Taking Back Control

… or implementing control idioms in user code

Daniel Hillerström/2018-12-13

Intern, Google Aarhus, Denmark

PhD student, The University of Edinburgh, UK

This talk

A programmer's introduction to effect handlers (my research topic).

- Toy examples
- My PhD work at glance
- Implementing asynchrony as a library
- Some (semi-)open problems
- The future

Google

This talk is largely based on materials from my CUFP'17 tutorial, c.f. http://cufp.org/2017/c3-daniel-hillerstrom-kc-concurrent-programming-with-effect-handlers.html

Why one might care

The programmer's perspective: take control from the runtime.

- Direct-style alternative to continuation passing style (CPS) and monadic programming
- Useful across a diverse spectrum
  - Probabilistic programming [Bingham et al., 2018]
  - Multi-stage programming [Yallop, 2017]
  - Concurrent programming [Dolan et al., 2017 and Leijen, 2017]
  - Modular program construction [Kammar et al., 2013]
- Expressive user-space for unikernels

The compiler writer's perspective: hand control to the programmer.

- Deep mathematical foundations [Plotkin and Power, 2001 and Plotkin and Pretnar, 2009]
- General enough to capture contemporary control idioms [Dolan et al., 2017, Leijen, 2017]
- Concrete enough to be amenable to optimisation [Wu and Schrijvers, 2015 and Leijen, 2018]
- Reduce complexity of the runtime/compiler [Dolan et al., 2016, Leijen, 2017]

Effect handlers

Operationally, effect handlers generalise exception handlers[†]

```
exception DivideByZero

let divide n d =
  match
     if d = 0 then raise DivideByZero
     else n / d
  with
  | result -> result
  | exception DivideByZero -> 0
```

[†]Benton and Kennedy (2001) style exception handlers

Google

Effect handlers

Operationally, effect handlers generalise exception handlers[†]

```
exception DivideByZero

let divide n d =
  match
     if d = 0 then raise DivideByZero
     else n / d
  with
  | result -> result
  | exception DivideByZero -> 0
```

transfers control to an enclosing handler

[†]Benton and Kennedy (2001) style exception handlers

Effect handlers

Operationally, effect handlers generalise exception handlers†

```
exception DivideByZero

let divide n d =
  match
     if d = 0 then raise DivideByZero
     else n / d
  with
  | result -> result
  | exception DivideByZero -> 0
```

†Benton and Kennedy (2001) style exception handlers

Google

Effect handlers

Operationally, effect handlers generalise exception handlers[†]

Terminology: abstract operation

```
effect DivideByZero : int

let divide n d =
  match
     if d = 0 then raise DivideByZero
     else n / d
  with
  | result -> result
  | exception DivideByZero -> 0
```

[†]Benton and Kennedy (2001) style exception handlers

Effect handlers

Operationally, effect handlers generalise exception handlers[†]

Terminology: abstract operation

```
effect DivideByZero : int

let divide n d =
  match
     if d = 0 then perform DivideByZero
     else n / d
  with
  | result -> result
  | exception DivideByZero -> 0
```

[†]Benton and Kennedy (2001) style exception handlers

Effect handlers

Operationally, effect handlers generalise exception handlers†

Terminology: abstract operation

```
effect DivideByZero : int

let divide n d =
  match
     if d = 0 then perform DivideByZero
     else n / d
  with
  | result -> result
  | effect DivideByZero k -> continue k 0
```

†Benton and Kennedy (2001) style exception handlers

Effect handlers

Operationally, effect handlers generalise exception handlers[†]

Terminology: abstract operation

```
effect DivideByZero : int

let divide n d =
  match
     if d = 0 then perform DivideByZero
     else n / d
  with
  | result -> result
  | effect DivideByZero k -> continue k 0
```

[†]Benton and Kennedy (2001) style exception handlers

Effect handlers

Operationally, effect handlers generalise exception handlers†

Terminology: abstract operation

```
effect DivideByZero : int

let divide n d =
  match
     if d = 0 then perform DivideByZero
     else n / d
  with
  | result -> result
  | effect DivideByZero k -> continue k 0
```

transfers control back to the invocation site with the provided value

†Benton and Kennedy (2001) style exception handlers

Effect handlers

Operationally, effect handlers generalise exception handlers†

Terminology: abstract operation

```
effect DivideByZero : int

let divide n d =
  match
     if d = 0 then perform DivideByZero
     else n / d
  with
  | result -> result
  | effect DivideByZero k -> continue k 0
```

transfers control back to the invocation site
with the provided value

```
continue : ('a,'b) continuation -> 'a -> 'b
```


†Benton and Kennedy (2001) style exception handlers

Handlers in action

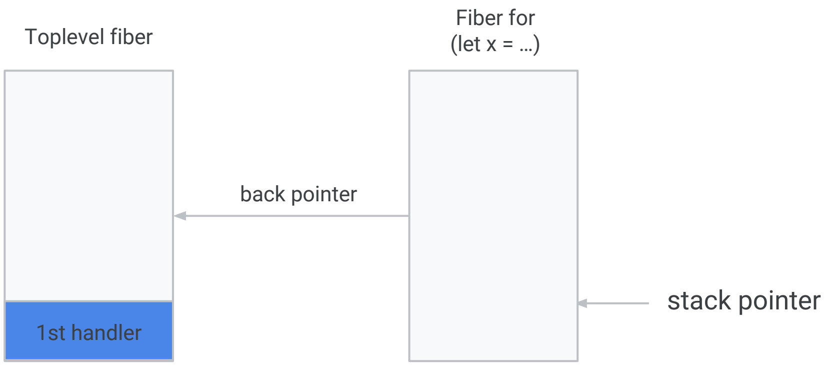
File: https://github.com/dhil/google-tech-talk-2018/blob/master/live/guess_the_number.ml

Execution stack

Fiber: heap allocated stack; grows and shrinks on demand.

Execution stack: a stack of fibers.

```
effect E : unit
match
  let x = match perform E with
          | effect F k -> ...
  in ...
with
| effect E k -> continue k ()
```

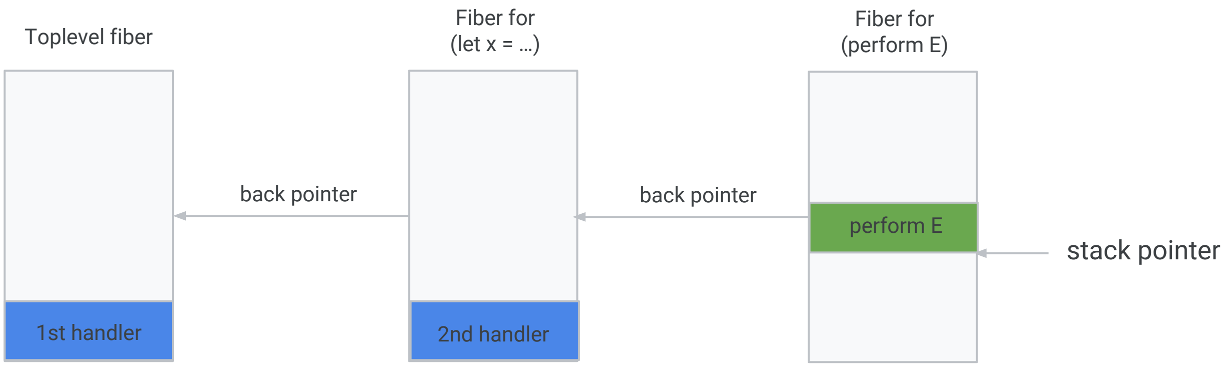


Execution stack

Fiber: heap allocated stack; grows and shrinks on demand.

Execution stack: a stack of fibers.

```
effect E : unit
match
  let x = match perform E with
          | effect F k -> ...
  in ...
with
| effect E k -> continue k ()
```

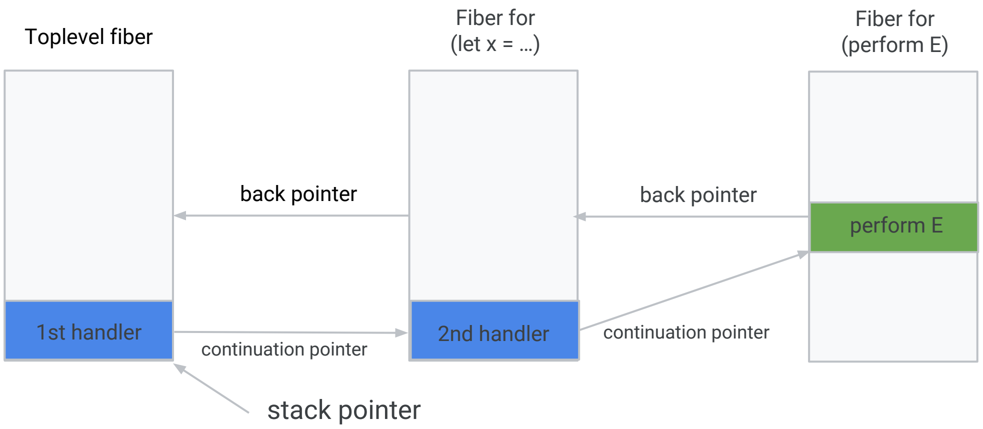


Execution stack

Fiber: heap allocated stack; grows and shrinks on demand.

Execution stack: a stack of fibers.

```
effect E : unit
match
  let x = match perform E with
          | effect F k -> ...
  in ...
with
| effect E k -> continue k ()
```

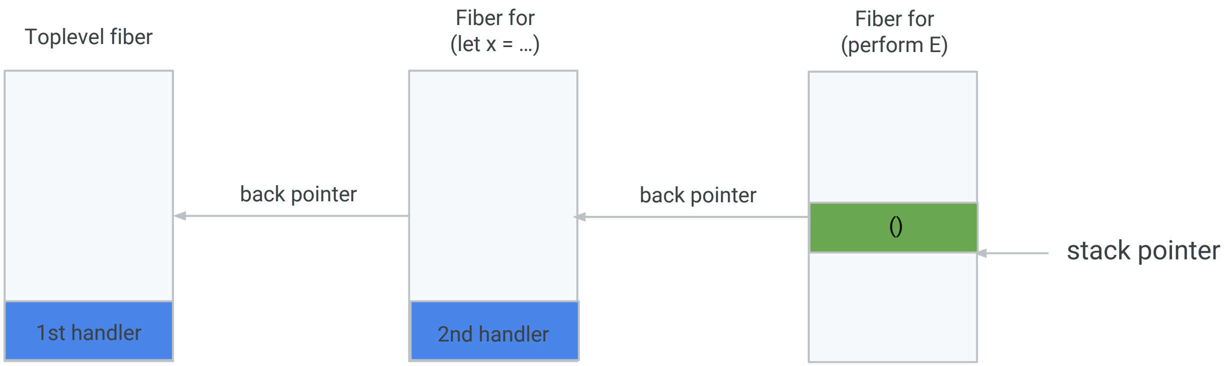


Execution stack

Fiber: heap allocated stack; grows and shrinks on demand.

Execution stack: a stack of fibers.

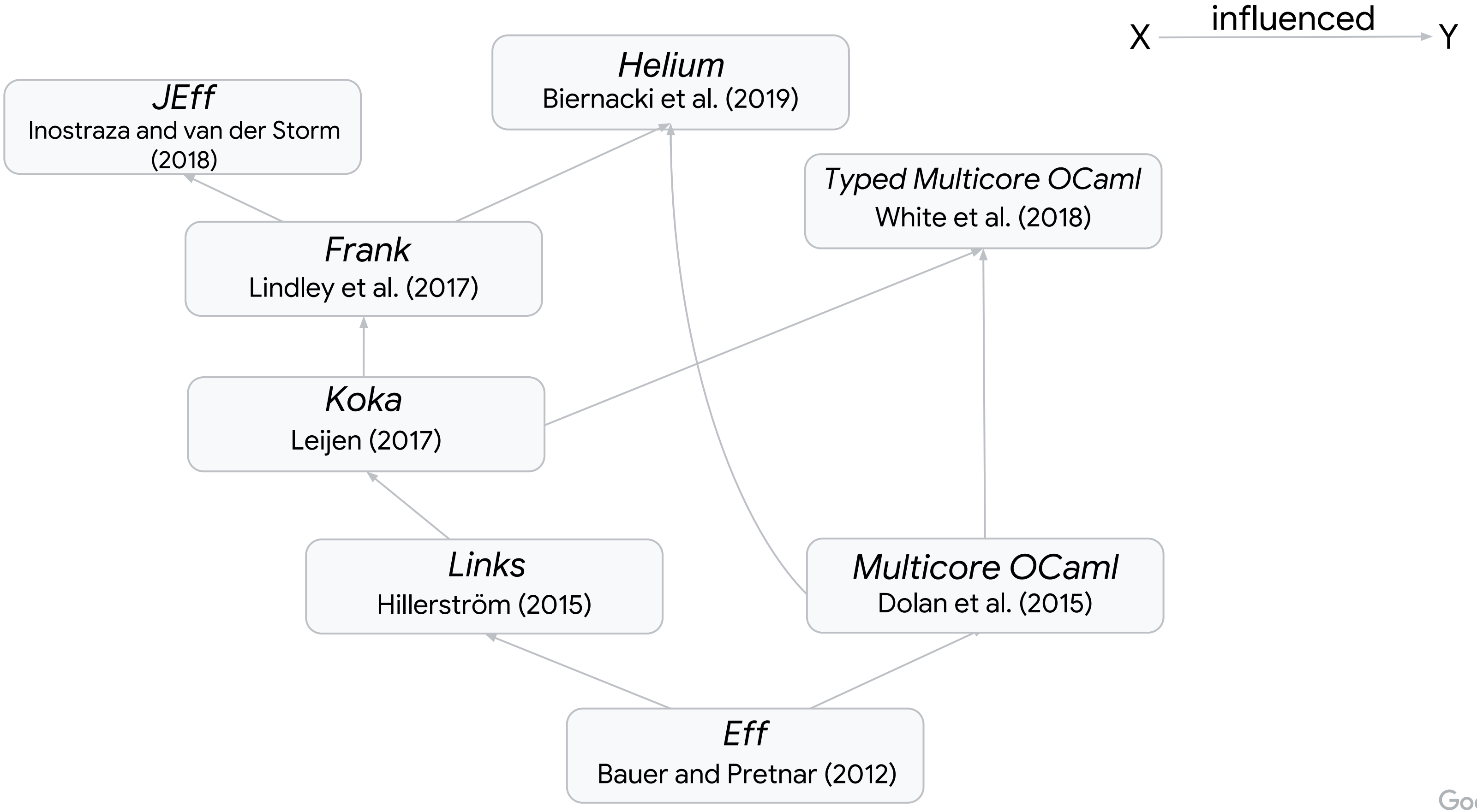
```
effect E : unit
match
  let x = match perform E with
          | effect F k -> ...
  in ...
with
| effect E k -> continue k ()
```



Generators and iterators

Files: https://github.com/dhil/google-tech-talk-2018/blob/master/live/generators.ml
https://github.com/dhil/google-tech-talk-2018/blob/master/live/pi.ml

An overview of implementations



Library implementations omitted, c.f. https://github.com/yallop/effects-bibliography

My PhD at glance

Year 0	Applications of effect handlers (wrt. parallelism and concurrency)
Year 1	Compilation strategies. Abstract machines, CPS translations.
Year 2	Expressive power.
Year 3	??? Commences once I return.

My PhD at gl

Ye

…ency)

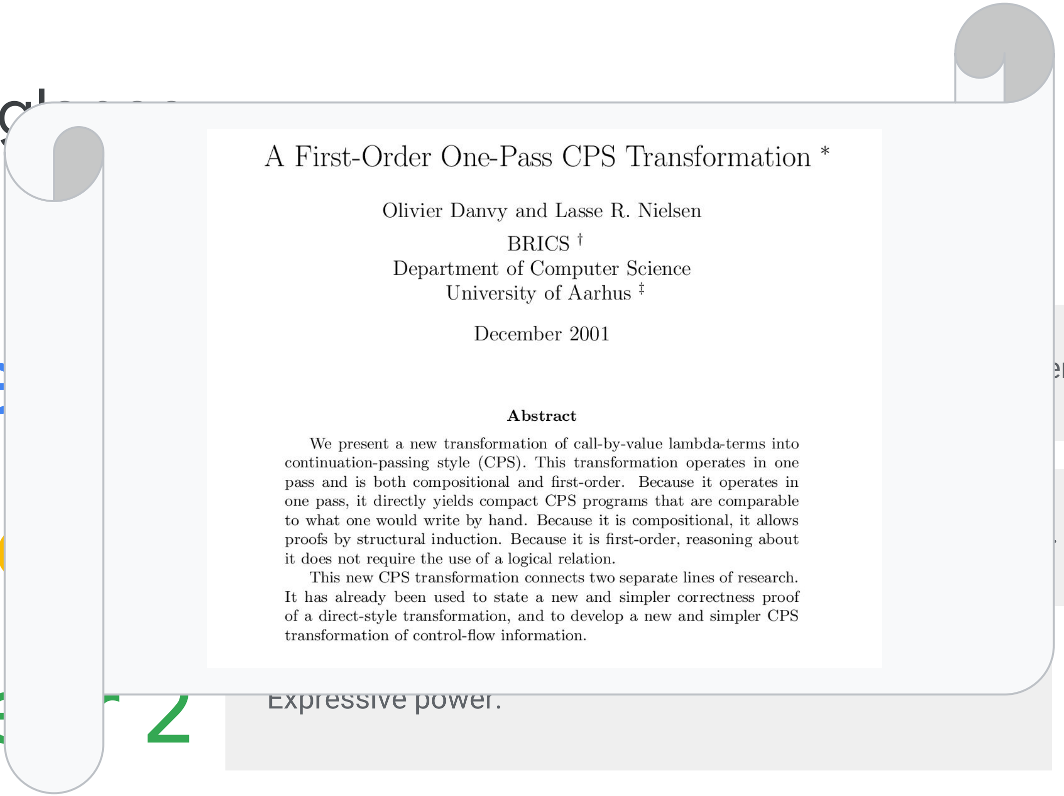
Y

Ye 2

Expressive power.

Year 3

??? Commences once I return.



A First-Order One-Pass CPS Transformation *

Olivier Danvy and Lasse R. Nielsen

BRICS †
Department of Computer Science
University of Aarhus ‡

December 2001

Abstract

We present a new transformation of call-by-value lambda-terms into continuation-passing style (CPS). This transformation operates in one pass and is both compositional and first-order. Because it operates in one pass, it directly yields compact CPS programs that are comparable to what one would write by hand. Because it is compositional, it allows proofs by structural induction. Because it is first-order, reasoning about it does not require the use of a logical relation.

This new CPS transformation connects two separate lines of research. It has already been used to state a new and simpler correctness proof of a direct-style transformation, and to develop a new and simpler CPS transformation of control-flow information.

c.f. Hillerström and Lindley (2016), Hillerström et al. (2017), and Hillerström and Lindley (2018)

My PhD at glance

Year 0	Applications of effect handlers (wrt. parallelism and concurrency)
Year 1	Compilation strategies. Abstract machines, CPS translations.
Year 2	Expressive power.
Year 3	??? Commences once I return.

Implementing asynchrony

File: https://github.com/dhil/google-tech-talk-2018/blob/master/live/async_await.ml

Google

(Semi-)Open problems I

Abstract operations are not abstracted.

```
(* Module trace.ml *)
effect Trace : unit
let trace f =
  match f (fun () -> perform Trace) with
  | result -> result
  | effect Trace k -> print_endline “Called”; continue k ()

(* Module other.ml *)
open Trace
let f g =
  match g () with
  | _ -> ()
  | effect Trace _ -> ()

let _ = trace f (* prints nothing. *)
```

Biernacki et al. (2018), Biernacki et al. (2019), Convent et al. (2018), Zhang and Myers (2019) provide potential answers.

(Semi-)Open problems II

In general, effect handlers do not interact well with resources

```
let take_while predicate file =
  let fh = open_in file in
  let rec take acc =
    try
      let line = input_line fh in
      if predicate line then
        take (line :: acc)
      else acc
    with
    | End_of_file -> acc
  in
  let lines = take [] in
  close_in fh; lines
```

```
effect Abort : 'a
let leaks ()  =
  let predicate _ = perform Abort in
  match take_while predicate "fruits.dat" with
  | result -> result
  | effect Abort _ -> [] (* leaks. *)
```

Dolan et. al (2017) and Leijen (2018) provide potential answers

(Semi-)Open problems II

In general, effect handlers do not interact well with resources

```
let take_while predicate file =
  let fh = open_in file in
  let rec take acc =
    try
      let line = input_line fh in
      if predicate line then
        take (line :: acc)
      else acc
    with
    | End_of_file -> acc
  in
  let lines = take [] in
  close_in fh; lines
```

```
effect Choose : bool
let bad_descriptor () =
  let predicate _ = perform Choose in
  match take_while predicate "fruits.dat" with
  | result -> [result]
  | effect Choose k ->
     continue k true @ continue k false
     (* bad file descriptor exception *)
```

Dolan et. al (2017) and Leijen (2018) provide potential answers

(Semi-)Open problems III

Handler-oriented programming can occur a significant overhead

Some ideas on how to eliminate the overhead:

- Alternative, more efficient runtime representations of the handler stack
- Apply fusion laws (catamorphisms/folds) [Wu and Tom Schrijvers, 2015]
- Generalise tail-call elimination to “tail-resumptive elimination” [Leijen, 2018]
- Use a substructural typing discipline to guide optimisations
- Power of JIT compilation: profile-guided optimisations at runtime? (Speculation)

Concluding remarks and the future

Summary

- Effect handlers provide an abstraction for modular effectful programming
- Contemporary control idioms are really special instances of effect handlers
- OCaml provides an industrial-strength implementation of effect handlers

Future work

- Loads of design questions (type systems, modular abstraction, etc)
- Loads of compiler questions (optimisation schemes, runtime representations, etc)
- Effect handlers as a primitive in WebAssembly?

References

References I

- Andrej Bauer and Matija Pretnar, “Programming with Algebraic Effects and Handlers”, 2012
- Nick Benton and Andrew Kennedy, “Exceptional Syntax”, 2001
- Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman, “Pyro: Deep Universal Probabilistic Programming”, 2018
- Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski, “Handle with Care: Relational Interpretation of Algebraic Effects and Handlers”, 2018
- Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski, “Abstracting Algebraic Effects”, 2019
- Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin, “Encapsulating Effects in Frank”, draft 2018
- Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy, “Effective Concurrency through Algebraic Effects”, 2015
- Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White, “Concurrent System Programming with Effect Handlers”, 2017

References II

- Daniel Hillerström, “Handlers for Algebraic Effects in Links”, 2015
- Daniel Hillerström and Sam Lindley, “Liberating Effects with Rows and Handlers”, 2016
- Daniel Hillerström, Sam Lindley, Robert Atkey, KC Sivaramakrishnan, “Continuation Passing Style for Effect Handlers”, 2017
- Daniel Hillerström and Sam Lindley, “Shallow Effect Handlers”, 2018
- Pablo Inostroza and Tijs van der Storm, “JEff: Objects for Effect”, 2018
- Ohad Kammar, Sam Lindley, and Nicolas Oury, “Handlers in Action”, 2013
- Daan Leijen, “Type Directed Compilation of Row-Typed Algebraic Effects”, 2017
- Daan Leijen, “Structured Asynchrony with Algebraic Effects”, 2017
- Daan Leijen, “Algebraic Effect Handlers with Resources and Deep Finalization”, 2018
- Sam Lindley, Conor McBride, and Craig McLaughlin, “Do be do be do”, 2017
- Gordon D. Plotkin and John Power, “Adequacy for Algebraic Effects”, 2001
- Gordon D. Plotkin and Matija Pretnar, “Handlers of Algebraic Effects”, 2009

References III

- Leo White, “Effective Programming: Adding an Effect System to OCaml”, 2018 (talk)
- Jeremy Yallop, “Staged Generic Programming”, 2017
- Yizhou Zhang and Andrew Myers, “Abstraction-Safe Effect Handlers via Tunnelling”, 2019
- Nicolas Wu and Tom Schrijvers, “Fusion for Free: Efficient Algebraic Effect Handlers”, 2015

Google