Concurrent System Programming with Effect Handlers

KC Sivaramakrishnan

University of Cambridge

OCaml Labs

Multicore OCaml

- Native support for *concurrency* and *parallelism* in OCaml
- Lead from OCaml Labs, University of Cambridge
  - Collaborators Stephen Dolan (OCaml Labs), Leo White (Jane Street)
- Expected to hit mainline in late 2019
- In this talk,
  - Focus on the concurrency subsystem — *Effect Handlers*
  - Build scalable concurrent network services in *idiomatic fashion*
  - Challenges in adding concurrency to a industrial-strength *sequential* language
  - Future work: Effect handler based OS and network services

Concurrency ≠ Parallelism

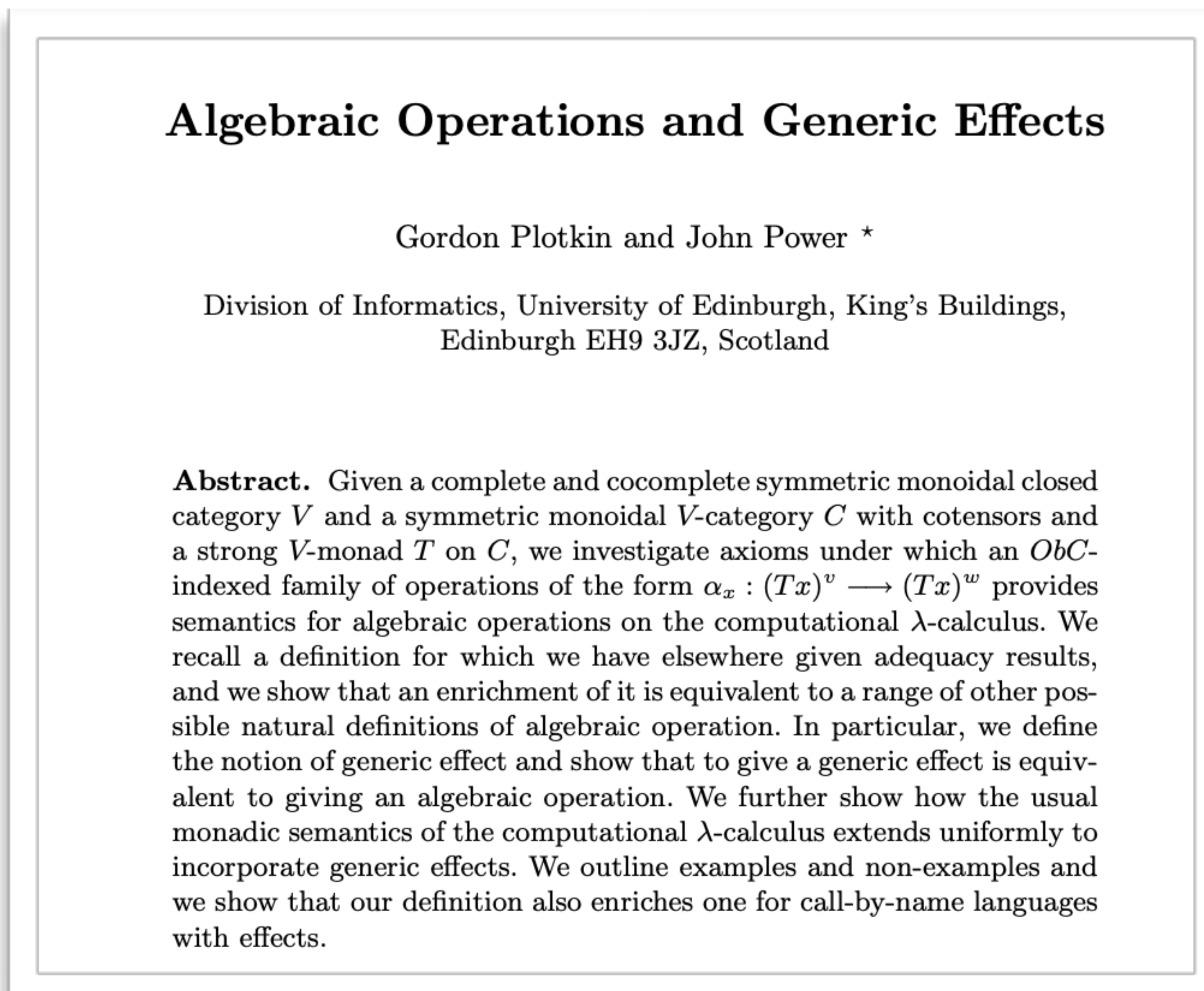
- Concurrency
  - Overlapped execution of processes
  - Fibers — language level lightweight threads
- Parallelism
  - Simultaneous execution of computations
  - Domains — System thread + Context
- Concurrency ∩ Parallelism ➔ ***Scalable Concurrency***

User-level Schedulers

- Multiplexing fibers over domain(s)
- Bake scheduler into the runtime (Go, GHC)
  - Lack of flexibility
  - Maintenance onus on the compiler developers
- Allow programmers to describe schedulers as OCaml libraries
  - Parallel search ➔ LIFO work-stealing
  - Web-server ➔ FIFO runqueue
  - Data parallel ➔ Gang scheduling
- ***Effect handlers***

Algebraic Effect Handlers : History

- Reasoning about computational effects in a pure setting
  - G. Plotkin and J. Power, Algebraic Operations and Generic Effects, 2002



Algebraic Operations and Generic Effects

Gordon Plotkin and John Power *

Division of Informatics, University of Edinburgh, King's Buildings, Edinburgh EH9 3JZ, Scotland

Abstract. Given a complete and cocomplete symmetric monoidal closed category V and a symmetric monoidal V-category C with cotensors and a strong V-monad T on C, we investigate axioms under which an ObC-indexed family of operations of the form $\alpha_x : (Tx)^v \longrightarrow (Tx)^w$ provides semantics for algebraic operations on the computational λ-calculus. We recall a definition for which we have elsewhere given adequacy results, and we show that an enrichment of it is equivalent to a range of other possible natural definitions of algebraic operation. In particular, we define the notion of generic effect and show that to give a generic effect is equivalent to giving an algebraic operation. We further show how the usual monadic semantics of the computational λ-calculus extends uniformly to incorporate generic effects. We outline examples and non-examples and we show that our definition also enriches one for call-by-name languages with effects.

Algebraic Effect Handlers : History

- Reasoning about computational effects in a pure setting
  - G. Plotkin and J. Power, Algebraic Operations and Generic Effects, 2002
- Handlers for programming
  - G. Plotkin and M. Pretnar, Handlers of Algebraic Effects, 2009

Handlers of Algebraic Effects

Gordon Plotkin * and Matija Pretnar **

Laboratory for Foundations of Computer Science,
School of Informatics, University of Edinburgh, Scotland

Abstract. We present an algebraic treatment of exception handlers and, more generally, introduce handlers for other computational effects representable by an algebraic theory. These include nondeterminism, interactive input/output, concurrency, state, time, and their combinations; in all cases the computation monad is the free-model monad of the theory. Each such handler corresponds to a model of the theory for the effects at hand. The handling construct, which applies a handler to a computation, is based on the one introduced by Benton and Kennedy, and is interpreted using the homomorphism induced by the universal property of the free model. This general construct can be used to describe previously unrelated concepts from both theory and practice.

Algebraic Effect Handlers : History

- Reasoning about computational effects in a pure setting
  - G. Plotkin and J. Power, Algebraic Operations and Generic Effects, 2002
- Handlers for programming
  - G. Plotkin and M. Pretnar, Handlers of Algebraic Effects, 2009
- Many prototype languages integrate algebraic effect handlers
  - Eff, Links, Koka, Frank, ….
  - *Multicore OCaml* is the first industrial-strength language to integrate effect handlers

Basics: recovering from errors (Demo)

Dynamic Semantics

- Powerful control operator to manipulate control flow
  - Equivalent in power to other delimited control operators (shift/reset, prompt/control, etc)
    - Type inference is simpler — no answer type polymorphism problem
    - *Much more pleasant to program with*
- Generalises other primitives that manipulate control-flow
  - async/await, generators, coroutines, promises
  - *Can be implemented as libraries rather than as primitives*
- Effect handler languages
  - Eff, Koka, Links, Frank, Unison, …
  - (Multicore) OCaml is the first industrial-strength language with effect handlers

Coroutines
(Demo)

Asynchronous I/O

- Direct-style

```
let handle conn =
  let request = read conn in
  write conn (respond_to request)
```

- Callback style

```
let handle conn =
  let ongoing = read conn in
  when_completed ongoing (fun req ->
    write conn (respond_to req))
```

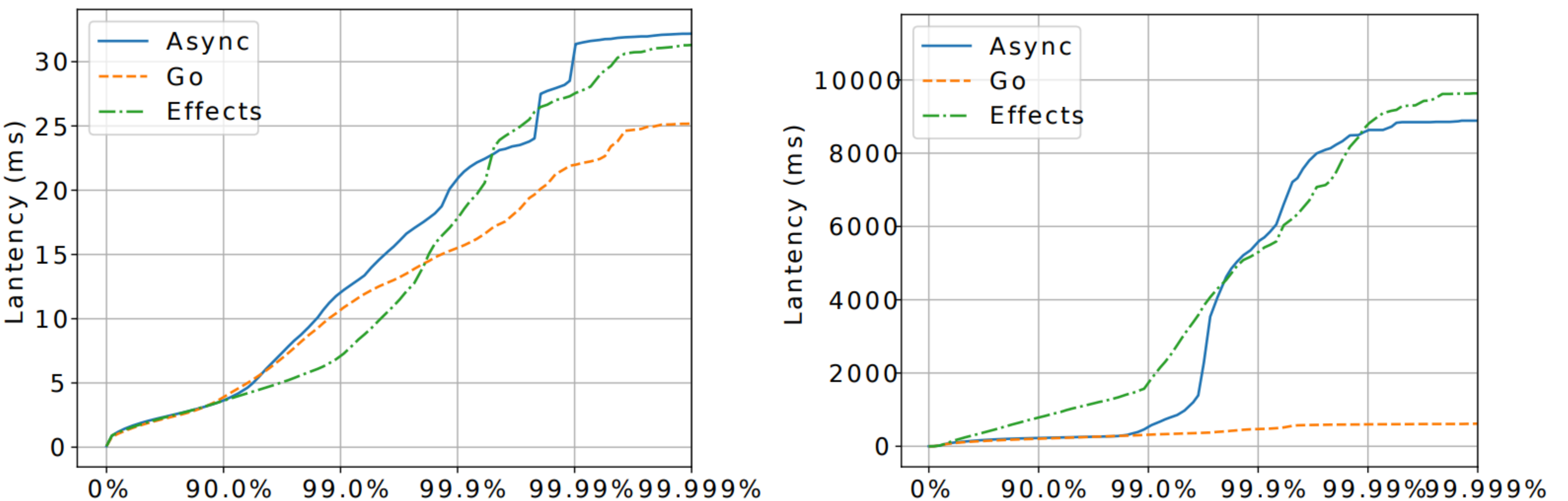
Callback hell!

http://ocamllabs.io/multicore/compare.js

Can we write fast asynchronous I/O code in direct-style?

Yes *(Async I/O demo)*

Performance



(a) Medium contention: 1k connections, 10k requests/sec

(b) High contention: 10k connections, 30k requests/sec

Fig. 2: Latency profile of client requests

Effect System

- WIP effect system for tracking effects in the type
  - Make unhandled effect a *compile time error*
- Nominal => Structural
  - No explicit effect declaration
  - Row polymorphism
- Effect polymorphism
  - `val map : ('a -[!p]-> 'b) -> 'a list -[!p] -> 'b list`

Representing continuations

- Continuations are heap-allocated, dynamically resized stacks
  - 10s of bytes initially
- *Linear* delimited continuations
  - Capturing a continuation is very cheap
  - Simplifies reasoning about resources — sockets, fds, locks etc
- Overheads
  - Stacks managed on the heap => stack overflow checks
  - FFI is more complex
  - ~1% avg (~9% max) slowdown compared to trunk

Enforcing linearity

- Continuations must be used *exactly* once
  - Not 0 times or 1+ times
  - No linear types => enforce dynamically
- Enforce *at-most once* use by invalidating the continuation on first-use
  - Raises exception on subsequent uses
- Enforcing *at-least once* use is tricky but important

Enforcing at-least once use

```
let process_file filename =
  let fd = Unix.openfile filename …
  try
    process fd; Unix.close fd
  with e -> Unix.close fd; raise e
```

```
let process fd =
… perform DoesNotReturn …

try process_file “hello.ml” with
| effect DoesNotReturn k -> ()
```

- Make use of the GC for enforcing at least once use

```
Gc.finalise k (fun k -> ignore(
  try discontinue k ThreadKilled with
  | Continuation_already_used -> ()
  | e -> failwith (Printexc.to_string e)))
```

Interrupts

- Interrupting ongoing computations is hard
- Synchronously, by polling (Go)
  - Code pollution, timeliness…
- Asynchronously, by stopping (GHC, C)
  - No context awareness => tricky with resource handling
  - Signal handlers are callbacks => introduce concurrency in an otherwise sequential program
- Interrupts are *“asynchronous effects”*

Preemptive multi-threading

```
val handle_signal : int (* signal number *)
                  -> ('a -[!r]-> 'b)
                  -> 'a -[Signal: int -> unit | !r]-> 'b
```

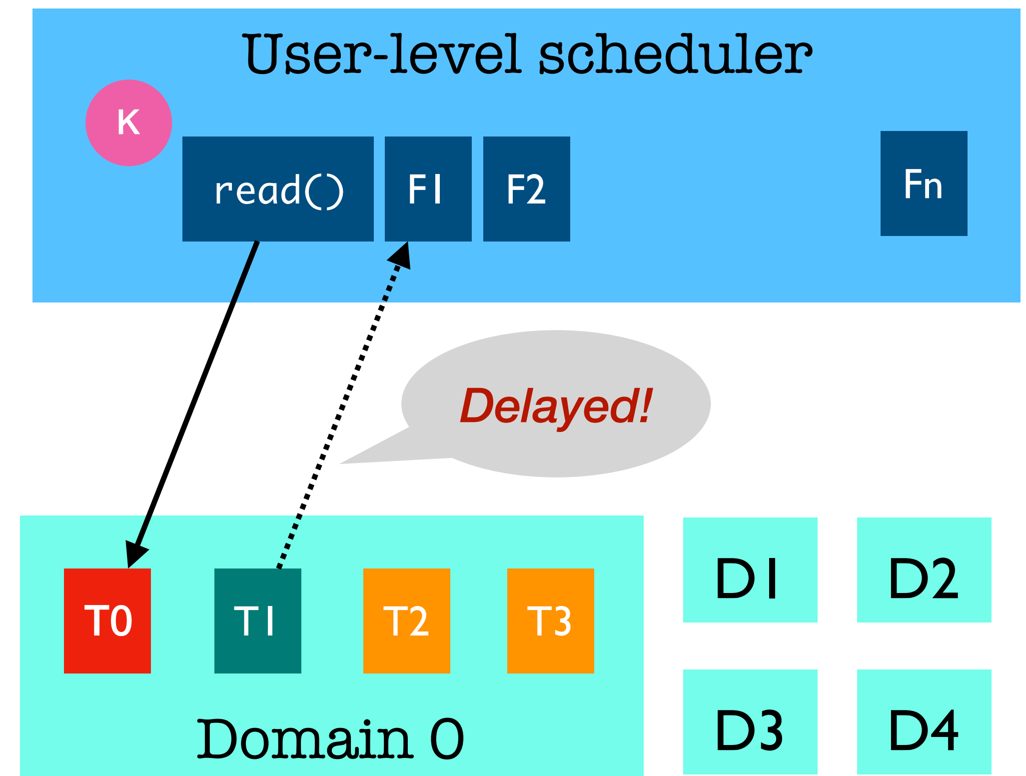
```
match (handle Sys.sigvtalrm main) () with
| _ -> dequeue ()
| effect (Async f) k ->
    enqueue (continue k); run f
| effect Yield k ->
    enqueue (continue k); dequeue ()
| effect (Signal Sys.sigvtalrm) k (* context *) ->
    enqueue (continue k); dequeue ()
```

Overlapping I/O with Compute

- Scalable OS networking & disk IO interfaces are exposed as callbacks
  - select, epoll, kqueue, Windows IOCP etc
  - Effect handlers can expose direct-style API!
- What about cases where the above doesn't work?
  - Posix says "*File descriptors associated with regular files shall always select true for ready to read, ready to write, and error conditions.*"
- Slow disks (NFS, HDD) => overlap computation with I/O?
- Similarly calls to DB engines, cached RPC calls, 3-rd party libraries…

Overlapping I/O with Compute

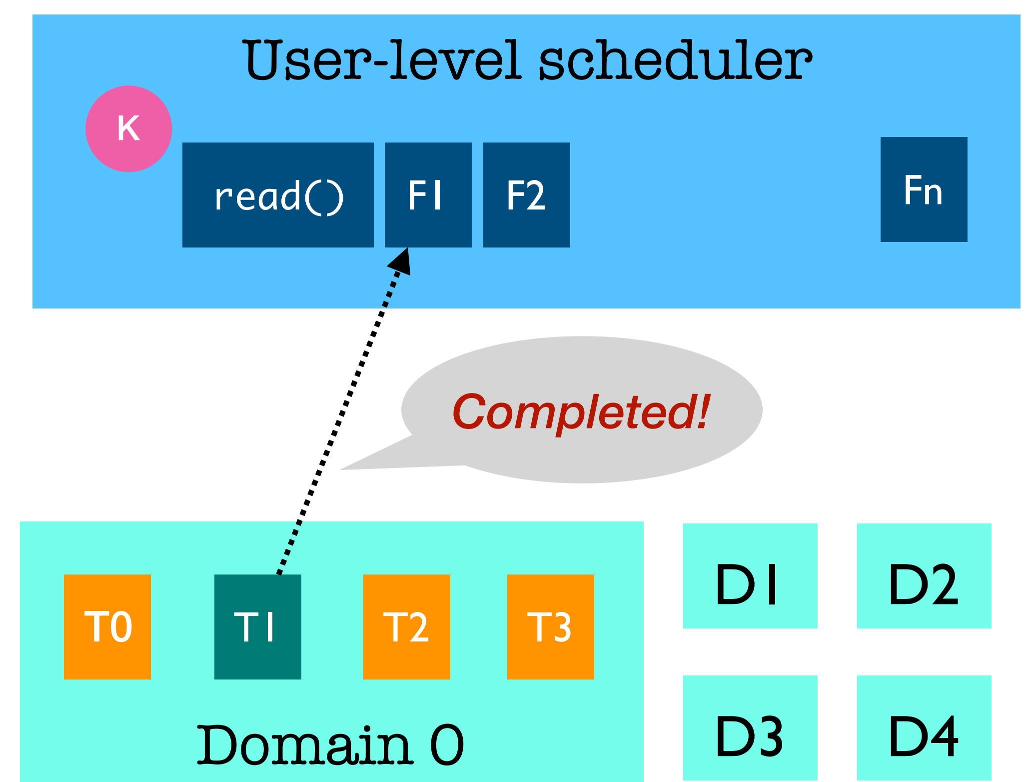
```
| effect (Delayed id) k ->
    Hashtbl.add ongoing_io id k;
    dequeue ()
```



Overlapping I/O with Compute

```
| effect (Delayed id) k ->
    Hashtbl.add ongoing_io id k;
    dequeue ()

| effect (Completed id) k ->
    let k' = Hashtbl.find ongoing_io id in
    Hashtbl.remove ongoing_io id;
    enqueue (continue k);
    continue k' ()
```



Summary

- Effect handlers are a great new tool for programming!
- They work really well for system programming
  - as long as you stick to the linear version
- They make nasty OS interfaces easier to use
  - and find salvation from callback hell!

ocamllabs/ocaml-multicore