Effective Programming in OCaml

“KC” Sivaramakrishnan
Multicore OCaml

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

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

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

**Overlapped execution**

**Simultaneous execution**
Multicore OCaml

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

**Effect Handlers**
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**
Concurrency is not parallelism

*Parallelism is a performance hack*

*whereas*

*concurrency is a program structuring mechanism*
Concurrency is not parallelism

*Parallelism is a performance hack*

*whereas*

*concurrency is a program structuring mechanism*

- OS threads give you parallelism and concurrency
  - Too heavy weight for concurrent programming
  - Http server with 1 OS thread per request is a terrible idea
Concurrency is not parallelism

*Parallelism is a performance hack*

*whereas*

*concurrency is a program structuring mechanism*

- OS threads give you parallelism and concurrency
  - Too heavy weight for concurrent programming
  - Http server with 1 OS thread per request is a terrible idea
- Programming languages provide concurrent programming mechanisms as *primitives*
  - async/await, generators, coroutines, etc.
Concurrency is not parallelism

Parallelism is a performance hack whereas concurrency is a program structuring mechanism

• OS threads give you parallelism and concurrency
  ✦ Too heavy weight for concurrent programming
  ✦ Http server with 1 OS thread per request is a terrible idea

• Programming languages provide concurrent programming mechanisms as **primitives**
  ✦ async/await, generators, coroutines, etc.

• Often include different primitives for concurrent programming
  ✦ JavaScript has async/await, generators, promises, and callbacks!!
Concurrent Programming in OCaml

- OCaml does not have primitive support for concurrent programming
Concurrent Programming in OCaml

- OCaml does not have primitive support for concurrent programming
- **Lwt** and **Async** - concurrent programming libraries in OCaml
  - Callback-oriented programming with monadic syntax `>>=`
  - But do not satisfy monad laws
Concurrent Programming in OCaml

- OCaml does not have primitive support for concurrent programming

- **Lwt** and **Async** - concurrent programming libraries in OCaml
  - Callback-oriented programming with monadic syntax `>>=`
  - But do not satisfy monad laws

- Suffers many pitfalls of *callback-oriented programming*
  - No backtraces, No exception, monadic syntax, too many closures
Concurrent Programming in OCaml

- OCaml does not have primitive support for concurrent programming

- **Lwt** and **Async** - concurrent programming libraries in OCaml
  - Callback-oriented programming with monadic syntax ` >>=`
  - But do not satisfy monad laws

- Suffers many pitfalls of *callback-oriented programming*
  - No backtraces, No exception, monadic syntax, too many closures

- Go (goroutines) and GHC Haskell (threads) have better abstractions — lightweight threads
Concurrent Programming in OCaml

- OCaml does not have primitive support for concurrent programming
- **Lwt** and **Async** - concurrent programming libraries in OCaml
  - Callback-oriented programming with monadic syntax `>>=`
  - But do not satisfy monad laws
- Suffers many pitfalls of *callback-oriented programming*
  - No backtraces, No exception, monadic syntax, too many closures
- Go (goroutines) and GHC Haskell (threads) have better abstractions — lightweight threads

*Should we add lightweight threads to OCaml?*
Effect Handlers

- A mechanism for programming with user-defined effects
Effect Handlers

• A mechanism for programming with user-defined effects

• Modular basis of non-local control-flow mechanisms

  ✦ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines
Effect Handlers

• A mechanism for programming with user-defined effects

• Modular basis of non-local control-flow mechanisms
  - Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines

• Effect handlers ~= first-class, restartable exceptions
  - Similar to exceptions, performing an effect separate from handling it
An example

```plaintext
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 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

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

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

computation

effect declaration

effect E : string
An example

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

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

effect declaration

```ml
effect E : string
```

computation

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

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

effect declaration

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

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

suspends current computation

computation

handler
An example

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

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

**effect declaration**

- `effect E : string`  

**computation**

- `let comp () =`  
- `let main () =`  
- `let comp ()`  
- `with effect E k ->`  
- `print_string "0 ";`  
- `print_string (perform E);`  
- `print_string "3 ";`  
- `print_string "1 ";`  
- `continue k "2 ";`  
- `print_string "4 ";`  

**suspended current computation**

- `print_string "0 ";`  
- `print_string (perform E);`  
- `print_string "3 ";`  

**delimited continuation**

- `with effect E k ->`  
- `print_string "1 ";`  
- `continue k "2 ";`  
- `print_string "4 ";`  

**handler**

- `print_string "0 ";`  
- `print_string (perform E);`  
- `print_string "3 ";`  
- `print_string "1 ";`  
- `continue k "2 ";`  
- `print_string "4 ";`  

**computation handler**

- The `with effect E k ->` structure is a computation handler that captures the effect and suspends the current computation.
An example

```
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
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 ";
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 ";
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 ";
```

Fiber: A piece of stack + effect handler
Stepping through the example

```ocaml
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
Stepping through the example

```
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 "
```
Stepping through the example

```ocaml
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
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 ";
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 →

main

comp

sp →

k

0 1
Stepping through the example

```ocaml
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 "
```

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 ";
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 "

0 1 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 "

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 "

0 1 2 3 4
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
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 ()
```

Diagram:
- `foo` to `bar` with `parent` connection
- `bar` to `baz` with `parent` connection
- `pc` to `foo`
- `sp` to `baz`
Handlers can be nested

```ocaml
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()
```

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

- Direct-style (no monads)
- User-code need not be aware of effects
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

```javascript
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
    
    ```javascript
    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
module L = MkGen (struct
  type 'a t = 'a list
  let iter = List.iter
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)
Generators: Tree

```ocaml
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 *)

```ocaml
let rec make = function
  | 0 -> Leaf
  | n -> let t = make (n-1) in Node (t, n, t)
```
Generators: Tree

```ocaml
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)

let rec make = function
 | 0 -> Leaf
 | n -> let t = make (n-1)
    in Node (t,n,t)

let t = make 2
```

(* Make a complete binary tree of depth [n] using [O(n)] space *)
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 *)
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
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

- Effective OCaml
  - Track both user-defined and built-in (ref, io, exceptions) effects
  - OCaml becomes a pure language (in the Haskell sense — divergence allowed)
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

• Effective OCaml
  ✦ Track both user-defined and built-in (ref, io, exceptions) effects
  ✦ **OCaml becomes a pure language** (in the Haskell sense — divergence allowed)

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

**Syntax is still in the works**
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*
Retrofitting Challenges

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

**Backwards compatibility before fancy new features**
Systems Programming

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

• 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
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

```ocaml
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
```
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.

```ocaml
defensive_style

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
```
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.

```ocaml
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
```

This function `copy` reads a line from the input channel `ic`, outputs it to the output channel `oc`, and loops until it encounters the end of the file or encounters an error. The `with` clause handles exceptions, closing the channels and raising the error if an `End_of_file` exception is encountered, or if an error occurred during the process.
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.

```ocaml
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
```

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

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

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

```plaintext
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 ()

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

• 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

```ocaml
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

```ocaml
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

```ocaml
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, llldb, perf, libunwind, etc.
  - DWARF stack unwinding support
- *Multicore OCaml supports DWARF stack unwinding across fibers*

```ocaml
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 *)
```

Stack grows down
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 *)
```
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: ...
let foo () =
  (∗ a ∗)
try
  (∗ b ∗)
  perform E
  (∗ d ∗)
with effect E k ->
  (∗ c ∗)
  continue k ()
  (∗ e ∗)
let foo () =
  (* a *)
try
  (* b *)
  perform E
  (* d *)
with effect E k ->
  (* c *)
  continue k ()
  (* e *)

- Each of the instruction sequences involves a stack switch

<table>
<thead>
<tr>
<th>Instruction Sequence</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>a to b</td>
<td>Create a new stack &amp; run the computation</td>
</tr>
<tr>
<td>b to c</td>
<td>Performing &amp; handling an effect</td>
</tr>
<tr>
<td>c to d</td>
<td>Resuming a continuation</td>
</tr>
<tr>
<td>d to e</td>
<td>Returning from a computation &amp; free the stack</td>
</tr>
</tbody>
</table>
let foo () = (* a *)
try (* b *)
perform E (* d *)
with effect E k -> (* c *)
continue k () (* e *).
let foo () = (* a *)
  try
    (* b *)
    perform E (* d *)
  with effect E k -> (* c *)
    continue k ()
  (* e *)

<table>
<thead>
<tr>
<th>Instruction Sequence</th>
<th>Significance</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a to b</td>
<td>Create a new stack &amp; run the computation</td>
<td>23</td>
</tr>
<tr>
<td>b to c</td>
<td>Performing &amp; handling an effect</td>
<td>5</td>
</tr>
<tr>
<td>c to d</td>
<td>Resuming a continuation</td>
<td>11</td>
</tr>
<tr>
<td>d to e</td>
<td>Returning from a computation &amp; free the stack</td>
<td>7</td>
</tr>
</tbody>
</table>

- Each of the instruction sequences involves a stack switch
- Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
  - 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

<table>
<thead>
<tr>
<th>Variant</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator (baseline)</td>
<td>202</td>
</tr>
<tr>
<td>hw-generator</td>
<td>837 (3.76x)</td>
</tr>
<tr>
<td>eh-generator</td>
<td>1879 (9.30x)</td>
</tr>
</tbody>
</table>
## Performance: Generators

### Multicore OCaml

<table>
<thead>
<tr>
<th>Variant</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator (baseline)</td>
<td>202</td>
</tr>
<tr>
<td>hw-generator</td>
<td>837 (3.76x)</td>
</tr>
<tr>
<td>eh-generator</td>
<td>1879 (9.30x)</td>
</tr>
</tbody>
</table>

### nodejs 14.07

<table>
<thead>
<tr>
<th>Variant</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator (baseline)</td>
<td>492</td>
</tr>
<tr>
<td>generator</td>
<td>43842 (89.1x)</td>
</tr>
</tbody>
</table>
Performance: WebServer

- Effect handlers for asynchronous I/O in direct-style
  - [https://github.com/kayceesrk/ocaml-aeio/](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/](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/](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
Thanks!

Install Multicore OCaml

$ opam switch create 4.10.0+multicore \
   --packages=ocaml-variants.4.10.0+multicore \
   --repositories=multicore=git+https://github.com/ocaml-multicore/multicore-opam.git,default

- Multicore OCaml — https://github.com/ocaml-multicore/ocaml-multicore
- Effects Examples — https://github.com/ocaml-multicore/effects-examples
- Sivaramakrishnan et al, “Retrofitting Parallelism onto OCaml”, ICFP 2020
- Dolan et al, “Concurrent System Programming with Effect Handlers”, TFP 2017
Bonus Slides
No effects performance
No effects performance
No effects performance

- ~1% faster than stock (geomean of normalised running times)
  - Difference under measurement noise mostly
  - Outliers due to difference in allocators