Automatic Differentiation via Algebraic Effects and Handlers

Jesse Sigal

LFCS
School of Informatics
University of Edinburgh

30 June 2020

Overview of this talk

Overview of this talk

- An introduction to Koka via examples.

Overview of this talk

- An introduction to Koka via examples.
- How to make smooth functions an effect.

Overview of this talk

- An introduction to Koka via examples.
- How to make smooth functions an effect.
- Koka's effect type system.

Overview of this talk

- An introduction to Koka via examples.
- How to make smooth functions an effect.
- Koka's effect type system.
- Forward mode and perturbation confusion.

Overview of this talk

- An introduction to Koka via examples.
- How to make smooth functions an effect.
- Koka's effect type system.
- Forward mode and perturbation confusion.
- Micro-benchmarks.

Overview of this talk

- An introduction to Koka via examples.
- How to make smooth functions an effect.
- Koka's effect type system.
- Forward mode and perturbation confusion.
- Micro-benchmarks.
- Reverse mode and more benchmarking.

Thank you to...

Thank you to...

- Gordon Plotkin and Matija Pretnar for inventing algebraic effects and handlers, many others for working in the area since.

Thank you to...

- Gordon Plotkin and Matija Pretnar for inventing algebraic effects and handlers, many others for working in the area since.
- Daan Leijen and others for making Koka.

An introduction to effects and handlers with Koka

An introduction to effects and handlers with Koka

```
effect ctl exception(info : string) : a
```

An introduction to effects and handlers with Koka

```
effect ctl exception(info : string) : a

fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y
```

```
effect ctl exception(info : string) : a

fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y

fun main()
  (
    handler
      ctl exception(info) ->
        print("Exception: " ++ info)
  ) {
```

```
effect ctl exception(info : string) : a

fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y

fun main()
  (
    handler
      ctl exception(info) ->
        print("Exception: " ++ info)
  ) {
    val a = divide(4, 2)
    print("'a' is " ++ show(a) ++ ", ")
    val b = divide(4, 0)
    print("'b' is " ++ show(b))
  }
```

Running our program

```
fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y

fun main()
  (
    handler
      ctl exception(info) ->
        print("Exception: " ++ info)
  ) {
    val a = divide(4, 2)
    print("'a' is " ++ show(a) ++ ", ")
    val b = divide(4, 0)
    print("'b' is " ++ show(b))
  }
```

Output:

Running our program

```
fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y

fun main()
  (
    handler
      ctl exception(info) ->
        print("Exception: " ++ info)
  ) {
    val a = divide(4, 2)
    print("'a' is " ++ show(a) ++ ", ")
    val b = divide(4, 0)
    print("'b' is " ++ show(b))
  }
```

Output:

Running our program

```
fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y

fun main()
  (
    handler
      ctl exception(info) ->
        print("Exception: " ++ info)
  ) {
    val a = divide(4, 2)
    print("'a' is " ++ show(a) ++ ", ")
    val b = divide(4, 0)
    print("'b' is " ++ show(b))
  }
```

Output:

Running our program

```
fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y

fun main()
  (
    handler
      ctl exception(info) ->
        print("Exception: " ++ info)
  ) {
    val a = divide(4, 2)
    print("'a' is " ++ show(a) ++ ", ")
    val b = divide(4, 0)
    print("'b' is " ++ show(b))
  }
```

Output:

Running our program

```
fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y

fun main()
  (
    handler
      ctl exception(info) ->
        print("Exception: " ++ info)
  ) {
    val a = divide(4, 2)
    print("'a' is " ++ show(a) ++ ", ")
    val b = divide(4, 0)
    print("'b' is " ++ show(b))
  }
```

Output: 'a' is 2,

Running our program

```
fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y

fun main()
  (
    handler
      ctl exception(info) ->
        print("Exception: " ++ info)
  ) {
    val a = divide(4, 2)
    print("'a' is " ++ show(a) ++ ", ")
    val b = divide(4, 0)
    print("'b' is " ++ show(b))
  }
```

Output: ’a’ is 2,

Running our program

```
fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y

fun main()
  (
    handler
      ctl exception(info) ->
        print("Exception: " ++ info)
  ) {
    val a = divide(4, 2)
    print("'a' is " ++ show(a) ++ ", ")
    val b = divide(4, 0)
    print("'b' is " ++ show(b))
  }
```

Output: ’a’ is 2,

Running our program

```
fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y

fun main()
  (
    handler
      ctl exception(info) ->
        print("Exception: " ++ info)
  ) {
    val a = divide(4, 2)
    print("'a' is " ++ show(a) ++ ", ")
    val b = divide(4, 0)
    print("'b' is " ++ show(b))
  }
```

Output: 'a' is 2,

Running our program

```
fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y

fun main()
  (
    handler
      ctl exception(info) ->
        print("Exception: " ++ info)
  ) {
    val a = divide(4, 2)
    print("'a' is " ++ show(a) ++ ", ")
    val b = divide(4, 0)
    print("'b' is " ++ show(b))
  }
```

Output: `'a' is 2, Exception: Divided by zero`

Handlers are first class in Koka

Handlers are first class in Koka

```
effect ctl exception(info : string) : a

val print-exception = handler
  ctl exception(info) ->
    print("Exception: " ++ info)

fun divide(x, y)
  if y == 0 then exception("Divided by zero") else x / y

fun main()
  with print-exception
  val a = divide(4, 2)
  print("'a' is " ++ show(a) ++ ", ")
  val b = divide(4, 0)
  print("'b' is " ++ show(b))
```

Handlers can resume execution

Handlers can resume execution

```
effect<a> ctl default() : a

fun main()
  with ctl default() ->
    resume(1)
  val a = default()
  val b = default()
  print(show(a : int) ++ ", " ++ show(b : int))
```

Output:

Handlers can resume execution

```
effect<a> ctl default() : a

fun main()
  with ctl default() ->
    resume(1)
  val a = default()
  val b = default()
  print(show(a : int) ++ ", " ++ show(b : int))
```

Output:

Handlers can resume execution

```
effect<a> ctl default() : a

fun main()
  with ctl default() ->
    resume(1)
  val a = default()
  val b = default()
  print(show(a : int) ++ ", " ++ show(b : int))
```

Output:

Handlers can resume execution

```
effect<a> ctl default() : a

fun main()
  with ctl default() ->
    resume(1)
  val a = default()
  val b = default()
  print(show(a : int) ++ ", " ++ show(b : int))
```

Output:

Handlers can resume execution

```
effect<a> ctl default() : a

fun main()
  with ctl default() ->
    resume(1)
  val a = default()
  val b = default()
  print(show(a : int) ++ ", " ++ show(b : int))
```

Output:

Handlers can resume execution

```
effect<a> ctl default() : a

fun main()
  with ctl default() ->
    resume(1)
  val a = default()
  val b = default()
  print(show(a : int) ++ ", " ++ show(b : int))
```

Output:

Handlers can resume execution

```
effect<a> ctl default() : a

fun main()
  with ctl default() ->
    resume(1)
  val a = default()
  val b = default()
  print(show(a : int) ++ ", " ++ show(b : int))
```

Output: 1, 1

Smooth functions as an effect

Smooth functions as an effect

```
type nullary {Const(x : float64)}
type unary {Negate}
type binary {Plus; Times}

effect smooth<a>
  ctl ap0(n : nullary) : a
  ctl ap1(u : unary, arg : a) : a
  ctl ap2(b : binary, arg1 : a, arg2 : a) : a
```

Helper functions

Helper functions

```
inline fun c(i)
  ap0(Const(i))

inline fun (~.)(x)
  ap1(Negate, x)

inline fun (+.)(x, y)
  ap2(Plus, x, y)

inline fun (*.)(x, y)
  ap2(Times, x, y)
```

Evaluating expressions

Evaluating expressions

```
val evaluate = handler
  ctl ap0(n) -> match(n) {Const (i) -> resume(i)}
  ctl ap1(u,x) -> match(u) {Negate -> resume(~x : float64)}
  ctl ap2(b,x,y) -> match(b)
    Plus -> resume(x + y : float64)
    Times -> resume(x * y : float64)
```

Evaluating expressions

```
val evaluate = handler
  ctl ap0(n) -> match(n) {Const (i) -> resume(i)}
  ctl ap1(u,x) -> match(u) {Negate -> resume(~x : float64)}
  ctl ap2(b,x,y) -> match(b)
    Plus -> resume(x + y : float64)
    Times -> resume(x * y : float64)

fun term(x, y)
    c(1.0) +. (x *. x *. x) +. ((~.)(y *. y))
```

Evaluating expressions

```
val evaluate = handler
  ctl ap0(n) -> match(n) {Const (i) -> resume(i)}
  ctl ap1(u,x) -> match(u) {Negate -> resume(~x : float64)}
  ctl ap2(b,x,y) -> match(b)
    Plus -> resume(x + y : float64)
    Times -> resume(x * y : float64)

fun term(x, y)
    c(1.0) +. (x *. x *. x) +. ((~.)(y *. y))

fun main()
  with evaluate
  term(2.0, 4.0)
```

Output: -7

Summary so far

Summary so far

- Effects and handlers are a control flow construct like try-catch.

Summary so far

- Effects and handlers are a control flow construct like try-catch.
- Handlers in Koka are first class.

Summary so far

- Effects and handlers are a control flow construct like try-catch.
- Handlers in Koka are first class.
- Importantly, effects and handers allow resumption!

Summary so far

- Effects and handlers are a control flow construct like try-catch.
- Handlers in Koka are first class.
- Importantly, effects and handers allow resumption!
- We can then create an effect for smooth functions.

Summary so far

- Effects and handlers are a control flow construct like try-catch.
- Handlers in Koka are first class.
- Importantly, effects and handers allow resumption!
- We can then create an effect for smooth functions.
- We define an `evaluate` handler to interpret our effect in the case of floats.

What if we forget `evaluate`?

What if we forget evaluate?

```
evaluate-examples.kk(6, 5): error: there are unhandled effects for
 the main expression
  inferred effect : (smooth/smooth<float64>)
  unhandled effect: smooth/smooth<float64>
  hint            : wrap the main function in a handler
```

What if we forget `evaluate`?

```
evaluate-examples.kk(6, 5): error: there are unhandled effects for
 the main expression
  inferred effect : (smooth/smooth<float64>)
  unhandled effect: smooth/smooth<float64>
  hint            : wrap the main function in a handler
```

This is because `term` has the type

```
forall<a>. (a, a) -> (smooth<a>) a
```

meaning that when executed the operations of `smooth<a>` may occur.

Effect typing in Koka with row types

Effect typing in Koka with row types

```
effect<a> ctl default() : a
effect ctl exception(info : string) : a
```

Effect typing in Koka with row types

```
effect<a> ctl default() : a
effect ctl exception(info : string) : a

fun divide(x : int, y : int) : exception int
  if y == 0 then exception("Divided by zero") else x / y
```

Effect typing in Koka with row types

```
effect<a> ctl default() : a
effect ctl exception(info : string) : a

fun divide(x : int, y : int) : exception int
  if y == 0 then exception("Divided by zero") else x / y

fun combined() : <exception,default<int>> int
  divide(default(), default())
```

Effect typing in Koka with row types

```
effect<a> ctl default() : a
effect ctl exception(info : string) : a

fun divide(x : int, y : int) : exception int
  if y == 0 then exception("Divided by zero") else x / y

fun combined() : <exception,default<int>> int
  divide(default(), default())

fun main() : console () {
  with ctl exception(info) -> print("Exception: " ++ info)
  with ctl default() -> resume(0)
  print((combined : () -> <default<int>,exception,console> int)())
}
```

Handlers for different effects are order independent

Handlers for different effects are order independent

```
effect ctl eff0() : int
effect ctl eff1() : int
```

Handlers for different effects are order independent

```
effect ctl eff0() : int
effect ctl eff1() : int

fun main01()
  with ctl eff0() -> resume(1)
  with ctl eff1() -> resume(2)
  print(eff0() + eff1() : int)

fun main10()
  with ctl eff1() -> resume(2)
  with ctl eff0() -> resume(1)
  print(eff0() + eff1() : int)
```

Handlers have scope

Handlers have scope

```
effect<a> ctl default() : a

fun main()
  val a =
    with ctl default() -> resume(1)
    (default : () -> <console,default<int>> int)()
  val b =
    with ctl default() -> resume(2)
    default()
  print(show(a : int) ++ ", " ++ show(b : int))
```

Output: 1, 2

The inner-most handler has priority

The inner-most handler has priority

```
effect<a> ctl default() : a

fun main() : console () {
  with ctl default() ->
    resume(2)
  with ctl default() ->
    resume(1)
  val a = (default : () -> <console,default<int>,default<int>> int)()
  val b = default()
  print(show(a : int) ++ ", " ++ show(b : int))
}
```

Output: `1, 1`

The left `default<int>` corresponds to the inner handler.

Handlers can be bypassed dynamically

Handlers can be bypassed dynamically

```
effect<a> ctl default() : a

fun main()
  with ctl default() ->
    resume(2)
  with ctl default() ->
    resume(1)
  val a = (default : () -> <console,default<int>,default<int>> int)()
  val b = mask<default>{
    (default : () -> <console,default<int>> int)()
  }
  print(show(a : int) ++ ", " ++ show(b : int))
```

Output: 1, 2

Handlers must be bypassed for different instantiation

```
effect<a> ctl default() : a

fun main()
  with ctl default() ->
    resume(True)
  with ctl default() ->
    resume(1)
  val a = (default : () -> <console,default<int>,default<bool>> int)()
  val b = mask<default>{
    (default : () -> <console,default<bool>> bool)()
  }
  print(show(a : int) ++ ", " ++ show(b : bool))
```

Output: `1, True`

Handling an effect can use other effects

Handling an effect can use other effects

```
effect<a> ctl default() : a

fun main()
  with ctl default() -> resume(1)
  with ctl default() -> resume(default() + 1)
  print(default() : int)
```

Output: 2

Effect polymorphism

Effect polymorphism

```
fun twice(f : () -> e ()) : e ()
  f()
  f()
```

Effect polymorphism

```
fun twice(f : () -> e ()) : e ()
  f()
  f()

fun while_(pred : () -> <div|e> bool, body : () -> <div|e> ())
          : <div|e> ()
  if pred() then {body(); while_(pred, body)} else ()
```

Effect polymorphism

```
fun twice(f : () -> e ()) : e ()
  f()
  f()

fun while_(pred : () -> <div|e> bool, body : () -> <div|e> ())
          : <div|e> ()
  if pred() then {body(); while_(pred, body)} else ()

fun main() : <div,console> ()
  var i := 3
  while_ ({i > 0} : () -> <div,console,local<_h>> bool)
    i := i - 1
    print(show(i) ++ " ")
```

Output: 2 1 0

Helper functions continued

Helper functions continued

```
inline fun op0(n)
  match(n) {Const(x) -> c(x)}

inline fun op1(u, x)
  match(u) {Negate -> (~.)(x)}

inline fun op2(b, x, y)
  match(b)
    Plus -> x +. y
    Times -> x *. y
```

Helper functions continued

```
inline fun op0(n)
  match(n) {Const(x) -> c(x)}

inline fun op1(u, x)
  match(u) {Negate -> (~.)(x)}

inline fun op2(b, x, y)
  match(b)
    Plus -> x +. y
    Times -> x *. y
```

```
inline fun der1(u, x)
  match(u) {Negate -> (~.)(c(1.0))}

inline fun der2L(b, x, y)
  match(b)
    Plus -> c(1.0)
    Times -> y

inline fun der2R(b, x, y)
  match(b)
    Plus -> c(1.0)
    Times -> x
```

Forward mode AD

Forward mode AD

```
type dual<a>
  Dual(v : a, dv : a)

val forward = handler
  ctl ap0(n) ->
    resume(Dual(op0(n), c(0.0)))
  ctl ap1(u,x) ->
    resume(Dual(op1(u,x.v), der1(u,x.v) *. x.dv))
  ctl ap2(b,x,y) ->
    resume(Dual(op2(b,x.v,y.v), (der2L(b,x.v,y.v) *. x.dv) +.
                                (der2R(b,x.v,y.v) *. y.dv)))
```

Forward mode AD

```
type dual<a>
  Dual(v : a, dv : a)

val forward = handler
  ctl ap0(n) ->
    resume(Dual(op0(n), c(0.0)))
  ctl ap1(u,x) ->
    resume(Dual(op1(u,x.v), der1(u,x.v) *. x.dv))
  ctl ap2(b,x,y) ->
    resume(Dual(op2(b,x.v,y.v), (der2L(b,x.v,y.v) *. x.dv) +.
                                (der2R(b,x.v,y.v) *. y.dv)))

fun d(f : dual<a> -> <smooth<dual<a>>,smooth<a>|e> dual<a>, x : a)
      : <smooth<a>|e> a
  val res = forward{f(Dual(x,mask<smooth>{c(1.0)}))}
  res.dv
```

Using forward mode

Using forward mode

```
fun main()
  with evaluate
  d(fn(x) {term(x, c(4.0))}, c(2.0))
```

Using forward mode

```
fun main()
  with evaluate
  d(fn(x) {term(x, c(4.0))}, c(2.0))
```

We are calculating

$$\frac{d}{dx}\left(1 + x^3 + (-y^2)\right) = 3x^2$$

at $x = 2$ and $y = 4$.

Output: 12

Nesting forward mode

Nesting forward mode

We can also nest forward mode to calculate second derivatives.

Nesting forward mode

We can also nest forward mode to calculate second derivatives.

```
fun main()
  with evaluate
  d(fn(y) {d(fn(x) {x *. x *. x}, y)}, c(1.0))
```

Output: 6

Nesting forward mode

We can also nest forward mode to calculate second derivatives.

```
fun main()
  with evaluate
  d(fn(y) {d(fn(x) {x *. x *. x}, y)}, c(1.0))
```

Output: 6

What about perturbation confusion, e.g. correctly calculating

$$\frac{d}{dx}\left(x \cdot \left(\frac{d}{dy}x + y\Big|_{y=1}\right)\right)\Bigg|_{x=1} = 1$$

from J. M. Siskind and B. A. Pearlmutter, 'Perturbation Confusion and Referential Transparency'

Perturbation confusion

Perturbation confusion

$$\frac{d}{dx}\left(x \cdot \left(\frac{d}{dy}x + y\Big|_{y=1}\right)\right)_{x=1}$$

```
fun confusion()
  with evaluate
  d(fn(x) {x *. d(fn(y) {x +. y}, c(1.0))}, c(1.0))
```

Perturbation confusion

```
fun confusion()
  with evaluate
  d(fn(x) {x *. d(fn(y) {x +. y}, c(1.0))}, c(1.0))
```

```
forward-confusion-one.kk(9,17): error: types do not match (due to an
 infinite type)
  context       :                x *. d(fn(y) {x +. y}, c(1.0))
  term          :                     d(fn(y) {x +. y}, c(1.0))
  inferred type: _a
  expected type: dual<_a>
  hint          : give a type to the function definition?
```

x : dual<a> and y : dual<dual<a>>

Perturbation confusion, with lifting

Perturbation confusion, with lifting

```
fun lift(x : a) : <smooth<dual<a>>,smooth<a>> dual<a>
  mask<smooth>{Dual(x, c(0.0))}
```

Perturbation confusion, with lifting

```
fun lift(x : a) : <smooth<dual<a>>,smooth<a>> dual<a>
  mask<smooth>{Dual(x, c(0.0))}
```

Thus `lift(x) : dual<dual<a>>`, and so

```
fun confusion-lift()
  with evaluate
  d(fn(x) {x *. d(fn(y) {lift(x) +. y}, c(1.0))}, c(1.0))
```

produces the correct output of 1.

Perturbation confusion, with lifting the wrong way

Perturbation confusion, with lifting the wrong way

```
fun lift(x : a) : <smooth<dual<a>>,smooth<a>> dual<a>
  mask<smooth>{Dual(x, c(0.0))}
fun confusion-lift()
  with evaluate
  d(fn(x) {x *. lift(d(fn(y) {x +. y}, c(1.0)))}, c(1.0))
```

Perturbation confusion, with lifting the wrong way

```
fun lift(x : a) : <smooth<dual<a>>,smooth<a>> dual<a>
  mask<smooth>{Dual(x, c(0.0))}

fun confusion-lift()
  with evaluate
  d(fn(x) {x *. lift(d(fn(y) {x +. y}, c(1.0)))}, c(1.0))
```

```
forward-confusion-lift-wrong.kk(9,17): error: types do not match (due
 to an infinite type)
  context        :                 lift(d(fn(y) {x +. y}, c(1.0)))
  term           :                 lift
  inferred effect: <smooth<_a>|_e1>
  expected effect: <smooth<dual<_a>>,smooth<_a>|_e>
  because        : effect cannot be subsumed
  hint           : give a type to the function definition?
```

Perturbation confusion, with lifting the wrong way explained

```
fun d(f : dual<a> -> <smooth<dual<a>>,smooth<a>|e> dual<a>, x : a)
     : <smooth<a>|e> a
fun lift(x : a) : <smooth<dual<a>>,smooth<a>> dual<a>
fun confusion-lift()
  with evaluate
  d(fn(x) {x *. lift(d(fn(y) {x +. y}, c(1.0)))}, c(1.0))
```

Working inside out, both `x` and `y` have type `dual<a>`. Thus, the inner `d` needs effect context `<smooth<a>|e>`. Using `lift` requires unification with `<smooth<dual<a>>,smooth<a>|e'>`. The leftmost occurrences of `smooth` must unify, causing an infinite type.

Perturbation confusion, with lifting the wrong way continued

Therefore, if we hide the leftmost `smooth`, type checking succeeds.

```
fun lift(x : a) : <smooth<dual<a>>,smooth<a>> dual<a>
  mask<smooth>{Dual(x, c(0.0))}
```

```
fun confusion-lift-mask()
  with evaluate
  d(fn(x) {x *. lift(mask<smooth>{d(fn(y) {x +. y}, c(1.0))})}, c(1.0))
```

Output: 2

The answer of 2 is the result of confusing the dual components of each derivative.

Perturbation confusion, with lifting the wrong way again

Perturbation confusion, with lifting the wrong way again

```
fun lift(x : a) : <smooth<dual<a>>,smooth<a>> dual<a>
  mask<smooth>{Dual(x, c(0.0))}
fun confusion-lift()
  with evaluate
  d(fn(x) {x *. d(fn(y) {lift(x +. x) +. y}, c(1.0))}, c(1.0))
```

```
forward-confusion-lift-other.kk(9,26): error: types do not match (due
 to an infinite type)
  context         :                                  lift(x +. x)
  term            :                                  lift
  inferred effect: <smooth<dual<_a>>|_e1>
  expected effect: <smooth<dual<dual<_a>>>,smooth<dual<_a>>|_e>
  because         : effect cannot be subsumed
  hint            : give a type to the function definition?
```

Summary so far

Summary so far

- Koka has an effect type system using row types.

Summary so far

- Koka has an effect type system using row types.
- Handlers create a scope, with the innermost handler having priority.

Summary so far

- Koka has an effect type system using row types.
- Handlers create a scope, with the innermost handler having priority.
- We can choose to skip the innermost hander with a masking operation.

Summary so far

- Koka has an effect type system using row types.
- Handlers create a scope, with the innermost handler having priority.
- We can choose to skip the innermost hander with a masking operation.
- Koka has effect polymorphism.

Summary so far

- Koka has an effect type system using row types.
- Handlers create a scope, with the innermost handler having priority.
- We can choose to skip the innermost hander with a masking operation.
- Koka has effect polymorphism.
- Forward mode and a derivative operator are succintly expressed using handlers.

Summary so far

- Koka has an effect type system using row types.
- Handlers create a scope, with the innermost handler having priority.
- We can choose to skip the innermost hander with a masking operation.
- Koka has effect polymorphism.
- Forward mode and a derivative operator are succintly expressed using handlers.
- The derivative operator can be iterated.

Summary so far

- Koka has an effect type system using row types.
- Handlers create a scope, with the innermost handler having priority.
- We can choose to skip the innermost hander with a masking operation.
- Koka has effect polymorphism.
- Forward mode and a derivative operator are succintly expressed using handlers.
- The derivative operator can be iterated.
- The value and effect type systems help avoid perturbation confusion.

Benchmarks!

Benchmarks!

$$\frac{1}{x} = \sum_{n=0}^{\infty} \left(-(x-1)\right)^n$$

Benchmarks!

$$\frac{1}{x} = \sum_{n=0}^{\infty} \left(-(x-1)\right)^n$$

```
fun approx-recip(iters : int, x : a) : <smooth<a>,div> a {
  var acc := c(1.0)
  var prev := c(1.0)
  repeat(iters) {
    prev := (prev *. (~.)(x -. c(1.0)))
    acc := (acc +. prev)
  }
  acc
}
```

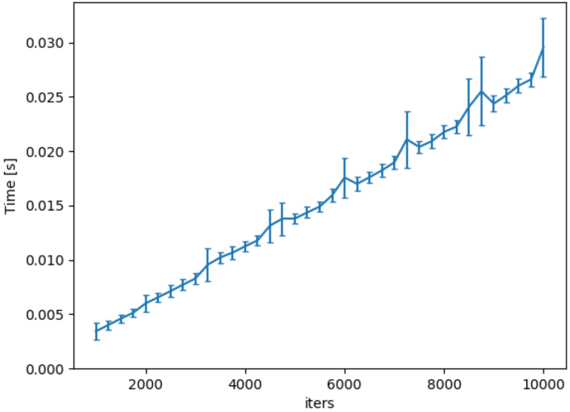
Benchmark: `evaluate`

Benchmark: evaluate

```
fun main()
  val sz =
    match get-args()
      Nil -> 500
      Cons(arg0, _) ->
        match parse-int(arg0)
          Just(sz) -> sz
          _        -> 500
  with evaluate
  val res =
    approx-recip(sz, c(0.5))
  println(res : float64)
```

Benchmark: evaluate

```
fun main()
  val sz =
    match get-args()
      Nil  -> 500
      Cons(arg0, _) ->
        match parse-int(arg0)
          Just(sz) -> sz
          _        -> 500
  with evaluate
  val res =
    approx-recip(sz, c(0.5))
  println(res : float64)
```



Benchmark: `forward`

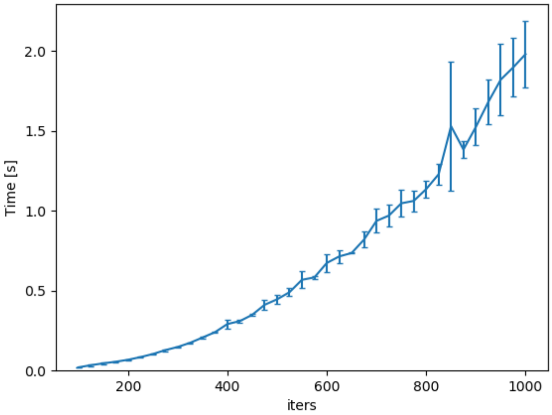
Benchmark: forward

```
fun main()
  val sz =
    match get-args()
      Nil -> 500
      Cons(arg0, _) ->
        match parse-int(arg0)
          Just(sz) -> sz
          _        -> 500
  with evaluate
  val res = d(fn(x)
    {approx-recip(sz, x)},
    c(0.5))
  println(res : float64)
```

Benchmark: forward

```
fun main()
  val sz =
    match get-args()
      Nil -> 500
      Cons(arg0, _) ->
        match parse-int(arg0)
          Just(sz) -> sz
          _        -> 500
  with evaluate
  val res = d(fn(x)
    {approx-recip(sz, x)},
    c(0.5))
  println(res : float64)
```

Oh no, quadratic! (Koka performance bug)



Tail resumptive effects

Tail resumptive effects

Both `evaluate` and `forward` are *tail-resumptive*, i.e. the last thing they do is call `resume`. Koka can take advantage of this special case by using `fun` handler cases:

> *Unlike a general `ctl` operation, there is no need to yield upward to the handler, capture the stack, and eventually resume again. This gives `fun` (and `val`) operations a performance cost very similar to virtual method calls which can be quite efficient. (Koka documentation)*

Tail resumptive effects

Both `evaluate` and `forward` are *tail-resumptive*, i.e. the last thing they do is call `resume`. Koka can take advantage of this special case by using `fun` handler cases:

> *Unlike a general `ctl` operation, there is no need to yield upward to the handler, capture the stack, and eventually resume again. This gives `fun` (and `val`) operations a performance cost very similar to virtual method calls which can be quite efficient. (Koka documentation)*

```
val evaluate = handler
  ctl ap0(n) -> match(n) {Const(i) -> resume(i)}
  ctl ap1(u,x) -> match(u) {Negate -> resume(~x : float64)}
  ctl ap2(b,x,y) -> match(b)
    Plus -> resume(x + y : float64)
    Times -> resume(x * y : float64)
```

Tail resumptive effects

Both `evaluate` and `forward` are *tail-resumptive*, i.e. the last thing they do is call `resume`. Koka can take advantage of this special case by using `fun` handler cases:

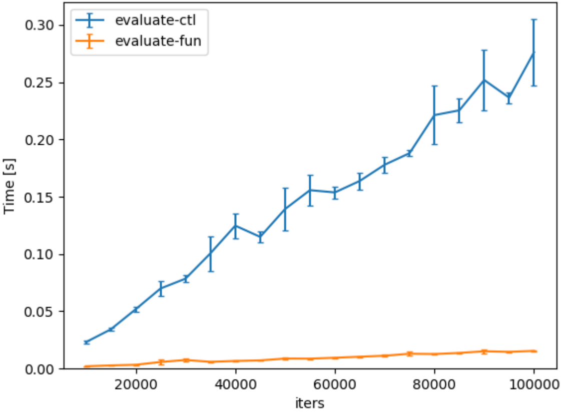
> *Unlike a general `ctl` operation, there is no need to yield upward to the handler, capture the stack, and eventually resume again. This gives `fun` (and `val`) operations a performance cost very similar to virtual method calls which can be quite efficient. (Koka documentation)*

```
val evaluate = handler
  fun ap0(n) -> match(n) {Const(i) -> i}
  fun ap1(u,x) -> match(u) {Negate -> (~x : float64)}
  fun ap2(b,x,y) -> match(b)
    Plus -> (x + y : float64)
    Times -> (x * y : float64)
```

Performance gains for `evaluate`

Performance gains for `evaluate`

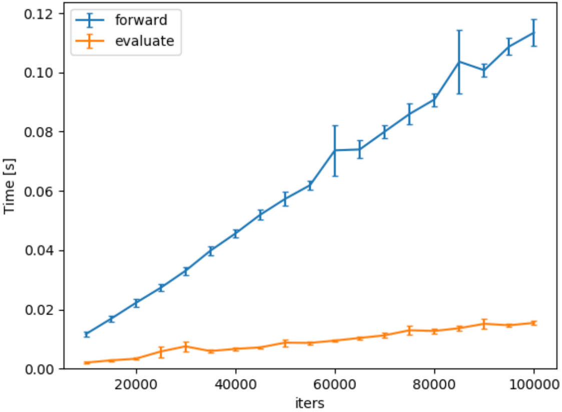
We get almost a x20 speed-up!



Performance of `forward`

Performance of `forward`

Forward mode AD is a constant multiple slower than no AD as both lines are linear. Unfortunately it is about x10 slower than no AD, whereas the theoretical optimum is x2-3, but not quadratic!



Why even have control operators then?

Why even have control operators then?

```
effect<a> ctl choice(l : a, r : a) : a
```

Why even have control operators then?

```
effect<a> ctl choice(l : a, r : a) : a

val all-choices = handler
  return(x) -> [x]
  ctl choice(l, r) -> resume(l) ++ resume(r)
```

Why even have control operators then?

```
effect<a> ctl choice(l : a, r : a) : a

val all-choices = handler
  return(x) -> [x]
  ctl choice(l, r) -> resume(l) ++ resume(r)

fun main()
  val res =
    with all-choices
    [choice("a","x"),choice("b","y")]
  res.map(join)
```

Output: `["ab","ay","xb","xy"]`

Reverse mode AD

Reverse mode AD

```
type prop<h,a>
    Prop(v : a, dv : ref<h, a>)
```

Reverse mode AD

```
type prop<h,a>
    Prop(v : a, dv : ref<h, a>)

val reverse = handler
  ctl ap0(n) ->
    val r = Prop(op0(n), ref(c(0.0)))
    resume(r)
  ctl ap1(u,x) ->
    val r = Prop(op1(u,x.v), ref(c(0.0)))
    resume(r)
    set(x.dv, !x.dv +. (der1(u,x.v) *. !r.dv))
  ctl ap2(b,x,y) ->
    val r = Prop(op2(b,x.v,y.v), ref(c(0.0)))
    resume(r)
    set(x.dv, !x.dv +. (der2L(b,x.v,y.v) *. !r.dv))
    set(y.dv, !y.dv +. (der2R(b,x.v,y.v) *. !r.dv))
```

Essentially the same as “Demystifying”

Reverse mode AD continued

Reverse mode AD continued

```
fun grad(f, x)
  val z = Prop(x, ref(c(0.0)))
  reverse{set(f(z).dv, mask<smooth>{c(1.0)})}
  !z.dv
```

Reverse mode AD continued

```
fun grad(f, x)
  val z = Prop(x, ref(c(0.0)))
  reverse{set(f(z).dv, mask<smooth>{c(1.0)})}
  !z.dv

fun main()
  with evaluate
  grad(fn(x) {term(x, c(4.0))}, c(2.0))
```

Output: 12

Reverse mode AD continued

```
fun grad(f, x)
  val z = Prop(x, ref(c(0.0)))
  reverse{set(f(z).dv, mask<smooth>{c(1.0)})}
  !z.dv

fun main()
  with evaluate
  grad(fn(x) {term(x, c(4.0))}, c(2.0))
```

Output: 12

Can also be nested like forward mode, and multiple modes can be mixed.

Benchmark: reverse

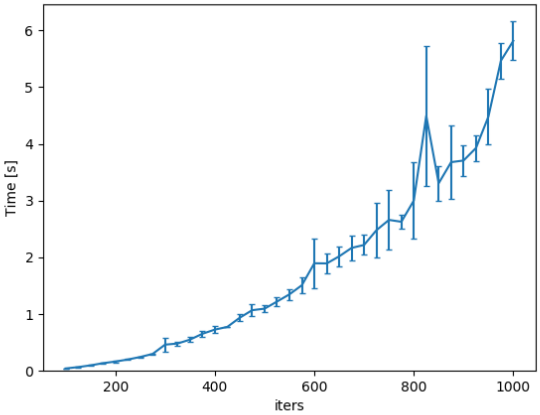
Benchmark: reverse

```
fun main()
  val sz =
    match get-args()
      Nil -> 500
      Cons(arg0, _) ->
        match parse-int(arg0)
          Just(sz) -> sz
          _        -> 500
  with evaluate
  val res = grad(
    fn(x) {approx-recip(sz, x)},
    c(0.5))
  println(res : float64)
```

Benchmark: reverse

```
fun main()
  val sz =
    match get-args()
      Nil  -> 500
      Cons(arg0, _) ->
        match parse-int(arg0)
          Just(sz) -> sz
          _        -> 500
  with evaluate
  val res = grad(
    fn(x) {approx-recip(sz, x)}
    c(0.5))
  println(res : float64)
```

Also quadratic, and very slow



A workaround using control effects

A workaround using control effects

```
val reverse = handler
  ctl ap0(n) ->
    val r = evaluate({Prop (op0(n), ref(c(0.0)))})
    resume(r)
  ctl ap1(u,x) ->
    val r = evaluate({Prop (op1(u,x.v), ref(c(0.0)))})
    resume(r)
    with evaluate
    set(x.dv, !x.dv +. (der1(u,x.v) *. !r.dv))
  ctl ap2(b,x,y) ->
    val r = evaluate({Prop (op2(b,x.v,y.v), ref(c(0.0)))})
    resume(r)
    with evaluate
    set(x.dv, !x.dv +. (der2(b,L,x.v,y.v) *. !r.dv))
    set(y.dv, !y.dv +. (der2(b,R,x.v,y.v) *. !r.dv))
```

Benchmarking the workaround

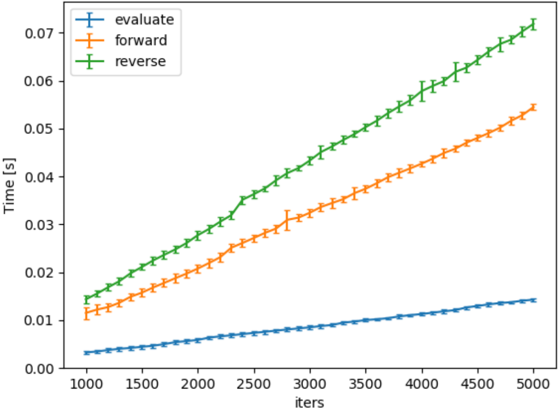
Benchmarking the workaround

```
reverse : forall<h,e>. (() -> <st<h>,smooth<prop<h,float64>>|e> ())
        -> <st<h>|e> ()
```

Benchmarking the workaround

```
reverse : forall<h,e>. (() -> <st<h>,smooth<prop<h,float64>>|e> ())
        -> <st<h>|e> ()
```

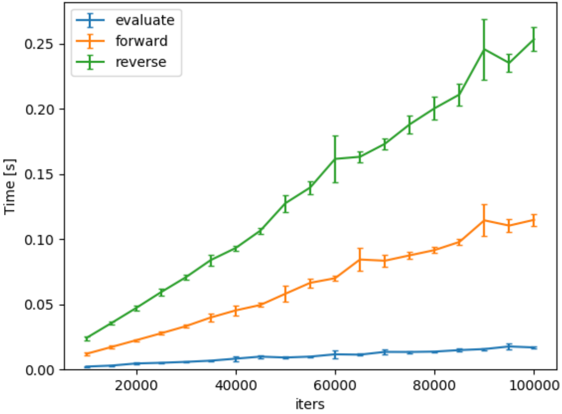
Everything is linear! However, we lose compositionality. Also, things are still slow.



Better solution, use taped reverse AD

Show code

Comparison of taped reverse AD



Checkpointed reverse mode

Checkpointed reverse mode

```
rec effect checkpoint<h,a,e>
  ctl check(
    prog : () -> <checkpoint<h,a,e>,smooth<prop<h,a>>,div|e> prop<h,a>
  ) : prop<h,a>
```

Checkpointed reverse mode

```
rec effect checkpoint<h,a,e>
  ctl check(
    prog : () -> <checkpoint<h,a,e>,smooth<prop<h,a>>,div|e> prop<h,a>
  ) : prop<h,a>

fun lift(action : () -> <smooth<prop<h,a>>|e> b)
        : <smooth<prop<h,a>>, smooth<a>|e> b {
```

Checkpointed reverse mode continued

Checkpointed reverse mode continued

```
fun evaluatet(
      s : ref<h,a>
  ,   action : (() -> <checkpoint<h,a,e>,div,
                       smooth<prop<h,a>>,smooth<a>|e> b)
  ) : <div,smooth<a>|e> b
  with handler
    ctl check(p) ->
      val r = evaluatet(s, {lift{p()}})
      resume(r)
  with handler
    ctl ap0(n) -> resume(Prop(op0(n), s))
    ctl ap1(u,x) -> resume(Prop(op1(u,x.v), s))
    ctl ap2(b,x,y) -> resume(Prop(op2(b,x.v,y.v), s))
  action()
```

Checkpointed reverse mode continued

Checkpointed reverse mode continued

```
fun reversec(
  action : (() -> <checkpoint<h,a,<st<h>|e>>,div,
                 smooth<prop<h,a>>,smooth<a>,st<h>|e> ())
  ) : <div,st<h>,smooth<a>|e> ()
  with handler
    ctl check(p) ->
      val s = ref(c(0.0))
      val res = evaluatet(s, {lift{p()}})
      val r = Prop(res.v, ref(c(0.0)))
      resume(r)
      reversec{set((lift{p()}).dv, !r.dv)}
  with reverse
  action()
```

Conclusion

Conclusion

- We've seen how to implement various AD modes using algebraic effects.

Conclusion

- We've seen how to implement various AD modes using algebraic effects.
- The effect type system helps us avoid perturbation confusion.

Conclusion

- We've seen how to implement various AD modes using algebraic effects.
- The effect type system helps us avoid perturbation confusion.
- Algebraic effects impose some overhead, but still provide the correct asymptotics.

Conclusion

- We've seen how to implement various AD modes using algebraic effects.
- The effect type system helps us avoid perturbation confusion.
- Algebraic effects impose some overhead, but still provide the correct asymptotics.
- Algebraic effects are making their way into other languages, OCaml 5.0 has them, and so will WASM.

Hessian-vector product via forward-over-reverse

```
fun grad-list(f, xs)
  val z : list<_> = map(xs, fn(x) {Prop(x, ref(c(0.0)))})
  reverse{set(f(z).reverse/dv, mask<smooth>{c(1.0)})}
  map(z, fn(x) {!x.reverse/dv})

fun hessian-vector(f, w, v)
  val backward = fn(r)
    grad-list(f, zipwith(w, v, fn(wi, vi) {lift(wi) +. (r *. lift(vi))}))
  val prod = forward{
    backward(Dual(mask<smooth>{c(0.0)},mask<smooth>{c(1.0)}))
  }
  prod.map(dv)
```