Effect Handlers, Evidently

Daniel Hillerström

MSR Intern, Redmond, WA, USA

PhD student, The University of Edinburgh, Scotland, UK

I have a dream...

King County

I have a dream...

I have a dream...

King County

I have a dream...

I have a dream...

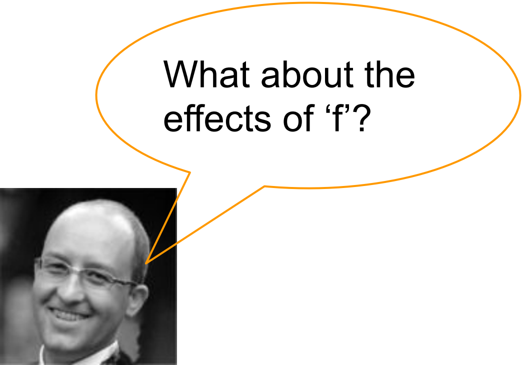
King County

I also have a dream...

Me too!

The dream of robust programming

```
f : a -> b
```

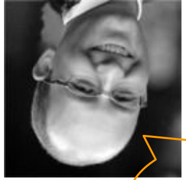


The dream of robust programming

$$f : a \rightarrow \varepsilon\ b$$

some effect signature

The dream of robust programming



Leijen-style row polymorphism

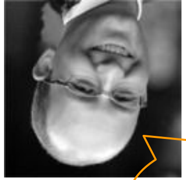
f : a -> ε b

some effect signature

Signatures

```
print         : string -> <io|e> ()
read-line     : ()     -> <io|e> string
string-to-int : string -> <exn|e> int
forever       : (() -> e ()) -> <div|e> ()
```

The dream of robust programming



Leijen-style row polymorphism

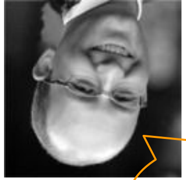
$$f : a \rightarrow \varepsilon\ b$$

some effect signature

Signatures

```
print         : string -> io ()
read-line     : ()     -> io string
string-to-int : string -> exn int
forever       : (() -> ()) -> div ()
```

The dream of robust programming



Leijen-style row polymorphism

```
f : a -> ε b
```

some effect signature

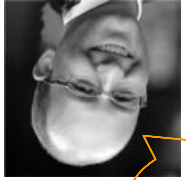
Signatures

```
print         : string -> io ()
read-line     : ()     -> io string
string-to-int : string -> exn int
forever       : (() -> ()) -> div ()
```

Composing multiple effects

```
echo-int() : <div, exn, io> () {
  forever({
     print(
       string-to-int(read-line()))
  })
}
```

The dream of robust programming



Leijen-style row polymorphism

$$f : a \rightarrow \varepsilon \; b$$

some effect signature

Signatures

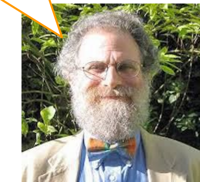
```
pri             string -> io ()
                       -> io string
                  ring -> exn int
f/               (() -> ()) -> div ()
```

No, no. Use effect handlers! They compose!

Composing multiple effects

```
echo-int() : <div, exn, i
  forever({
     print(
       string-to-int(read-li
  })
}
```

This not ergonomic. Have you considered monads?



Effect handlers (Plotkin & Pretnar, 2009)

- Captures control idioms uniformly
- Can implement any (algebraic) effects
- Practical programming abstraction based on strong mathematical foundations
  - Generators and iterators (Leijen, 2017a)
  - Async/await (Dolan et al., 2017 and Leijen, 2017b)
  - Co-routines (Kiselyov et al., 2013)
  - Deep learning (Bingham et al., 2018)
  - Multi-stage programming (Yallop, 2017)
  - Parsing (Wu et al., 2014)
  - Modular program construction (Kammar et al., 2013)

Effect handlers (Plotkin & Pretnar, 2009)

- Captures control idioms uniformly
- Can implement any (algebraic) effects
- Practical programming abstraction based on strong mathematical foundations

Problem: state-of-the-art implementations are inefficient (always linear search),
can we do better? (open question)
This summer: try to make handlers as cheap as a virtual method call

  - Deep learning (Bingham et al., 2018)
  - Multi-stage programming (Yallop, 2017)
  - Parsing (Wu et al., 2014)
  - Modular program construction (Kammar et al., 2013)

Example: Read-only state

The interface

```
effect reader {
  fun ask() : int
}
```

Programming against the interface

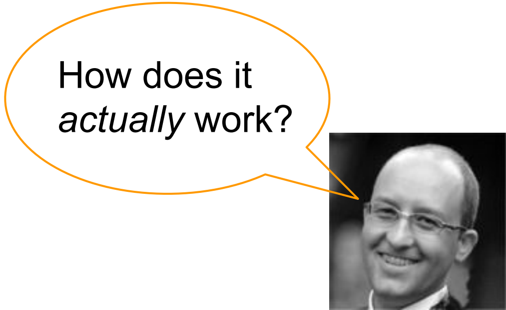
```
fun add() : reader int {
  ask() + ask()
}
```

The implementation

```
fun reader(f : () -> reader a) : a {
  handle(f) {
    return x -> x
    ask()     -> resume(2) // ask is 2
} }
```

Run the computation

```
reader(add) → 4
```



Example: Read-only state

```
handle({ ask() + ask() }) {
  return x -> x
  ask()    -> resume(2) // ask is 2
}
```

Example: Read-only state

```
handle({ ask() + ask() }) {
  return x -> x
  ask()    -> resume(2) // ask is 2
}
```

Example: Read-only state

```
handle({     2 + ask() }) {
  return x -> x
  ask()    -> resume(2) // ask is 2
}
```

Example: Read-only state

```
handle({     2 + ask() }) {
  return x -> x
  ask()    -> resume(2) // ask is 2
}
```

Example: Read-only state

```
handle({     2 +     2 }) {
  return x -> x
  ask()    -> resume(2) // ask is 2
}
```

Example: Read-only state

```
handle({       4       }) {
  return x -> x
  ask()    -> resume(2) // ask is 2
}
```

Example: Read-only state

```
handle({        4        }) {
  return x -> x
  ask()    -> resume(2) // ask is 2
}
```

Once done, it transfers control to the return-clause.

Return may be viewed as a special operation `return<b>(x : a) : b`

Example: Read-only state

```
handle(                    ) {
   return x -> 4
   ask()    -> resume(2) // ask is 2
}
```

Once done, it transfers control to the return-clause.

Return may be viewed as a special operation `return<b>(x : a) : b`

Example: Read-only state

4

Operational semantics

handle(E[op V]) H → N{x -> V, r -> λy.**handle**(E[y]) H}
  if H^{op} = (op x r -> N) and inner-most(H, E, op)

handle(V) H → N{x -> V}
  if H^{ret} = (return x -> N)

Example: Generators and Iterators

The interface

```
effect gen {
  fun yield(x : int) : ()
}
```

Programming against the interface

```
fun range(a : int, b : int) : <div,gen> () {
 if (a > b) then ()
 else { yield(a)
        range(a+1, b) }
}
```

The implementation

```
fun for-each(g : ()    -> gen ()
            ,f : (int) ->     ()) : () {
  handle(g) {
    return x -> ()
    yield(x) -> resume(f(x))
} }
```

```
for-each(print-int, {range(0, 3)})
```

Prints 0123

Example: State

The interface

```
effect state {
  fun get() : int
  fun set(s : int) : ()
}
```

Programming against the interface

```
fun state-example() : int {
  fun add() : state int {
     val a = get()
     set(40)
     a + get()
  }
  run-state(add, 2)
}
```

The implementation

```
fun run-state(f : () -> state a, v : int) : a {
  var s := v; // local reference cell.
  handle(f) {
     return x   -> x
     get()      -> resume(s)
     set(s-new) -> {
       s := s-new
       resume(())
} }  }
```

Lexically scoped state!
Manipulated via a
structured interface.

Example: Co-routines as a library

```
effect coop {
  fun yield() : ()
}

rectype co {
  Co(p: ((), list<co>) -> ())
}


fun coop-with(p) {
  Co(fun(_, ps) { coop(p, ps) }
})


fun cooperate(rs) {
  coop({ () }, map(coop-with, rs))
}
```

```
fun coop(p : () -> coop (), ps : list<co>) : () {
  handle(p)(rs = ps) {
   return x ->
     match(rs) {
       Nil -> ()
       Cons(Co(r), rs0) -> r((), rs0)
     }
   yield() ->
     match(rs) {
       Nil -> ()
       Cons(Co(r), rs0) -> r((), rs0 + [Co(resume)])
} } }
```

```
fun coop-with(p : () -> <coop> ()) : co {
  Co(fun(_ : (), ps : list<co>) {
          coop(p, ps)
  })
}
```

Effects combine seamlessly

```
f : () -> <coop,state> ()
g : () -> <coop,gen,reader> ()
k : (int) -> <coop,reader> ()

fun do-something() : () {
  cooperate([{run-state(f)}
            ,{reader({for-each(k,g)})}
            ])
}
```

The problem

- Effect handlers are typically implemented like exception handlers.
  Simple, but does not scale performance-wise.

```
reader({
  H1({
     H2({...
        Hn({ask()})
       ...})
      })
  })
})
```

Runtime stack

reader
H1
H2
...
Hn

Yikes! Jumping through n hoops is expensive!

We must be able to do better...

Idea: Statically bind operations to their handlers

What if we push the handlers downwards to the invocation sites of operations?

```
reader({
  Hn({...
     H2({
       H1({ask ev-reader ()})
      })
     ...})
  })
})
```

Can we do an evidence-passing translation of effect handlers? (inspiration (Kaes, 1988))

The gist of the proposed translation

```
f : () -> <state, gen> () ~> f' : (st : ev-st, g : ev-gen) -> ()


handle ({handle f H_st}) H_gen ~> run-handler H_gen'
                                    (λev-gen.
                                      run-handler H_st'
                                        (λev-st. f'ev-st ev-gen))
```

Necessity of stack unwinding

An immediate observation is stack unwinding is necessary in the general case.

Different kind of cases:

```
●  abort ()      -> ()                                // discards the resumption
●  choose()      -> resume(True) ++ resume(False)    // multi-shot resumptions
●  count ()      -> resume(()) + 1                    // resume is not in tail position
●  put   (s-new) -> s := s-new; resume(())            // tail-resumptive
```

Unnecessary to unwind the stack in the latter case.

Many interesting effects have tail-resumptive implementations

Closing over evidence

Another observation is that evidence-passing cannot work in general.
Case: escaping resumptions.

```
f : () -> reader ()

val r = handle(f) {
     return x -> Nothing
     ask() -> Just(resume)
}
match(r) {
 Nothing -> Nothing
 Just(resume) ->
   handle({resume(42)}) {
    return x -> Nothing
    ask()    -> Just(0)
} }
```

Closing over evidence

Another observation is that evidence-passing cannot work in general.
Case: escaping resumptions.

```
f : () -> reader ()

val r = handle(f) {
     return x -> Nothing
     ask() -> Just(resume)
}
match(r) {
 Nothing -> Nothing
 Just(resume) ->
   handle({resume(42)}) {
    return x -> Nothing
    ask()    -> Just(0)
} }
```

Overloading of ‘ask’

Closing over evidence

Another observation is that evidence-passing cannot work in general.
Case: escaping resumptions.

```
f : () -> reader ()

val r = handle(f) {
     return x -> Nothing
     ask() -> Just(resume)
}
match(r) {
 Nothing -> Nothing
 Just(resume) ->
   handle({resume(42)}) {
    return x -> Nothing
    ask()    -> Just(0)
} }
```

Overloading of ‘ask’

‘resume’ is already closed over the evidence

Resumptions must not escape their handlers. We need a lexical scope restriction!

Type-directed evidence-passing translation I

$$\frac{\varnothing \vdash e : \sigma_2 \mid \varepsilon \leadsto e' : \sigma_2' \mid P \qquad P' := \mathrm{order}(P, \varepsilon)}{\vdash_{val} \lambda x^{\sigma 1}.\ e : \sigma_1 \to \varepsilon\ \sigma_2 \mid \varepsilon \leadsto \underline{\lambda P'.}\lambda x^{[\![\sigma 1]\!]} : P' \Rightarrow [\![\sigma_1]\!] \to \varepsilon\ \sigma_2'}\ [abs]$$

$$\frac{\begin{array}{l} P_1 \vdash e_1 : \sigma_1 \to \varepsilon\ \sigma \mid \varepsilon \leadsto e_1' : \sigma_1' \to \varepsilon\ \sigma' \mid P_2 \qquad (P_3, e_1'') := \mathrm{dispatch}(e_1', P_2, \varepsilon) \\ P_3 \vdash e_2 : \sigma_1 \mid \varepsilon \leadsto e_2' : \sigma_1' \mid P_4 \end{array}}{P_1 \vdash e_1\ e_2 : \sigma \mid \varepsilon \leadsto e_1''\ e_2' : \sigma' \mid P_4}\ [app]$$

It isn't quite right.

Bad example

```
fun do-something() : reader () { … }
fun invoke-return(f : () -> e ()) : (() -> e ()) { f(); f }

do-something ~> do-something’(ev : ev-reader) { … }
invoke-return ~> invoke-return

invoke-return(do-something’) // ill-typed...
```

Bad example

```
fun do-something() : reader () { … }
fun invoke-return(f : () -> e ()) : (() -> e ()) { f(); f }
```

when polymorphic in effects

Can we characterise when the translation goes wrong?

Bad example

```
fun do-something() : reader () { … }
fun invoke-return(f : () -> e ()) : (() -> e ()) { f(); f }
```

when polymorphic in effects

Can we characterise when the translation goes wrong?

Works well for known effects, e.g. <reader,state,coop|e>

Another translation

Every potentially effectful function takes an additional argument, for example

```
f : (int) -> <state,reader|e> bool ~> f' : (ev-dict, int) -> bool

map : ((a) -> e b), list<a>) -> e list<b>
  ~> map' : (ev-dict, (ev-dict, a) -> b, list<a>) -> list<b>
```

Improves on the dynamic lookup, but still not great

```
fun add'(ev-dict : ev) {
  ev-dict[ask] () + ev-dict[ask] () // O(lg |ev-dict|) lookup time
}
```

However, the literature to the rescue:
Ohori (1992), Gaster & Jones (1996), Leijen (2005), Blume et al. (2007), etc...

Summary

- Effect handlers capture many contemporary control idioms uniformly
- Evidence-passing translation of effect handlers seems to be possible*
  * if we restrict the expressiveness.
- Effect types guide the translation

- A refined implementation of the first translation in Koka
- Still need to work out the full metatheory for the second translation (and fully implement it)
- Prototype target language in Haskell
- Better understanding of the ‘unnecessary’ power of effect handlers

Thanks, I had a great summer