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

Compiling Links Effect Handlers to the OCaml Backend

ML Workshop '16

Daniel Hillerström [1] Sam Lindley [1] KC Sivaramakrishnan [2]

[1]The University of Edinburgh, UK

[2]University of Cambridge, UK

September 22, 2016

The Programming Language Links

Meet Links (Cooper et al., 2006)

- a ML-like strict functional programming language,
- a single-source language for multi-tier web-programming,
- with a syntax reminiscent of JavaScript, e.g. `fun foo(x,y) { ... }`,
- and a strong type system including linear types,
- with effect typing based on row polymorphism,
- and it provides *effect handlers* for controlling effects (Hillerström, 2015).

Links has three backends, each written in OCaml:

- a JavaScript compiler for the client,
- an interpreter for the server,
- and an SQL generator for the database,
- and with this work a compiler for the server.

See more at `http://www.links-lang.org`.

Algebraic Effects and Abstract Computations

An algebraic effect is a collection of *abstract operations*, e.g.

$$Nondet = \{\texttt{Choose : Bool}\}$$

Using abstract operations we can define effectful computations *abstractly*, e.g.

```
fun toss() { if (do Choose) Heads else Tails }
```

Algebraic Effects and Abstract Computations

An algebraic effect is a collection of *abstract operations*, e.g.

$$\mathit{Nondet} = \{\texttt{Choose} : \texttt{Bool}\}$$

Using abstract operations we can define effectful computations *abstractly*, e.g.

```
sig toss : () {Choose:Bool|e}-> Toss
fun toss() { if (do Choose) Heads else Tails }
```

Algebraic Effects and Abstract Computations

An algebraic effect is a collection of *abstract operations*, e.g.

$$Nondet = \{\texttt{Choose} : \texttt{Bool}\}$$

Using abstract operations we can define effectful computations *abstractly*, e.g.

```
sig toss : () {Choose:Bool|e}-> Toss
fun toss() { if (do Choose) Heads else Tails }
```

Evaluation of an abstract computation. . .

```
links> toss();
*** Error: Unhandled operation: Choose
```

. . . but, what is the semantics of `Choose`?

Abstract Operation Instantiation with Handlers

A handler instantiates abstract operations with concrete implementations, e.g.

```
handler randomResult {
  case Return(x) -> x
  case Choose(resume) -> resume(random() > 0.5)
}
```

The function `resume` is the captured (delimited) continuation of the operation.

Abstract Operation Instantiation with Handlers

A handler instantiates abstract operations with concrete implementations, e.g.

```
sig randomResult : (() {Choose:Bool|e}-> a) ->
                    () {Choose{p}  |e}-> a
handler randomResult {
  case Return(x) -> x
  case Choose(resume) -> resume(random() > 0.5)
}
```

The function `resume` is the captured (delimited) continuation of the operation.

Abstract Operation Instantiation with Handlers

A handler instantiates abstract operations with concrete implementations, e.g.

```
sig randomResult : (() {Choose:Bool|e}-> a) ->
                    () {Choose{p}  |e}-> a
handler randomResult {
  case Return(x) -> x
  case Choose(resume) -> resume(random() > 0.5)
}
```

The function `resume` is the captured (delimited) continuation of the operation.

Interpretation of `toss` with this handler:

```
links> randomResult(toss)();
Tails : Toss
```

Abstract Operation Instantiation with Handlers

A handler instantiates abstract operations with concrete implementations, e.g.

```
sig allChoices : (() {Choose:Bool|e}->  a) ->
                  () {Choose{p}  |e}-> [a]
handler allChoices {
  case Return(x) -> [x]
  case Choose(resume) -> resume(true) ++ resume(false)
}
```

The function `resume` is the captured (delimited) continuation of the operation.

Interpretation of `toss` with this handler:

```
links> allChoices(toss)();
[Heads, Tails] : [Toss]
```

Handlers can be Abstract Too

Consider the following abstract handler:

```
sig flip : (() {Choose:Bool |e}-> a) ->
            () {Choose:Bool |e}-> a)
handler flip {
  case Return(x)       -> x
  case Choose(resume) -> resume(not(do Choose))
}
```

We may use `allChoices` to interpret `flip(toss)`:

```
links> allChoices(flip(toss))();
[Tails, Heads] : [Toss]
```

Classification of Handlers

Handlers can be classified according to their continuation consumption.

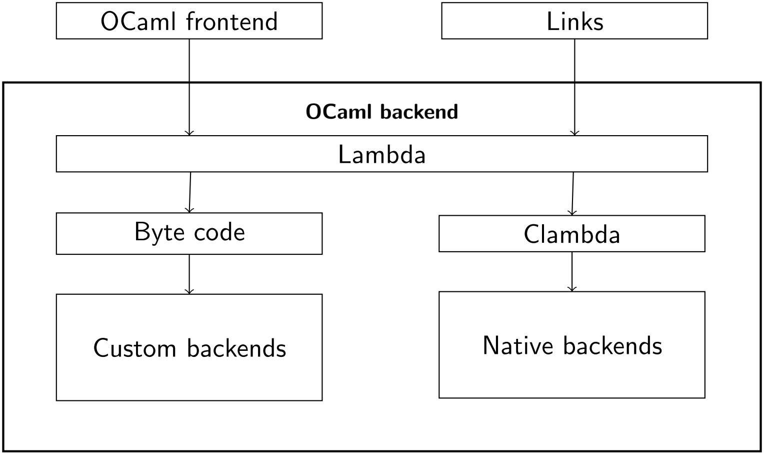
Type	Example(s)	Cont. consumption
Exception handler	`maybeResult`	0
Linear handler	`randomResult`, `flip`	1
Multi-shot handler	`allChoices`	> 1

Aim

Handlers are not only for coin tossing. In particular, we have a reconstruction of the concurrency model of Links using handlers (Hillerström, 2016).

Thus we are interested in making this abstraction *efficient* and *safe* while retaining *modularity*.

Compiler Backend



Multicore OCaml Handlers

Multicore OCaml (Dolan et al., 2015) provides

- effect handlers as an abstraction for concurrency,
- an efficient, native implementation of *linear* effect handlers,
- an explicit copying construct for on demand multi-shot handlers.

Consider the following example in Links and OCaml:

```
links> allChoices(flip(toss))()
[Tails, Heads] : [Toss]

ocaml# allChoices (flip toss) ();;
```

Multicore OCaml Handlers

Multicore OCaml (Dolan et al., 2015) provides

- effect handlers as an abstraction for concurrency,
- an efficient, native implementation of *linear* effect handlers,
- an explicit copying construct for on demand multi-shot handlers.

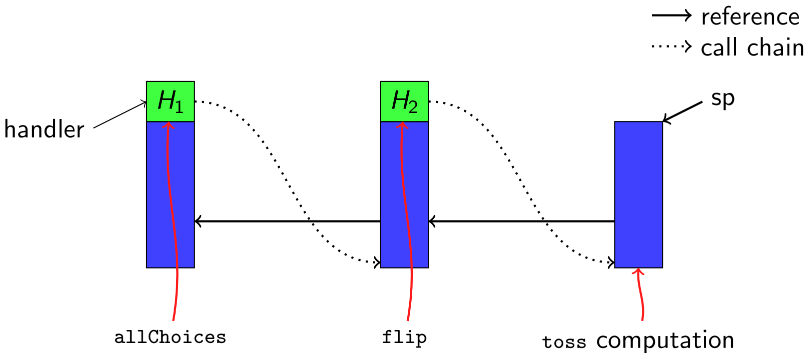
Consider the following example in Links and OCaml:

```
links> allChoices(flip(toss))()
[Tails, Heads] : [Toss]

ocaml# allChoices (flip toss) ();;
Exception: Invalid_argument "continuation already taken".
```

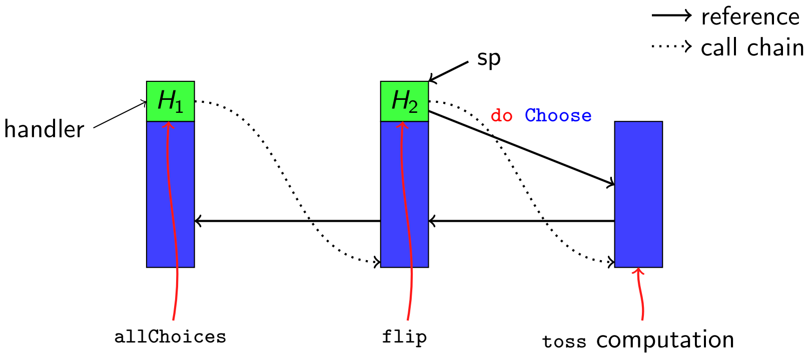
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:



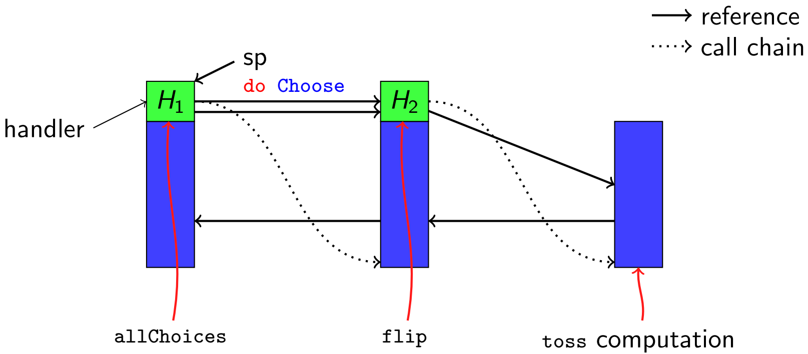
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:



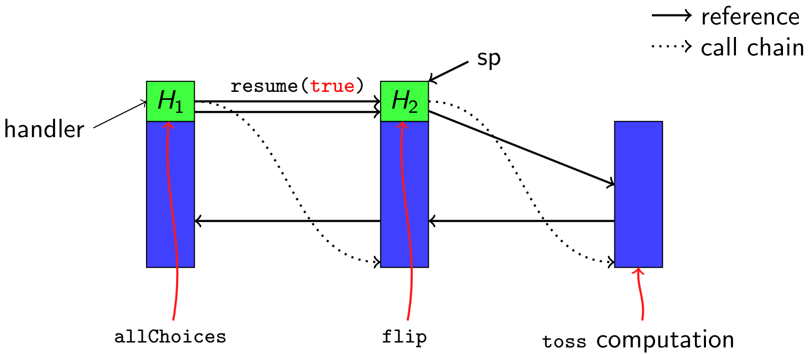
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:



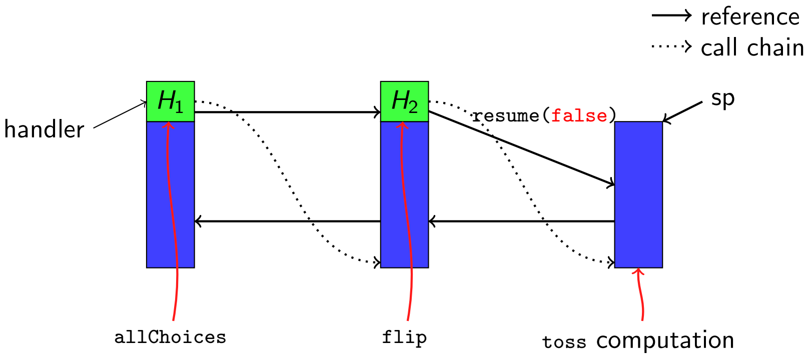
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:



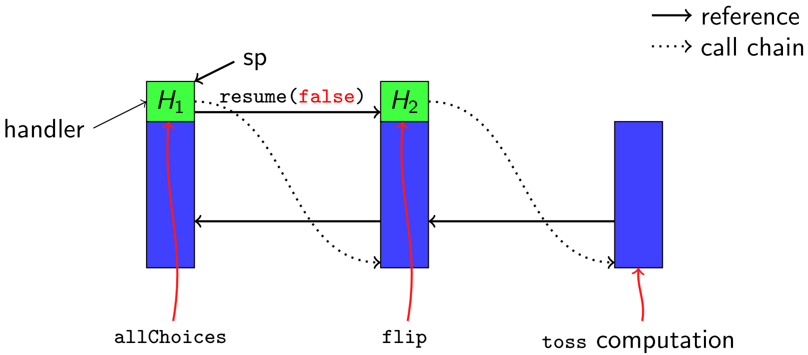
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:



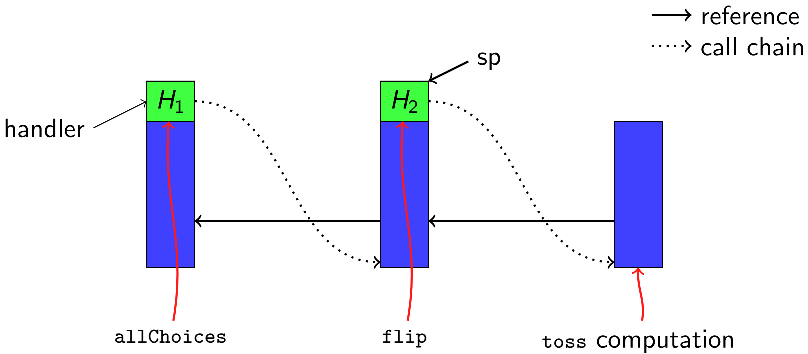
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:



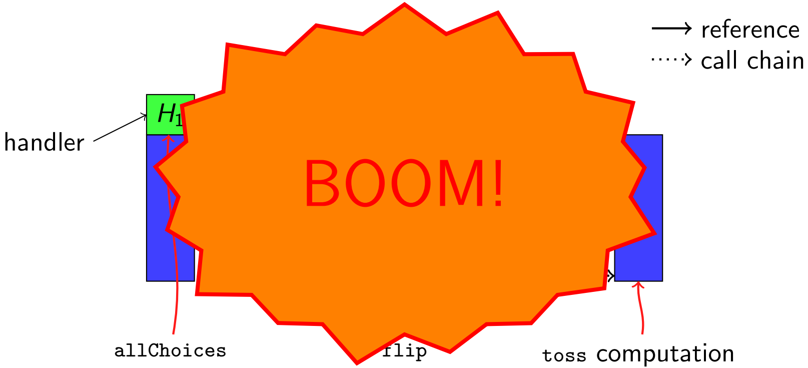
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:



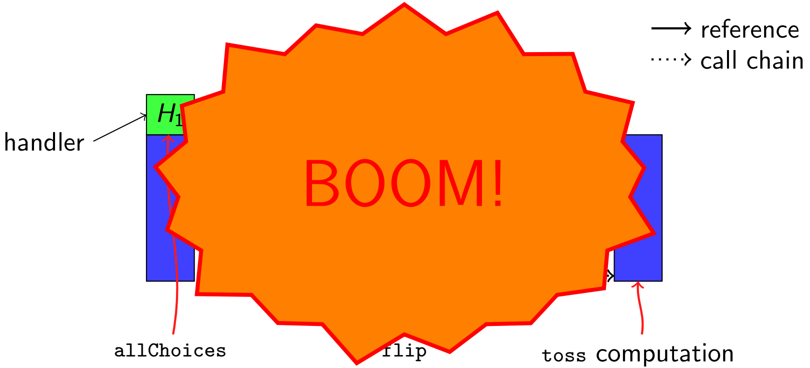
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:



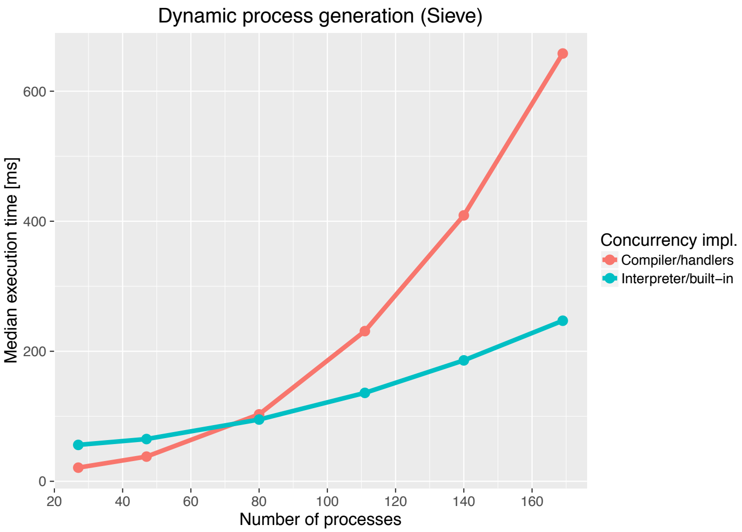
On Demand Multi-shot Handlers are a Fragile Abstraction

Runtime layout of `allChoices(flip(toss))`:

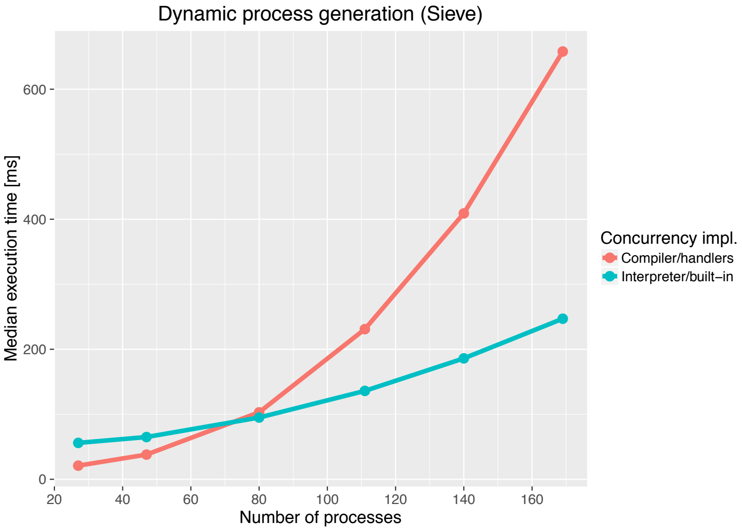


Conservative solution: implement every handler as a multi-shot handler.

What is the Penalty?

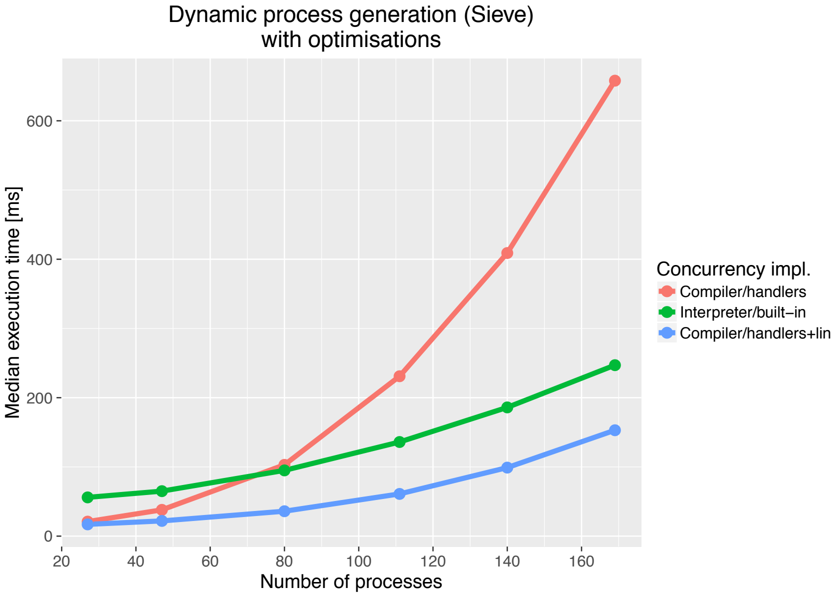


What is the Penalty?



Idea: let's use the linear type system to track the linearity of handlers.

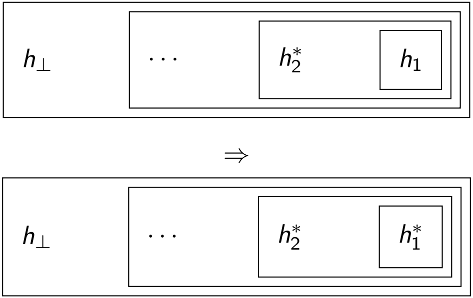
Does It Work?


Dynamic process generation (Sieve)
with optimisations
Median execution time [ms]
0
200
400
600
20
40
60
80
100
120
140
160
Number of processes
Concurrency impl.
Compiler/handlers
Interpreter/built−in
Compiler/handlers+lin

Scoping of Handler Promotion

Initial idea: use the effect system to propogate linearity information.

Ideally, we want this:

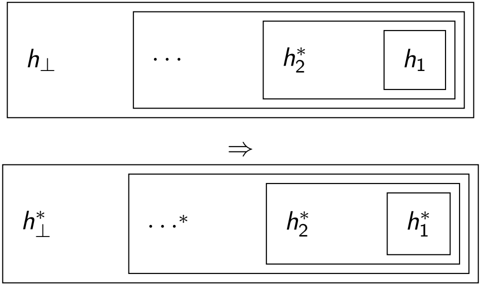


Legend	
h_n^*	Multi-shot handler
h_n	Linear handler

Scoping of Handler Promotion

Initial idea: use the effect system to propogate linearity information.

But, this is what really happens:

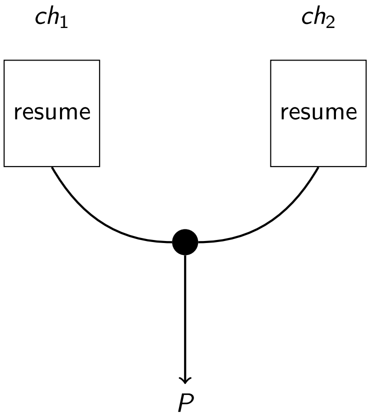


Need some way to capture the structure of the handler stack at the type-level.

Legend	
h_n^*	Multi-shot handler
h_n	Linear handler

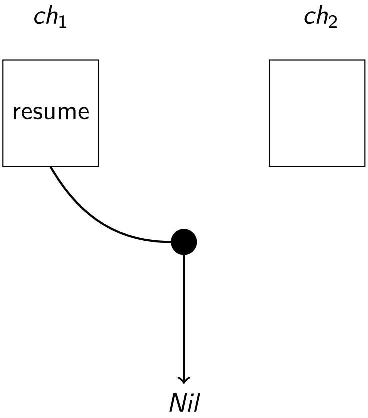
Use Case: Channel Selection

The linear type system is not expressive enough to capture tombstones.



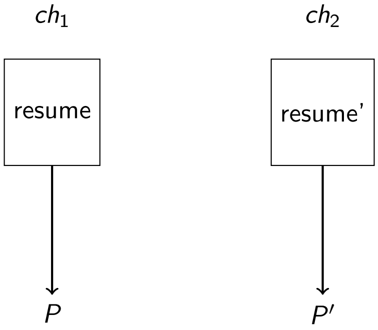
Use Case: Channel Selection

The linear type system is not expressive enough to capture tombstones.



Use Case: Channel Selection

The linear type system is not expressive enough to capture tombstones.



This is not the desired semantics!

Compiling Handlers – Related Work

Compilation options

- Continuation monad (Kammar et al., 2013)
- Free monad (Kiselyov et al., 2013; Bauer and Pretnar, 2015)
- Direct-style like Multicore OCaml (Dolan et al., 2015)
- Selective CPS translation (Leijen, 2016)
- Shift/reset control operators (Saleh and Schrijvers, 2016)

Summary

- Algebraic effects and handlers provide a modular abstraction for effectful programming.
- OCaml backend gives you a native code generator (almost) for free.
- Regard Links as an experimental frontend to OCaml with effect typing and linear types.
- Type-and-effect directed optimisations of handlers is promising.
- To capture common use cases we need a more expressive linear type system.

References I

Daniel Hillerström.
Handlers for algebraic effects in Links.
Master's thesis, School of Informatics, the University of Edinburgh, Scotland, August 2015.

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

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.

References II

Amr Hany Saleh and Tom Schrijvers.
Efficient algebraic effect handlers for Prolog.
Submitted to TPLP, 2016.

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.

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

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

References III

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.