Towards Compilation of Affine Algebraic Effects Handlers

Daniel Hillerström
daniel.hillerstrom@ed.ac.uk
http://homepages.inf.ed.ac.uk/s1467124

The University of Edinburgh

April 26, 2016

The Links language

The code examples in this talk are written in Links[1]:

- Pure, functional, web-oriented, research programming language.
- Sort of JavaScript syntax with sane semantics.
- Developed at the University of Edinburgh
- Conceived to solve the *impedance mismatch problem* in web-programming.
- Best thing about Links:

[1] ref. Cooper et al. (2006)

The Links language

The code examples in this talk are written in Links[1]:

- Pure, functional, web-oriented, research programming language.
- Sort of JavaScript syntax with sane semantics.
- Developed at the University of Edinburgh
- Conceived to solve the *impedance mismatch problem* in web-programming.
- Best thing about Links: **It has no users**

[1] ref. Cooper et al. (2006)

Programs are effectful

Virtually, every program comprise an effectful component, e.g.

- raise exceptions
- perform input/output
- mutate some state
- fork threads
- non-determinism
- ...and so forth

In most programming languages effects are dealt with *implicitly*. Algebraic effects and handlers provide a modular abstraction for modelling and controlling effects *explicitly*.

Algebraic effects by example: A coin toss[2]

Algebraic effects

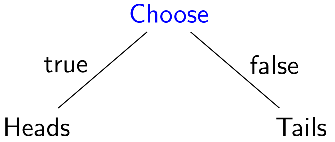
An algebraic effect is a collection of abstract operations.

For example, nondeterminism is given by a single operation
$nondet = \{Choose : \text{Bool}\}$

An effectful coin toss:

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

Visualised as a computation tree:



[2] The example is adopted from Kammar et al. (2013)

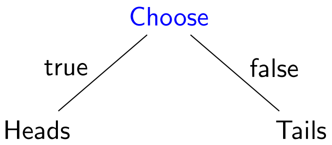
Effect handlers by example: A coin toss

Handlers

A handler instantiates abstract operations with a concrete implementation.

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

handler alwaysHeads {
  case Choose(k) -> k(true)
  case Return(x) -> x
}
```



Here `k` is the continuation of `do Choose`.
The result of `alwaysHeads(toss)` is `Heads`.

Project overview

I'm interested in making effect handlers a practical programming model.

- Phase 1 Front-end: handlers and row types[3] ✓
- Phase 2 Back-end: compile handlers to efficient, native code.
- Phase 3 Rebuild Links' concurrency model in terms of handlers

Continuations are the main performance bottleneck. OCaml multicore[4] provides an efficient implementation of *linear* handlers. My plan is to translate Links IR to OCaml Lambda IR.

[3] c.f. Hillerström and Lindley (2016)

[4] ref. Dolan et al. (2015)

Categorising handlers

Exception[5]	`handler maybeResult {`
	`  case Fail(k)   -> Nothing`
	`  case Return(x) -> Just(x)`
	`}`

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

Multi-shot	`handler allResults {`
	`  case Choose(k) -> k(true) ++ k(false)`
	`  case Return(x) -> [x]`
	`}`

[5] where *exception* = {*Fail* : *Void*}

Categorising handlers

Exception[5]	`handler maybeResult {`
	`  case Fail(k)   -> Nothing`
	`  case Return(x) -> Just(x)`
	`}`

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

Multi-shot	`handler allResults {`
	`  case Choose(k) -> k(true) ++ k(false)`
	`  case Return(x) -> [x]`
	`}`

Affine handlers invoke their continuations at most once.
Idea: Use the type system to track the nature of handlers, and specialise the run-time implementations during code generation.

[5] where $\mathit{exception} = \{\mathit{Fail} : \mathit{Void}\}$

Composing handlers by example: Drunk coin toss

Consider a drunkard tossing a coin[6]:

```
fun drunkToss() {
  if (do Choose) toss()
  else do Fail
}
```

We may compose handlers to fully interpret `drunkToss`:
`randomResult(maybeResult(drunkToss))`.

Possible outcomes: {`Just(Heads)`,`Just(Tails)`,`Nothing`}.

[6] Technical detail: `switch(do Fail) { }` required for example to type check.

Runtime stack of handlers

Composition gives rise to stack of handlers at runtime:

`randomResult(maybeResult(drunkToss))`

maybeResult
randomResult
$\bot$

Handling `Choose` in `drunkToss` causes the stack to be unwinded.

Optimisations

The stack representation is simple, but inefficient for large compositions.
OCaml does not perform optimisations for handlers.

Solution: Rediscover classical optimisations in the context of handlers:

- Fusion
- Inlining
- Reordering of handlers

Optimisation: Fusion

Criterion for handler fusion

If two adjacent handlers handle a disjoint set of operations, then they can be fused.

```
handler maybeResult {
  case Fail(k)   -> Nothing
  case Return(x) -> Just(x)
}

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

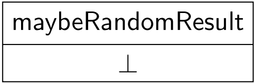
maybeResult
randomResult
$\perp$

Optimisation: Fusion

Criterion for handler fusion

If two adjacent handlers handle a disjoint set of operations, then they can be fused.

```
handler maybeRandomResult {
  case Fail(k)   -> Nothing
  case Choose(k) -> k(random() > 0.5)
  case Return(x) -> var y = Just(x); y
}
```



Optimisation: Inlining

Conservative criteria for handler inlining

A linear handlers can be inlined if[a]

- It invokes continuations in tail-position
- The handler is the top-element ($\top$)

[a] sometimes we can relax these criteria

```
handler maybeResult {
  case Fail(k)   -> Nothing
  case Return(x) -> Just(x)
}

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

```
randomResult(
 maybeResult(
    fun() {
     if (do Choose) toss()
     else do Fail
    }))
```

Optimisation: Inlining

Conservative criteria for handler inlining

A linear handlers can be inlined if[a]

- It invokes continuations in tail-position
- The handler is the top-element ($\top$)

[a] sometimes we can relax these criteria

```
handler maybeResult {
  case Fail(k)   -> Nothing
  case Return(x) -> Just(x)
}

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

```
randomResult(
 maybeResult(
    fun() {
     if (do Choose) toss()
     else do Fail
    }))
```

Cannot inline `maybeResult`: it is not linear

Optimisation: Inlining

Conservative criteria for handler inlining

A linear handlers can be inlined if[a]

- It invokes continuations in tail-position
- The handler is the top-element ($\top$)

[a] sometimes we can relax these criteria

```
handler maybeResult {
  case Fail(k)   -> Nothing
  case Return(x) -> Just(x)
}

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

```
randomResult(
 maybeResult(
    fun() {
     if (do Choose) toss()
     else do Fail
    }))
```

Cannot inline linear `randomResult`: it is not $\top$

Optimisation: Inlining

Conservative criteria for handler inlining

A linear handlers can be inlined if[a]

- It invokes continuations in tail-position
- The handler is the top-element ($\top$)

[a] sometimes we can relax these criteria

```
handler maybeResult {
  case Fail(k)   -> Nothing
  case Return(x) -> Just(x)
}

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

```
randomResult(
 maybeResult(
    fun() {
     if (do Choose) toss()
     else do Fail
    }))
```

Cannot inline linear `randomResult`: it is not $\top$

If we reorder the two handlers, then we can inline `randomResult`

Optimisation: Inlining

Conservative criteria for handler inlining

A linear handlers can be inlined if[a]

- It invokes continuations in tail-position
- The handler is the top-element ($\top$)

[a] sometimes we can relax these criteria

```
handler maybeResult {
  case Fail(k)   -> Nothing
  case Return(x) -> Just(x)
}

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

```
maybeResult(
  randomResult(
    fun() {
     if (do Choose) toss()
     else do Fail
    }))
```

Optimisation: Inlining

Conservative criteria for handler inlining

A linear handlers can be inlined if[a]

- It invokes continuations in tail-position
- The handler is the top-element ($\top$)

[a] sometimes we can relax these criteria

```
handler maybeResult {
  case Fail(k)   -> Nothing
  case Return(x) -> Just(x)
}
```

```
maybeResult(
 fun() {
  if (random() > 0.5)
   toss()[random()>0.5/do Choose]
  else do Fail
 }))
```

Summary

- Handlers provide a great abstraction for generic programming.
- I get native baseline performance for free from OCaml.
- Classical optimisation techniques provide a first good attempt at optimising handlers.

References

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. 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. URL `http://dx.doi.org/10.1007/978-3-540-74792-5_12`.

S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects, 9 2015. OCaml Workshop.

D. Hillerström and S. Lindley. Liberating effects with rows and handlers. Submitted, draft, March 2016.

O. Kammar, S. Lindley, and N. 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. ISBN 978-1-4503-2326-0. URL `http://doi.acm.org/10.1145/2500365.2500590`.