Practical Algebraic Effect Handlers in Multicore OCaml

“KC” Sivaramakrishnan

University of Cambridge
OCaml Labs
Multicore OCaml

- Native support for concurrency and parallelism
  
  [https://github.com/ocamllabs/ocaml-multicore](https://github.com/ocamllabs/ocaml-multicore)

- Led from OCaml Labs
  
  - KC, Stephen Dolan, Leo White (Jane Street) & others..

- In this talk: *Practical* algebraic effect handlers
  
  - Why algebraic effects in multicore OCaml?
  
  - How to make them practical?
    
    - Don’t break existing programs
    
    - Performance backwards compatibility
Concurrency ≠ Parallelism

- Concurrency
  - Overlapped execution of processes
  - Fibers — language level lightweight threads
  - 12M/s on 1 core. 30M/s on 4 cores.

- Parallelism
  - Simultaneous execution of computations
  - Domains — System thread + Context

- Concurrency ∩ Parallelism → Scalable Concurrency
User-level Schedulers

- Multiplexing fibers over domain(s)
- Bake scheduler into the runtime system (GHC)
  - Lack of flexibility
  - Maintenance onus on the compiler developers
- Allow programmers to describe schedulers!
  - Parallel search → LIFO work-stealing
  - Web-server → FIFO runqueue
  - Data parallel → Gang scheduling

- **Algebraic Effects and Handlers**
Algebraic effects & handlers

- Reasoning about computational effects in a pure setting
- Handlers for programming
  - G. Plotkin and M. Pretnar, Handlers of Algebraic Effects, 2009

**Eff**

Eff is a functional language with handlers of not only exceptions, but also of other computational effects such as state or I/O. With handlers, you can simply implement transactions, redirections, backtracking, multi-threading, and much more...

Reasons to like Eff

- Effects are first-class citizens
- Precise control over effects
- Strong theoretical
Algebraic Effects: Example

- Nice abstraction for programming with control-flow
- Separation effect *declaration* from its *interpretation*

```ocaml
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 ()
  with effect (Foo i) k ->
    continue k (i + 1)

('a,'b) continuation
```
Algebraic Effects: Example

- Nice abstraction for programming with control-flow
- Separation effect *declaration* from its *interpretation*

```ocaml
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 ()
  with effect (Foo i) k ->
    continue k (i + 1)
val r : int = 5
```

*fiber* — lightweight stack
Algebraic Effects in Multicore OCaml

- Unchecked
  ```ocaml
  effect Foo : unit
  let _ = perform Foo
  ```
  Exception: Unhandled.

- WIP: Effect System for OCaml
  ```ocaml
  effect foo = Foo : unit
  let _ = perform Foo
  ```
  Error: This expression performs effect foo, which has no default handler.

- Accurately track user-defined as well as native effects.
- Makes OCaml a pure language.

- Deep handler semantics
  ```ocaml
  let f () = (perform (Foo 3)) (* 3 + 1 *)
  + (perform (Foo 3)) (* 3 + 1 *)
  ```
  ```ocaml
  let r = try f () with effect (Foo i) k ->
  (* continuation resumed outside try/with *)
  continue k (i + 1)
  ```
Demo
Concurrent round-robin scheduler
Asynchronous I/O in direct-style

```javascript
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + ' to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            }).bind(this))
          })
        }
      })
    }
  }
})
```
Asynchronous I/O in direct-style

- Demo: Echo server
- Killer App

Facebook’s new skin for OCaml

Optimising compiler for OCaml to JavaScript
Concurrent data/sync structures

- Channels, MVars, Queues, Stacks, Countdown latches, etc.,
  - Need to interface with the scheduler!
- `MVar_put` & `MVar_get` as algebraic operations?

![Diagram showing program, MVars, Scheduler, Handler stack]

What is this interface?
Scheduler Interface

effect Suspend : (('a,unit) continuation -> unit) -> 'a

effect Resume : (('a,unit) continuation * 'a) -> unit

let rec spawn f =
  match f () with
  | () -> dequeue ()
  | effect Yield k -> enqueue k (); dequeue ()
  | effect (Fork f) k -> enqueue k (); spawn f
  | effect (Suspend f) k -> f k; dequeue ()
  | effect (Resume (k', v)) k ->
    enqueue k' v; ignore (continue k ())
MVar

type 'a mvar_state =
| Full of 'a * ('a * (unit,unit) continuation) Queue.t
| Empty of ('a,unit) continuation Queue.t

type 'a t = 'a mvar_state ref

let put v mv =
  match !mv with
  | Full (_, q) ->
      perform @@ Suspend (fun k -> Queue.push (v,k) q)
  | Empty q ->
      if Queue.is_empty q then
        mv := Full (v, Queue.create ())
      else
        let t = Queue.pop q in
        perform @@ Resume (t, v)

• Reagents https://github.com/ocamllabs/reagents
  • Composable lock-free programming
Preemptive Multithreading

• Conventional way: Build on top of signal handling

    open Sys
    set_signal sigalrm (Signal_handle (fun _ ->
        let k = (* Get current continuation *) in
        Sched.enqueue k;
        let k' = Sched.dequeue () in
        (* Set current continuation to k' *)
    ));

    Unix.setitimer interval Unix.ITIMER_REAL

• Not compositional: Signal handler is a \textit{callback}
  
    • \textit{Unclear where the handler runs.}

• Can we do better with effect handlers?
Preemptive Multithreading

- Treat asynchronous interrupts as effects!
- Can be raised asynchronously on demand

```plaintext
let rec spawn f =
  match f () with
  | () -> dequeue ()
  | effect Yield k -> yield k
  ...
  | effect TimerInterrupt k -> yield k

and yield k = enqueue k; dequeue ()
```

- What is the default behaviour for TimerInterrupt effect?
- Should all signals be handled this way? effect Signal : int -> unit
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
Tricky bug

• One-shot continuations + multicore schedulers

```ocaml
val call1cc : ('a cont -> 'a) -> 'a
val throw : 'a cont -> 'a -> 'b

let put v mv =
  match !mv with
  | Full (v', q) -> call1cc (fun k ->
    Queue.push (v,k) q;
    let k' = Sched.dequeue () in
    throw k' ())
  ....

• call1cc f, f run on the same stack!

• Possible that k is concurrently resumed on a different core!
```
Tricky bug

- No such bug here

```ocaml
let rec spawn f =
  match f () with
  | () -> dequeue ()
  | effect Yield k -> enqueue k (); dequeue ()
  | effect (Fork f) k -> enqueue k (); spawn f
  | effect (Suspend f) k -> f k; dequeue ()
  | effect (Resume (k', v)) k ->
    enqueue k' v; ignore (continue k ())
```

- f is run by the handler
  - Fiber performing suspend effect already suspended!
Native-code fibers — Vanilla
Native-code fibers — Effects

- system stack
- OCaml start program
- C call
- OCaml callback
- C call
- handle
- OCaml heap
Native-code fibers — Effects

• Stack overflow checks for OCaml functions
  • Eliminate SO checks for small tail recursive leaf functions
    • Slop space (16 words) at the bottom of stack
    • Frame sizes statically known
  • OCaml Compiler: 18K functions; *Eliminate checks for 11k functions*

• FFI calls are more expensive due to stack switching
  • Small context
    • No callee saved registers in OCaml
    • Allocation, exception, stack pointers in registers
  • Specialise for calls which {allocate / pass arguments on stack / do neither}
Performance: Vanilla OCaml

Normalised time (lower is better)

Effects ~0.9% slower
Performance: Chameneos-Redux

- Lwt
- Concurrency Monad
- GHC
- Fibers

**Direct-style**

**Specialised scheduler**
Generator from Iterator

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 ()
Performance : Generator

![Graph showing performance comparison between different generator types (Iterator, Fiber Generator, H/W Generator) across varying binary tree depths and time (in seconds). The graph illustrates the time taken by each generator type as the binary tree depth increases.]
Continuation cloning

• Our continuation are 1-shot.
  • Multi-shot continuations are useful for backtracking computations

• *Explicit cloning on demand!*
  • `Obj.clone_continuation : ('a,'b) continuation -> ('a,'b) continuation`

    ```
    effect Foo : unit

    let _ =
    try begin
      try perform Foo
      with effect Foo k -> continue k (perform Foo)
    end with effect Foo k ->
    continue (Obj.clone k) (); continue k ()
    ```

    *Continuation is resumed twice!*

Exception: Invalid_argument "continuation already taken".
Continuation cloning

Slowdown w.r.t. exceptional queens (X times)

# Queens

8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Exception (ref = 1)

Slowdown w.r.t. exceptional queens (X times)
Affine $\rightarrow$ Linear

- Affine continuations: resumed \textit{at-most} once
- Difficult to reason about resource cleanup

```ocaml
let fd = Unix.openfile "hello.ml" [Unix.O_RDWR] 0o640
try
  foo fd; Unix.close fd
with e -> Unix.close fd; raise e

let foo fd = perform DoesNotReturn
```
Affine → Linear

- Affine continuations: resumed *at-most* once
- Difficult to reason about resource cleanup

```ocaml
let fd = ref @@ Unix.openfile "hello.ml" [Unix.O_RDWR] 0o640
try
  foo !fd; Unix.close !fd
with e -> Unix.close !fd; raise e
| effect e k ->
  (* Dynamic wind *)
  Unix.close !fd;
let res = perform e in
fd := Unix.openfile "hello.ml" [Unix.O_RDWR] 0o640;
continue k res

let foo fd = perform Doesn'tReturn
```
Affine $\rightarrow$ Linear

- Affine continuations: resumed *at-most* once
  - Difficult to reason about resource cleanup
- Linear continuations: resumed *exactly* once
  - Implicit finalisers for fibers
  - Always unwind the stack with *exception* ThreadDeath
Affine $\rightarrow$ Linear

- Affine continuations: resumed *at-most* once
  - Difficult to reason about resource cleanup
- Linear continuations: resumed *exactly* once
  - Implicit finalisers for fibers
  - Always unwind the stack with *exception* ThreadDeath

raise ThreadDeath
• Affine continuations: resumed \textit{at-most} once
  • Difficult to reason about resource cleanup
• Linear continuations: resumed \textit{exactly} once
  • Implicit finalisers for fibers
  • Always unwind the stack with exception \texttt{ThreadDeath}

\[
\begin{align*}
K & \rightarrow K \\
\text{raise } \texttt{ThreadDeath} (??)
\end{align*}
\]
Summary

• Generalises control-flow programming
  • Async I/O, generators, promises, delimited control, etc.,

• Practicality
  • Native one-shot fibers for performance backwards compatibility
  • Backwards compatible effect system (Leo White, Hope 2016 Keynote)

• Real world Impact ➔ JavaScript :-)
  • React Fiber is based on OCaml effect handlers
  • Proposal to add effect handlers to EcmaScript

• Effect-based programming still in its infancy