Effective Concurrency with Algebraic Effects

Stephen Dolan\textsuperscript{1}, Leo White\textsuperscript{2}, \textbf{KC Sivaramakrishnan}\textsuperscript{1}, Jeremy Yallop\textsuperscript{1}, Anil Madhavapeddy\textsuperscript{1}
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 \cap Parallelism \Rightarrow Scalarable 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”
  - 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))

val r : int = 4

let r = try f () with Foo i -> i + 1
val r : int = 5

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)

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!

```ocaml
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!

```ocaml
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$^1$

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

![Diagram of Implementation](diagram.png)
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

![Graph showing performance over iterations]

- Lwt (bytecode)
- Fibers (bytecode)
- Concurrency Monad (bytecode)
Performance: Chameneos-Redux

Time (S)

Iterations (X100,000)

Lwt (native)  Fibers (bytecode)  Concurrency Monad (native)

GHC (native)
Generator from Iterator\textsuperscript{1}

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

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

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 ()
```

Performance: Generator

- **Time (S)**
  - 0
  - 1
  - 2
  - 3
  - 4

- **Binary tree depth**
  - 15
  - 16
  - 17
  - 18
  - 19
  - 20
  - 21
  - 22
  - 23
  - 24
  - 25

- **Types of Generators**
  - Iterator
  - Fiber Generator
  - H/W Generator
Concerns

- Unchecked effects
  - Risks $\sim$ exceptions
  - Effect inference in Eff$^1$

Concerns

- Unchecked effects
  - Risks \( \sim \) exceptions
  - Effect inference in \( \text{Eff}^1 \)
- Interfacing with monadic code (Lwt, Async)
  - Use monadic reflection to recover direct-style code\(^2\)

Concerns

• Unchecked effects
  • Risks \( \sim \) exceptions
  • Effect inference in \text{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\(^3\)

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](https://github.com/ocamllabs/ocaml-multicore)
  - Stand-alone effects: [https://github.com/kayceesrk/ocaml/tree/effects](https://github.com/kayceesrk/ocaml/tree/effects)
  - Effects examples: [https://github.com/kayceesrk/ocaml-eff-example](https://github.com/kayceesrk/ocaml-eff-example)