Effect handlers in OCaml

KC Sivaramakrishnan¹ & Stephen Dolan¹

Leo White², Jeremy Yallop¹,³, Armaël Guéneau⁴, Anil Madhavapeddy¹,³
Concurrency ≠ Parallelism

- Concurrency
  - Programming technique
  - Overlapped execution of processes

- Parallelism
  - (Extreme) 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)
- Allow programmers to describe schedulers!
  - Parallel search —> LIFO work-stealing
  - Web-server —> FIFO runqueue
  - Data parallel —> Gang scheduling
- Algebraic Effects and Handlers
Algebraic Effects: Example

```plaintext
effect Foo : int -> int

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

let r = 
  try
    f ()
  with effect (Foo i) k ->
    continue k (i + 1)

val r : int = 8
```
Dynamic wind

let dynamic_wind before_thunk thunk after_thunk =
  before_thunk ()
let res =
  match thunk () with
  | v -> v
  | exception e -> after_thunk (); raise e
  | effect e k ->
    after_thunk ()
    let res' = perform e in
    before_thunk ()
    continue k res'
  in
  after_thunk ();
res
Effect systems and modularity

• Right now, we type effects like ML exceptions
  • (we're in the market for an effect system, if anyone has one lying around...)

• We need modularity, because effects can:
  • be local, dynamic and fresh
  • be abstracted, renamed and reuse

• We don't know statically whether two effects are the same
Scheduler Demo\textsuperscript{1}

Generator from Iterator

```ocaml
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 ()
```

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
Native-code fibers — Vanilla

C

OCaml

C

OCaml

C

OCaml

C

OCaml

OCaml start program
C call
OCaml callback
C call
OCaml callback
Native-code fibers — Effects

1. Stack overflow checks for OCaml functions
   - Simple static analysis eliminates many checks

2. FFI calls are more expensive due to stack switching
   - 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

Time (S) vs Iterations (X100,000)
Performance : Generator

Binary tree depth

Time (S)

Iterator  
Fiber Generator  
H/W Generator
Javascript backend

- js_of_ocaml
  - OCaml bytecode —> Javascript
- js_of_ocaml compiler pass
  - Whole-program selective CPS transformation
- Work-in-progress!
  - Selective CPS translation
fin.

Multicore OCaml: https://github.com/ocamllabs/ocaml-multicore

Examples: https://github.com/kayceesrk/ocaml-eff-example