Effect Handlers in Multicore 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**

*Time*
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
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
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 monad 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 monad syntax
  - But do not satisfy monad laws
- Suffers many pitfalls of callback-oriented programming
  - No backtraces, exceptions can’t be used, monadic syntax
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 monad syntax
  ✦ But do not satisfy monad laws

• Suffers many pitfalls of callback-oriented programming
  ✦ No backtraces, exceptions can’t be used, monadic syntax

• 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 monad syntax
  - But do not satisfy monad laws

- Suffers many pitfalls of callback-oriented programming
  - No backtraces, exceptions can’t be used, monadic syntax

- 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 *declaration* separate from *interpretation* (c.f. exceptions)
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 *declaration* separate from *interpretation* (c.f. exceptions)

```latex
\begin{verbatim}
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 ";
\end{verbatim}
```
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 *declaration* separate from *interpretation* (c.f. exceptions)

```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 ";
```
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 declaration separate from interpretation (c.f. exceptions)

```ocaml
effect E : string

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 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 declaration** separate from *interpretation* (c.f. exceptions)

```plaintext
effect E : string

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**

**computation**

**handler**
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 *declaration* separate from *interpretation* (c.f. exceptions)

```
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

- 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 *declaration* separate from *interpretation* (c.f. exceptions)

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

let main () =
  try
    let comp () =
      with effect E k ->
        print_string "1 ";
        continue k "2 ";
        print_string "4 "
    in
      comp ();
    with effect E k ->
      print_string "5 ";
      print_string (perform E);
      print_string "6 ";
  with effect E k ->
    print_string "7 ";
    continue k "8 ";
    print_string "9 ";
  in
    print_string "10 ";
```

Effect declaration

```ocaml
let main () =
  try
    let comp () =
      with effect E k ->
        print_string "1 ";
        continue k "2 ";
        print_string "4 ";
      in
        comp ();
    with effect E k ->
      print_string "5 ";
      print_string (perform E);
      print_string "6 ";
  with effect E k ->
    print_string "7 ";
    continue k "8 ";
    print_string "9 ";
  in
    print_string "10 ";
```

Suspends current computation

Delimited continuation

Handler
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 *declaration* separate from *interpretation* (c.f. exceptions)

```haskell
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**

**delimited continuation**

**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 "

pc → main

sp →
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

```ocaml
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

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

```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

```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 ";

0 1
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 ";
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

```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 "
```

0 1 2 3 4
Handlers can be nested

```latex
\text{effect A : unit}
\text{effect B : unit}

\text{let baz () = perform A}
\text{let bar () = try baz () with effect B k \rightarrow continue k ()}
\text{let foo () = try bar () with effect A k \rightarrow continue k ()}
```

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 ()
```

```
```

pc →

foo → parent

bar → parent

baz → sp

```
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

```ocaml
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 =
  ...

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
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
Let's break down the code and discuss the results.

```ocaml
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
```

The key points are:

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

When we run the `main` function, the output looks like:

```
1.a
2.a
1.b
2.b
```
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
Generators: List

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

```ocaml
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)
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] *)
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)

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

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

Syntax is still in the works
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) effects
  ✦ OCaml becomes a pure language (in the Haskell sense)

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

• Today, Multicore OCaml effect handler static semantics is simple

Syntax is still in the works
(* OCaml extensible variant type *)
type 'a eff = ..

The declaration

effect E : string -> int

gets translated to

type _ eff = E : string -> int eff
Static Semantics

val perform: 'a eff -> 'a
val continue: ('a,'b) continuation -> 'a -> 'b
**Static Semantics**

```plaintext
compiled to

match e with
| None -> false
| Some b -> b
| effect (E s) k1 -> e1
| effect (F f) k2 -> e2

match_with (fun () -> e)
{ retc = (function None -> false
| Some b -> b);
  effc = (function
| (E s) -> (fun k1 -> e1)
| (F f) -> (fun k2 -> e2)
| e -> (fun k -> continue k (perform e)); }

assuming we have

(* Internal API *)
type 'a comp = unit -> 'a

type ('a,'b) handler = {
  retc: 'a -> 'b; (* value case *)
  effc: 'c.'c eff -> ('c,'b) continuation -> 'b; (* effect case *)
}

val match_with: 'a comp -> ('a,'b) handler -> 'b
```
Comparison to shift/reset

- Effect handlers equivalent in expressive power to other delimited control operators
  - Forster et al, “On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control”, JFP 2019
  - Macro-expressible to each other (ignoring types)
Comparison to shift/reset

• Effect handlers equivalent in expressive power to other delimited control operators
  ♦ Forster et al, “On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control”, JFP 2019
  ♦ Macro-expressible to each other (ignoring types)

• Nicer to program with thanks to the handler syntax

  goto :: while loop :: shift/reset :: effect handlers

  - Andrej Bauer
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*

- Low-latency and predictable performance
  - Fast exceptions, FFI
Retrofitting Challenges

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

• Low-latency and predictable performance
  ✦ Fast exceptions, FFI

• Excellent compatibility with debugging and profiling tools
  ✦ gdb, llad, perf, libunwind, etc.
Retrofitting Challenges

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

• Low-latency and predictable performance
  ✦ Fast exceptions, FFI

• Excellent compatibility with debugging and profiling tools
  ✦ gdb, lldb, perf, libunwind, etc.

Backwards compatibility
before
fancy new features
Defensive Programming

• OCaml is a systems programming language
  ✦ Manipulates resources such as files, sockets, buffers, etc.
Defensive 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
Defensive 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
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
```
Defensive 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
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
```
Defensive 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
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 -> raise e
```
Defensive 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
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

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 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

```ocaml

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 identified by End_of_file and Sys_error exceptions?
Discontinue

val discontinue: ('a,'b) continuation -> exn -> 'b

- 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
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*

• When calling an OCaml function, the caller expects the callee to return *exactly once* either with a value or an exception
  - Defensive programming already guards against exceptional return cases
Linearity

- With effect handlers if the captured continuation is dropped on the floor, then any function call may only return \textit{at-most once}

  - This breaks resource-safe legacy code
Linearity

- With effect handlers if the captured continuation is dropped on the floor, then any function call may only return \textit{at-most once}

\begin{itemize}
  \item This breaks resource-safe legacy code
\end{itemize}

```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 *)
```

```
Linearity

- We *assume* that well-formed programs resume captured continuations exactly once either using continue or discontinue

  - *Someone please add linear types to OCaml :-)*
• We *assume* that well-formed programs resume captured continuations exactly once either using continue or discontinue
  ✦ *Someone please add linear types to OCaml :-)*

• Linear use of continuations ensures that non-local control-flow and resources work well together
  ✦ No need for Scheme dynamic-wind
Linearity

• We assume that well-formed programs resume captured continuations exactly once either using continue or discontinue
  ✦ Someone please add linear types to OCaml :-) 

• Linear use of continuations ensures that non-local control-flow and resources work well together
  ✦ No need for Scheme dynamic-wind

• Core and Base provide unwind-protect implemented using exceptions
  ✦ Backwards compatibility of resourceful code ensured thanks to linearity and defensive programming
Foreign-function interface

(* meander.ml *)
external ocaml_to_c : unit -> int = "ocaml_to_c"

exception E1
exception E2

let c_to_ocaml () = raise E1;;
Callback.register "c_to_ocaml" c_to_ocaml;;

let omain () =
  try (* h1 *)
    try (* h2 *)
      ocaml_to_c ()
    with E2 -> -42
    with E1 -> 42;;

assert (omain () = 42)

/* meander.c */
#include <caml/mlvalues.h>
#include <caml/callback.h>

value ocaml_to_c (value unit) {
  caml_callback(*caml_named_value("c_to_ocaml"), Val_unit);
  return Val_int(0);
}
Stack Management

(* meander.ml *)

external ocaml_to_c : unit -> int = "ocaml_to_c"

exception E1
exception E2

let c_to_ocaml () = raise E1;;
Callback.register "c_to_ocaml" c_to_ocaml;;

let omain () =
  try (* h1 *)
    try (* h2 *)
      ocaml_to_c ()
      with E2 -> -42
      with E1 -> 42;;

assert (omain () = 42)

/* meander.c */
#include <caml/mlvalues.h>
#include <caml/callback.h

value ocaml_to_c (value unit) {
  caml_callback(*caml_named_value("c_to_ocaml"), Val_unit);
  return Val_int(0);
}
Stack Management

- Stack overflow checks
  - Reallocate with 2x stack space
- FFI requires stack switch
DWARF stack unwinding

Stock OCaml

Main C

- main
- ...
- caml_call_ocaml
- pc(fatal_uncaught)
- NULL
- ...
- omain
- pc(h1)
- sp(fatal_uncaught)
- pc(h2)
- sp(h1)
- ocaml_to_c
- caml_callback
- ...
- caml_call_ocaml
- pc(raise_exn_c)
- sp(h2)
- c_to_ocaml
- caml_raise_exn

Main OCaml

- #0 0x925dc in caml_raise_exn ()
- #1 0x6fd3e in camlMeander__c_to_ocaml_83 () at meander.ml:5
- #2 0x925a4 in caml_call_ocaml ()
- #3 0x8a84a in caml_callback_exn (...) at callback.c:145
- #4 caml_callback (...) at callback.c:199
- #5 0x76e0a in ocaml_to_c (unit=1) at meander.c:5
- #6 0x6fd77 in camlMeander__omain_88 () at meander.ml:10
- #7 0x6fe92 in camlMeander__entry () at meander.ml:13
- #8 0x6f719 in caml_program ()
- #9 0x925a4 in caml_call_ocaml ()
- #10 0x92e4c in caml_startup_common (...) at startup_nat.c:162
- #11 0x92eab in caml_startup_exn (...) at startup_nat.c:167
- #12 caml_startup (...) at startup_nat.c:172
- #13 0x6f55c in main (...) at main.c:44

Callback

- exn_ptr
- sp

External call
DWARF stack unwinding

- DWARF bytecode is a Turing complete language
DWARF stack unwinding

- DWARF bytecode is a Turing complete language
- In Multicore OCaml, we’ve encoded DWARF unwinding across callbacks, external calls and effect handlers
  ✦ gdb, lldb, perf continue to work!
DWARF stack unwinding

- DWARF bytecode is a Turing complete language

- In Multicore OCaml, we’ve encoded DWARF unwinding across callbacks, external calls and effect handlers
  - gdb, lldb, perf continue to work!

- Verified that the unwind tables are correct using an automated tool
  - Basitien et al, “Reliable and Fast DWARF-Based Stack Unwinding”, OOPSLA 2019
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
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 *)
```

<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>

- Each of the instruction sequences involves a stack switch
- 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 *)

<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
- For reference, 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](https://github.com/ocaml/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/](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

![Graph (a)](image1.png)  
**Throughput**

![Graph (b)](image2.png)  
**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