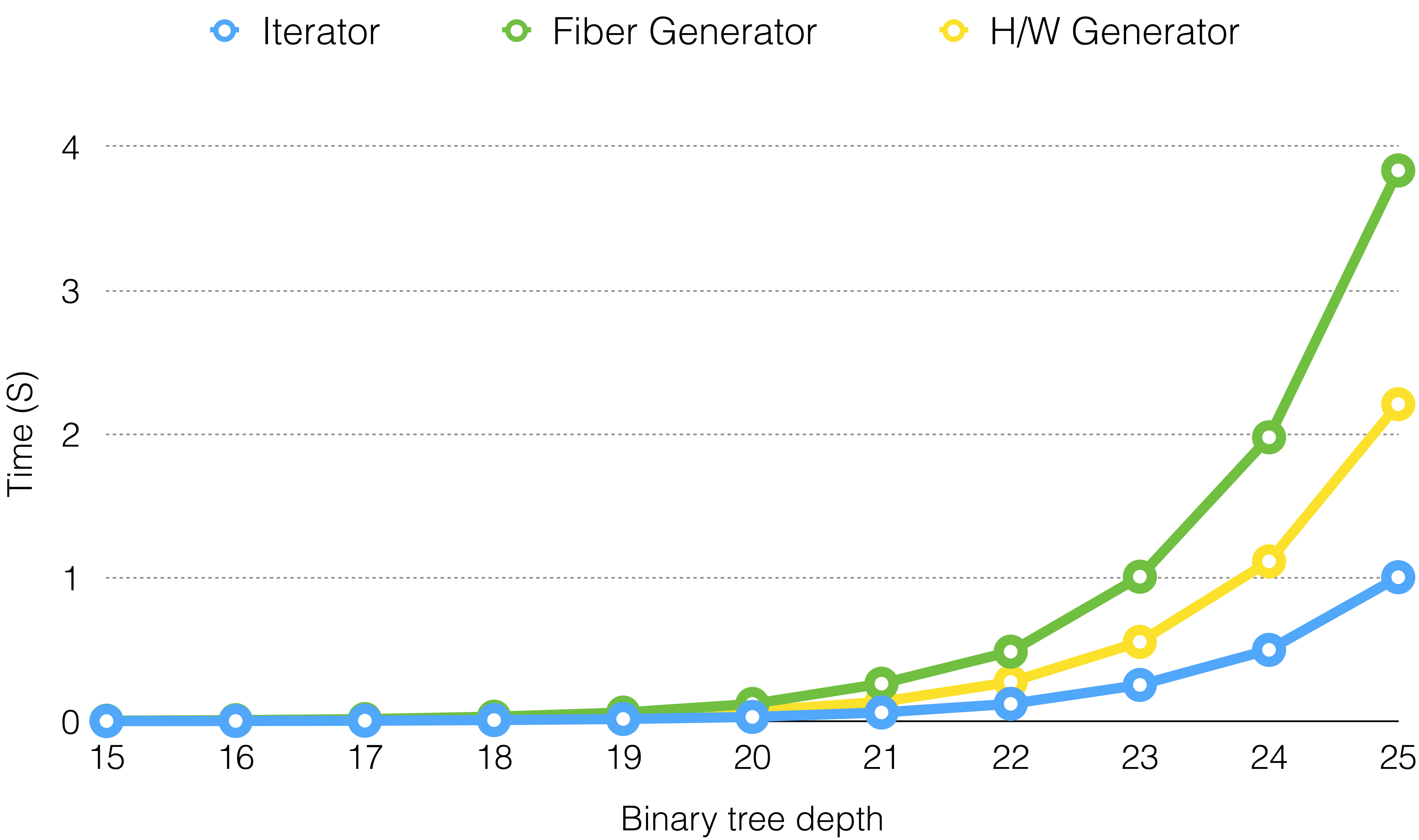
```
type 'a t =
| Leaf
| Node of 'a t * 'a * 'a t

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

(* val to_gen : 'a t -> (unit -> 'a option) *)
let to_gen (type a) (t : a t) =
  let module M = struct effect Next : a -> unit end in
  let open M in
  let step = ref (fun () -> assert false) in
  let first_step () =
    try
      iter (fun x -> perform (Next x)) t; None
    with effect (Next v) k ->
      step := continue k; Some v
  in
    step := first_step;
    fun () -> !step ()
```

[1] https://github.com/kayceesrk/ocaml15-eff/blob/master/generator.ml

Performance : Generator



Concerns

- Unchecked effects
  - Risks ~= exceptions
  - Effect inference in Eff[1]

[1] Matija Pretnar, “Inferring Algebraic Effects”, http://arxiv.org/abs/1312.2334

Concerns

- Unchecked effects
  - Risks ~= exceptions
  - Effect inference in Eff[1]
- Interfacing with monadic code (Lwt, Async)
  - Use monadic reflection to recover direct-style code[2]

[1] Matija Pretnar, "Inferring Algebraic Effects", http://arxiv.org/abs/1312.2334

[2] https://github.com/kayceesrk/ocaml15-eff/blob/master/reify_reflect.ml

Concerns

- Unchecked effects
  - Risks ~= exceptions
  - Effect inference in Eff[1]
- Interfacing with monadic code (Lwt, Async)
  - Use monadic reflection to recover direct-style code[2]
- Compilation to other backends (JS, Java?)
  - ES6 generators, ES7 async/await
  - Selective-CPS transform[3]

[1] Matija Pretnar, "Inferring Algebraic Effects", http://arxiv.org/abs/1312.2334

[2] https://github.com/kayceesrk/ocaml15-eff/blob/master/reify_reflect.ml

[3] T Rompf et al., "Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform", ICFP '09

Status

- Bytecode only. *Todo Native.*
- Several opportunities for optimisation
  - Continuations invoked at tail position
  - Dynamic search for effect handler
- Code
  - Multicore OCaml: https://github.com/ocamllabs/ocaml-multicore
  - Stand-alone effects: https://github.com/kayceesrk/ocaml/tree/effects
  - Effects examples: https://github.com/kayceesrk/ocaml-eff-example