This work is supported by

THE UNIVERSITY of EDINBURGH

School of informatics

EPSRC Centre for Doctoral Training in Pervasive Parallelism

EPSRC

Engineering and Physical Sciences Research Council

UNIVERSITY OF CAMBRIDGE

OCaml Labs

Runtime Agnostic Concurrency with Handlers

Pervasive Parallelism Lunch Talk

Daniel Hillerström

CDT Pervasive Parallelism
The University of Edinburgh, UK

October 12, 2016

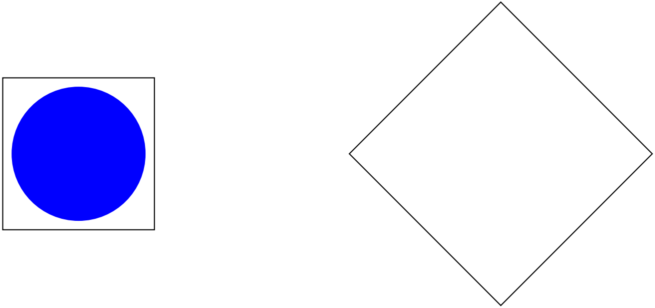
Outline

This talk consists of four parts

1. Motivation
2. Introduction to algebraic effects and their handlers
3. Abstract message-passing concurrency model (Hillerström, 2016)
4. Concurrency model instantiations (joint work with C. Dubach, S. Lindley, and KC Sivaramakrishnan) [work-in-progress]

Setting the stage

In the idealised world: write once, use everywhere (w.r.t some guarantees).

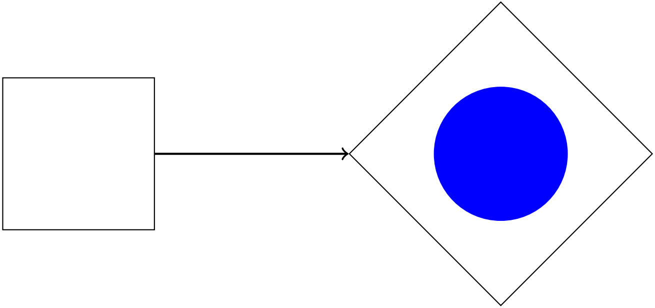


Really, really hard to achieve with different architectures in a parallel setting.

- ● Concurrent program.
- □ Rectangular computer architecture.
- ◇ Diamond computer architecture.

Setting the stage

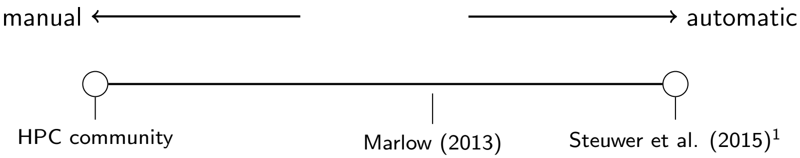
In the idealised world: write once, use everywhere (w.r.t some guarantees).



Really, really hard to achieve with different architectures in a parallel setting.

- Concurrent program.
- Rectangular computer architecture.
- Diamond computer architecture.

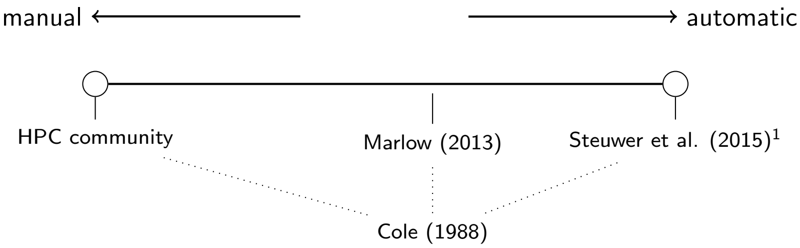
The axis of parallelisation



Lots of promising work in the area of automatic compiler-enabled parallelisation.

[1] Steuwer, Fensch, Lindley, and Dubach

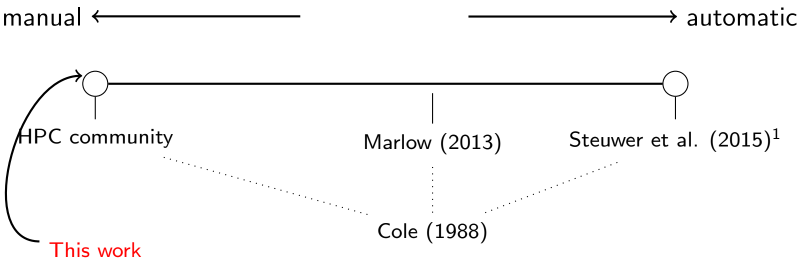
The axis of parallelisation



Lots of promising work in the area of automatic compiler-enabled parallelisation.

[1]Steuwer, Fensch, Lindley, and Dubach

The axis of parallelisation



Lots of promising work in the area of automatic compiler-enabled parallelisation.

But my work positions itself in the other extreme.

[1] Steuwer, Fensch, Lindley, and Dubach

Research hypothesis

Research hypothesis

Using algebraic effects and handlers we can decouple the concrete concurrency implementation from the use of concurrency primitives in our programs.

Computational effects

What are some examples of computational effects?

- Printing to standard output (print).
- Interacting with users (input).
- Nondeterminism (choice).
- Concurrency (fork, join, etc.)
- ... many more.

In short: computational effects are *pervasive*.

Nowadays, monads are the defacto abstraction for controlling effects (Moggi, 1991; Wadler 1992).

Algebraic effects (Plotkin and Power, 2001) combined with effect handlers (Plotkin and Pretnar, 2013) provide a modular alternative to monads.

Algebraic effects

Definition (Algebraic effect)

An algebraic effect is a collection of *abstract* operations. For example, we may think of

```
effect Incr : int -> int
effect Decr : int -> int
```

as describing an algebraic effect with two operations `Incr` and `Decr`. Essentially, an algebraic effect is an interface.

A silly example *abstract* computation:

```
let fortytwo () =
  let i = perform (Incr 42) in
  Printf.printf "%d\n" i;
  Printf.printf "%d\n" (perform (Decr i))
```

Handlers

From the previous slide

```
let fortytwo () =
  let i = perform (Incr 42) in
  Printf.printf "%d\n" i;
  Printf.printf "%d\n" (perform (Decr i))
```

Definition (Handler)

A handler is a *modular* interpreter over abstract computations.

Example interpretation of `Incr` and `Decr`:

```
let incr_decr m =
  match m () with
  | v -> v
  | effect (Incr i) comp -> continue comp (i+1)
  | effect (Decr i) comp -> continue comp (i-1)
```

To interpret `fortytwo` we apply `incr_decr`:

```
# incr_decr fortytwo;;
43
42
```

Implementations of algebraic effects and handlers

Programming languages with support for effect handlers

- *Eff* by Bauer and Pretnar (2015).
- Multicore OCaml by Dolan, White, Sivaramakrishnan, Yallop, and Madhavapeddy (2015).
- Links[2] by Hillerström and Lindley (2016)[3].
- Shonky by McBride (2016).
- Frank by Lindley, McBride, and McLaughlin (2017)[4].

Some embeddings in other programming languages

- Haskell library by Kiselyov, Sabry, and Swords (2013).
- Haskell library by Kammar, Lindley, and Oury (2013).
- Prolog library by Saleh and Schrijvers (2016).

---

[2] Originally developed by Cooper, Lindley, Wadler, and Yallop (2006).
[3] Later developed a compiler (Hillerström, Lindley, and Sivaramakrishnan, 2016).
[4] To appear at POPL 2017.

Implementations of algebraic effects and handlers

Programming languages with support for effect handlers

- *Eff* by Bauer and Pretnar (2015).
- Multicore OCaml by Dolan, White, Sivaramakrishnan, Yallop, and Madhavapeddy (2015).
- Links[2] by Hillerström and Lindley (2016)[3].
- Shonky by McBride (2016).
- Frank by Lindley, McBride, and McLaughlin (2017)[4].

Some embeddings in other programming languages

- Haskell library by Kiselyov, Sabry, and Swords (2013).
- Haskell library by Kammar, Lindley, and Oury (2013).
- Prolog library by Saleh and Schrijvers (2016).

---

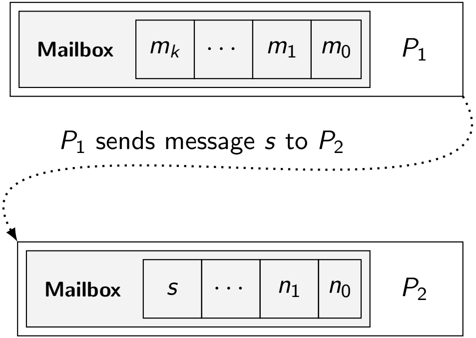
[2] Originally developed by Cooper, Lindley, Wadler, and Yallop (2006).

[3] Later developed a compiler (Hillerström, Lindley, and Sivaramakrishnan, 2016).

[4] To appear at POPL 2017.

Modelling message-passing concurrency

We are interesting in modelling a message-passing concurrency model.



Concurrency ingredients

Some *abstract* type `pid` that classifies process identifiers.

Three abstract operations for managing processes:

- `Spawn : (unit -> unit) -> pid` — for process creation.
- `Yield : unit` — for context switching.
- `Self : pid` — for self-referral.

Two operations for managing communication among processes:

- `Send : (pid * m) -> unit` — for sending a message of type `m`.
- `Recv : pid -> m option` — for receiving a message of type `m`.

An example abstract concurrent computation

This computation fits on a single slide:

```
let rec fib n parent () =
  let rec recv self =
    match perform (Recv self) with
    | None -> yield (); recv ()
    | Some msg -> msg
  in
  let send pid msg =
    perform (Send (pid, msg)); yield ()
  in
  if n < 2
  then send parent n
  else let self = perform Self in
       let _    = perform (Spawn (fib (n-1) self)) in
       let _    = perform (Spawn (fib (n-2) self)) in
       send parent ((recv self) + (recv self))
```

Scheduling

A handler for `Spawn` and `Yield` is a *scheduler* for processes. Consider the following code[5]:

```
let roundrobin f () =
  ...
  let rec spawn f =
    match f () with
    | () -> dequeue ()
    | effect (Spawn f) comp ->
       let child_pid = fresh_pid () in
       enqueue (fun () -> continue comp child_pid);
       spawn f
    | effect Yield comp      ->
       enqueue (fun () -> continue comp ()); dequeue ()
  in
  spawn f
```

[5] Adapted from Bauer and Pretnar (2015)

Communication

A mailbox can be represented as a mapping from `pid` to queues of `msg`.

```
let communication f =
  ...
  match f () with
  | v -> v
  | effect (Send (pid, msg)) k ->
     put pid msg;
     continue k ()
  | effect (Recv pid) k ->
     let msg = lookup pid in
     continue k msg
```

Model instantiation

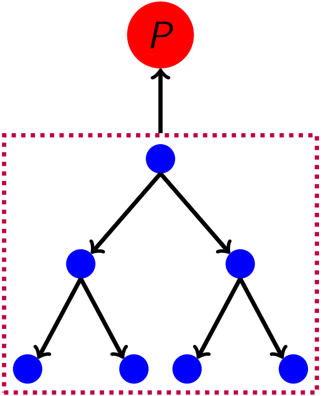
Let `type pid = int`. Now, we make a single-threaded instantiation of our concurrency model:

```
communication (roundrobin (fun () -> fib 7 (self ())))
```

Computes the value 8.

This instantiation is an implementation of *cooperative multitasking*.

Instantiation: Cooperative multitasking



- Actual system process.
- Abstract concurrent computation.
- Concurrency handler(s).

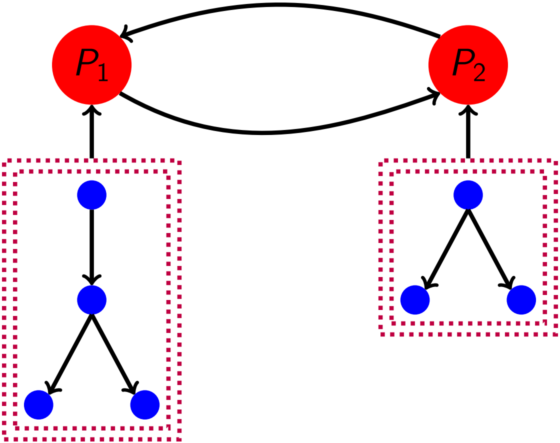
Observation: Model and implementation are decoupled

Observation

The abstract concurrent computations are implemented independently of our choice of model instantiation.

Can we instantiate our model with, perhaps, a parallel runtime such as MPI?

Instantiation: Ring scheduling



Actual system process.

Abstract concurrent computation.

Concurrency handler(s).

Not quite a free lunch

The change of implementation does not come for free

- Must change the initialisation of the application (our main function).
- Must implement new handlers that take advantage of the new runtime.

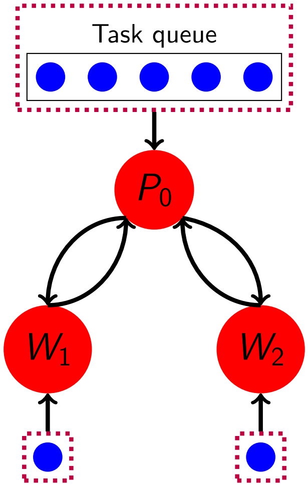
But the main parts of our program are left untouched. We may even reuse previous handlers.

Encapsulation of runtime specific parts inside of handlers

Let `type pid = int * int`.
Observation: calls to MPI routines only occur inside of handlers!

```
let mpihandler m () =
 ...
 let spawn f =
  match f () with
  | v  -> dequeue ()
  | effect (Spawn f) k -> (* Send task to neighbour *)
    let pid = fresh_pid neighbour in
    let _  = Mpi.send (Task (f,pid)) neighbour 1 Mpi.
   comm_world in
    continue k pid
  | effect (Send (pid,msg)) k -> (* Local or remote send? *)
    let _ = Mpi.send (Result (msg, pid)) neighbour 2 Mpi.
   comm_world in
    continue k ()
```

Instantiation: Single master, multiple workers



- Actual system process.
- Abstract concurrent computation.
- Concurrency handler(s).

Future work

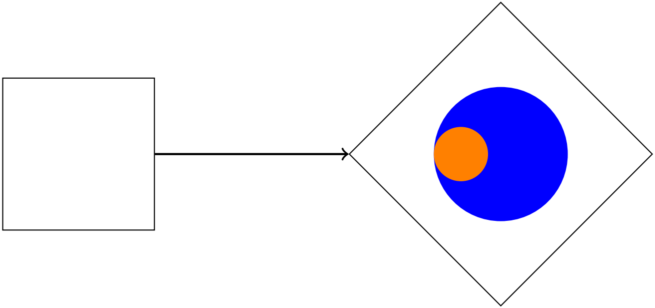
Next steps

- Measure/generate some performance figures
- Application to irregular parallel programs

Future, future work

- Serialisation of continuations
- Mixed-mode parallel programming

Summary



In summary

- Algebraic effects and their handlers provide a modular abstraction for controlling computational effects.
- Concurrency arises as "just" another natural controllable effect.
- Instantiate your concurrency model with your favourite implementation.
- Potential for *unleashing* and *taming* parallelism within abstract computations.

References I

Daniel Hillerström.
Compilation of effect handlers and their applications in concurrency.
Master's thesis, School of Informatics, the University of Edinburgh, Scotland, August 2016.

Daniel Hillerström, Sam Lindley, and KC Sivaramakrishnan.
Compiling Links effect handlers to the OCaml backend.
ML Workshop, 2016.

Daniel Hillerström and Sam Lindley.
Liberating effects with rows and handlers.
In *Proceedings of 1st International Workshop on Type-Driven Development (TyDe)*, TyDe 2016, pages 15–27, New York, NY, USA, September 2016. ACM.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy.
Effective concurrency through algebraic effects.
OCaml Workshop, 2015.

References II

Andrej Bauer and Matija Pretnar.
Programming with algebraic effects and handlers.
J. Log. Algebr. Meth. Program., 84(1):108–123, 2015.

Ohad Kammar, Sam Lindley, and Nicolas Oury.
Handlers in action.
In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 145–158, New York, NY, USA, 2013. ACM.

Daan Leijen.
Type directed compilation of row-typed algebraic effects.
Technical report, Microsoft Research, 2016.

References III

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach.
Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance opencl code.
In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 205–217. ACM, 2015.

Simon Marlow.
Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming.
Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming. O'Reilly Media, 2013.

Oleg Kiselyov, Amr Sabry, and Cameron Swords.
Extensible effects: an alternative to monad transformers.
In Chung-chieh Shan, editor, *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 59–70. ACM, 2013.

References IV

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop.
Links: Web programming without tiers.
In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006.

Amr Hany Saleh and Tom Schrijvers.
Efficient algebraic effect handlers for prolog.
CoRR, abs/1608.00816, 2016.

Eugenio Moggi.
Notions of computation and monads.
Inf. Comput., 93(1):55–92, 1991.

References V

Philip Wadler.
The essence of functional programming.
In Ravi Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 1–14. ACM Press, 1992.

Gordon D. Plotkin and John Power.
Adequacy for algebraic effects.
In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

Gordon D. Plotkin and Matija Pretnar.
Handling algebraic effects.
Logical Methods in Computer Science, 9(4), 2013.

References VI

Sam Lindley, Conor McBride, and Craig McLaughlin.
Do be do be do, 2017.
to appear at POPL 2017.

Conor McBride.
Shonky, 2016.
https://github.com/pigworker/shonky.

Murray Cole.
Algorithmic skeletons : a structured approach to the management of parallel computation.
PhD thesis, University of Edinburgh, UK, 1988.