

*One-shot*  
Algebraic Effects  
as  
Coroutines

@TFP 2020

Satoru Kawahara and Yuki Yoshi Kameyama

University of Tsukuba, Japan

# Algebraic Effects & Handlers

- ▶ Modular way to write computational effects
- ▶ Intuition: **Resumable exception**

# Algebraic Effects & Handlers

**Exceptions** *CANNOT* resume the rest of computation.

```
exception Error : int → exn

let handle_error th =
  try th () with
  | Error i → i

handle_error (fun () →
  (raise (Error 2)) + 3)
```


# Algebraic Effects & Handlers

**Exceptions** *CANNOT* resume the rest of computation.

```
exception Error : int → exn

let handle_error th =
  try th () with
  | Error i → i

handle_error (fun () →
  (raise (Error 2)) + 3)
```



# Algebraic Effects & Handlers

**Exceptions** *CANNOT* resume the rest of computation.

```
exception Error : int → exn
```

```
let handle_error th =
```

```
  try th () with
```

```
  | Error i → i
```

```
handle_error (fun () →
```

```
  (raise (Error 2)) + 3)
```



rest of computation

# Algebraic Effects & Handlers


**Algebraic Effect** *CAN* resume the rest of computation.

```
effect Defer : (() → ()) → ()  
  
let handle_defer th =  
  handle th () with  
  | Defer f k → k (); f ()  
  
handle_defer (fun () →  
  (#Defer (fun () → puts "world")));  
puts "hello")
```

# Algebraic Effects & Handlers

**Algebraic Effect** CAN resume the rest of computation.


```
effect Defer : (() → ()) → ()  
  
let handle_defer th =  
  handle th () with  
  | Defer f k → k (); f ()  
  
handle_defer (fun () →  
  (#Defer (fun () → puts "world"));  
  puts "hello")
```



# Algebraic Effects & Handlers

**Algebraic Effect** CAN resume the rest of computation.

```
effect Defer : (() → ()) → ()  
  
let handle_defer th =  
  handle th () with  
  | Defer f k → k (); f ()  
  
handle_defer (fun () →  
  (#Defer (fun () → puts "world"));  
  puts "hello")
```





# Algebraic Effects & Handlers

**Algebraic Effect** CAN resume the rest of computation.

```
effect Defer : (() → ()) → ()  
  
let handle_defer th puts "hello"  
  handle th () with  
  | Defer f k → k (); f ()  
  
handle_defer (fun () →  
  (#Defer (fun () → puts "world"));  
  puts "hello")
```

# Algebraic Effects & Handlers

- ▶ Modular way to define computational effects
- ▶ Intuition: Resumable exception
- 🔥 Hot topic for Researchers
  - theoretical development
  - implementing various control abstractions: delimited controls, monads, concurrency, probabilistic/reactive programming, etc.

# Algebraic Effects & Handlers

- ▶ Modular way to define computational effects
- ▶ Intuition: Resumable exception



Hot topic for Researchers

- theoretical development
- implementing various control abstractions: delimited controls, monads, concurrency, probabilistic/reactive programming, etc.



But is it hot topic *for Programmers?*

**Yes!**

## Yes!

*However*, we must solve these problems:

- ▶ Only **research** languages have AE as a built-in
- ▶ **Few** languages can implement AE as a library

# Algebraic Effects for Programmers

Is there any way to implement AE ?

- ▶ available in **popular** languages
- ▶ **without** significant overhead



## *One-shot Algebraic Effects* as Coroutines

## *One-shot Algebraic Effects* as Coroutines

- ✓ widely available in those which have coroutines
- ✓ shallow embedding: **without** large overhead



## *One-shot* Algebraic Effects as Coroutines

- ✓ widely available in those which have coroutines
- ✓ shallow embedding: **without** large overhead
- 👉 only with *one-shot* restriction

# Technically

To implement **AE** by **coroutines**.....

# Technically

To implement **AE** by **coroutines**.....

1. define **two languages**

- $\lambda_{eff}$  .....  $\lambda$  with *One-shot Algebraic Effects*
- $\lambda_{ac}$  .....  $\lambda$  with *Asymmetric Coroutines*

# Technically

To implement **AE** by **coroutines**.....

1. define **two languages**

- $\lambda_{eff}$  .....  $\lambda$  with *One-shot Algebraic Effects*
- $\lambda_{ac}$  .....  $\lambda$  with *Asymmetric Coroutines*

2. define a **translation** from  $\lambda_{eff}$  to  $\lambda_{ac}$

# Technically

To implement **AE** by **coroutines**.....

1. define **two languages**

- $\lambda_{eff}$  .....  $\lambda$  with *One-shot Algebraic Effects*
- $\lambda_{ac}$  .....  $\lambda$  with *Asymmetric Coroutines*

2. define a **translation** from  $\lambda_{eff}$  to  $\lambda_{ac}$

based on **macro expressibility** [Felleisen 1991]  
**local** and **compositional**

# Technically

To implement **AE** by **coroutines**.....

1. define two languages

- $\lambda_{eff}$  .....  $\lambda$  with *One-shot Algebraic Effects*
- $\lambda_{ac}$  .....  $\lambda$  with *Asymmetric Coroutines*

2. define a **translation** from  $\lambda_{eff}$  to  $\lambda_{ac}$

based on **macro expressibility** [Felleisen 1991]  
**local** and **compositional**

# Connection between two controls

## Algebraic Effects

.....

handle

... (#Op x) .....

.....

.....

## Coroutines

resume co; resume co

co

... (yield x) .....

.....

.....

# Connection between two controls

## Algebraic Effects

.....

handle

... (#Op x) .....

.....

.....

## Coroutines

resume co; resume co

co

... (yield x) .....

.....

.....



# Connection between two controls

## Algebraic Effects

```
..... k o; k u  
handle  
  .. (#Op x) .....  
  .....  
.....
```

## Coroutines

```
resume co; resume co  
co  
  .. (yield x) .....  
  .....  
.....
```

# Connection between two controls

## Algebraic Effects

```
..... k o; k u  
handle  
  .. (#Op x) .....  
  .....  
.....
```

The diagram illustrates the control flow for Algebraic Effects. It shows a sequence of operations: `.....`, `k o; k u`, `handle`, `.. (#Op x) .....`, `.....`, and `.....`. A red arrow points from the `handle` keyword to the `(#Op x)` expression. Two blue arrows point from the `o` and `u` in the `k o; k u` line to the `.....` block following `(#Op x)`. A green box encloses the `handle` block and the `.....` block following `(#Op x)`. A red box highlights the `.....` block following `(#Op x)`.

## Coroutines

```
resume co; resume co  
co  
  .. (yield x) .....  
  .....  
.....
```

The diagram illustrates the control flow for Coroutines. It shows a sequence of operations: `resume co; resume co`, `co`, `.. (yield x) .....`, `.....`, and `.....`. A green box encloses the `co` block and the `.....` block following `(yield x)`.

# Connection between two controls

## Algebraic Effects

```
..... k o; k u  
handle  
  .. (#Op x) .....  
  .....  
.....
```

The diagram illustrates the control flow for Algebraic Effects. It shows a sequence of code blocks. The first block contains the code `..... k o; k u`. Below it is a `handle` block, which is enclosed in a green box. Inside the `handle` block, there is a red box containing the code `.. (#Op x) ..` followed by a red box containing `.....`. A red arrow points from the `handle` block back to the `..... k o; k u` block. Two blue arrows point from the `o` and `u` in the first block to the `(#Op x)` and the red box in the `handle` block respectively. Below the `handle` block is another red box containing `.....`. At the bottom of the diagram is another `.....`.

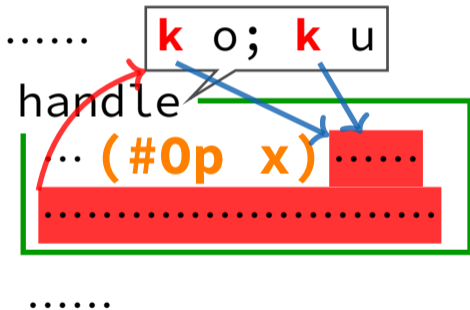
## Coroutines

```
resume co; resume co  
co  
  .. (yield x) .....  
  .....  
.....
```

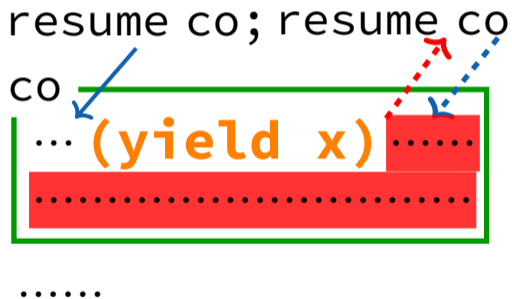
The diagram illustrates the control flow for Coroutines. It shows a sequence of code blocks. The first block contains the code `resume co; resume co`. Below it is a `co` block, which is enclosed in a green box. Inside the `co` block, there is a red box containing the code `.. (yield x) ..` followed by a red box containing `.....`. A blue arrow points from the `co` block back to the `resume co; resume co` block. Below the `co` block is another red box containing `.....`. At the bottom of the diagram is another `.....`.

# Connection between two controls

## Algebraic Effects



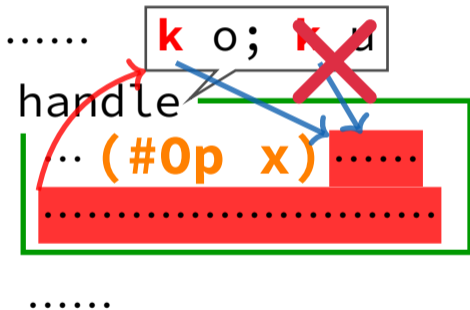
## Coroutines



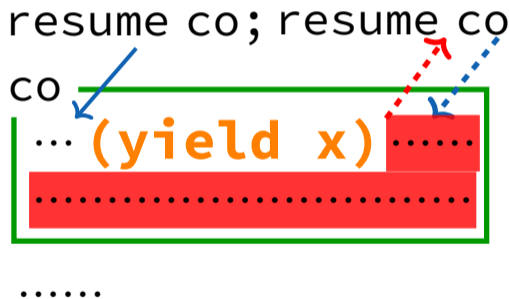
# Connection between two controls

## One-Shot

Algebraic Effects



Coroutines



# Translation

**One-shot**  
Algebraic Effects  $\rightarrow$  Asymmetric  
Coroutines

# Translation

**One-shot**  
Algebraic Effects  $\rightarrow$  Asymmetric  
Coroutines

$\Downarrow$   
 $\llbracket - \rrbracket : \lambda_{eff} \rightarrow \lambda_{ac}$

- simple ..... easy to implement
- local ..... as library

# Going to real world

Our translation as libraries



published on  GitHub



# Going to real world

Our translation as libraries

 Lua

 Ruby

 JavaScript

 Rust

published on  GitHub

implemented by *other users*  
based on ours

# Going to real world

Our translation as libraries

 Lua

 Ruby

 JavaScript

 Rust

published on  GitHub

implemented by *other users*  
based on ours

○ the fruits of versatility and simplicity

# Going to real world



Lua: *200 lines*

```
local Choice = inst()
local always_right = handler {
  [Choice] = function(k, l, r)
    return k(r)
  end
}

always_right(function()
  perform(Choice(1, 2)) + 3
end)
```



Ruby: *340 lines*

```
Choice = Effect.new
always_right = Handler.new
  .on(Choice){|k, l, r| k[r] }

always_right.run {
  Choice.perform(1, 2) + 3
}
```

# Evaluation

We compare the **performance** in Lua:

- ▲ our library based on the translation
- implementation based on **Free Monad**

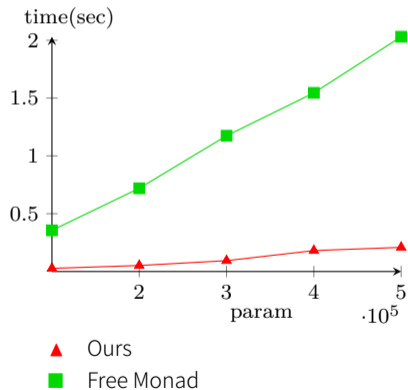
# Evaluation

We compare the **performance** in Lua:

- ▲ our library based on the translation
- implementation based on **Free Monad**

|             | Free<br>Monad | Ours | smaller is faster |
|-------------|---------------|------|-------------------|
| onestate    | 1             | 0.10 |                   |
| multistate  | 1             | 3.16 | ←                 |
| looper      | 1             | 0.11 |                   |
| same-fringe | 1             | 0.04 |                   |

## Single effect



- invoking effect inside a rec fun
- catching by handler outside the fun

○ 10x faster



Ours:

continuations as coroutines

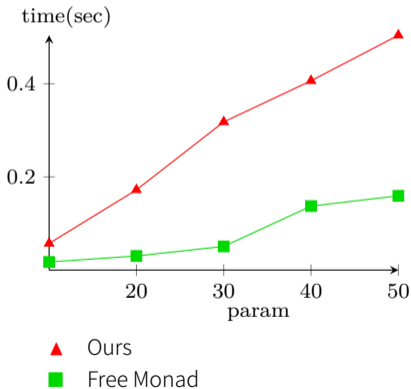


Free Monad:

continuations as function closures

## Nested handler

- onestate with fixed parameter
- catching the **outermost** handler



✗ 0.3x but.....

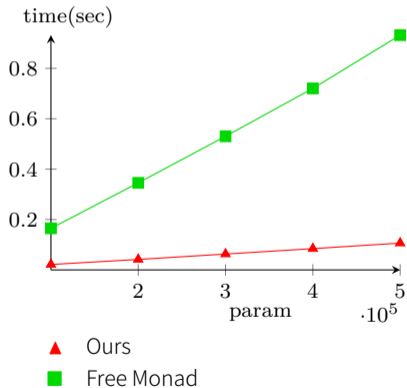
▶ Free Monad also slows down

0.6~0.07x of onestate

▶ Nesting 50 is extreme and will not be a real problem

# looper

## Iteration



- invoking effect inside of loop
- catching them outside of loop

○ 9x faster

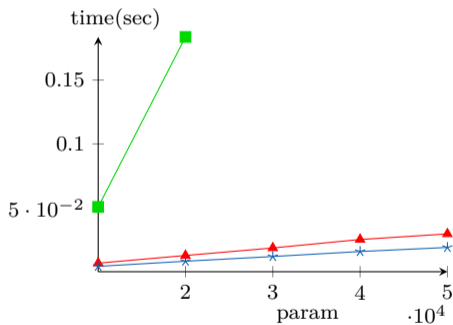
▶ Ours: built-in `for` loop

▶ Free Monad:  
`forM` with function closures



# same fringe

## Re-implementing native effects



- ★ Lua native coroutine
- ▲ Ours
- Free Monad

- re-implementing coroutine
- solving same-fringe problem

- 18x than Free Monad
  - 0.65x than Native Coroutines
- It can be run without large overhead

# Summary

A new **Algebraic Effects** implementation using **Coroutines**:

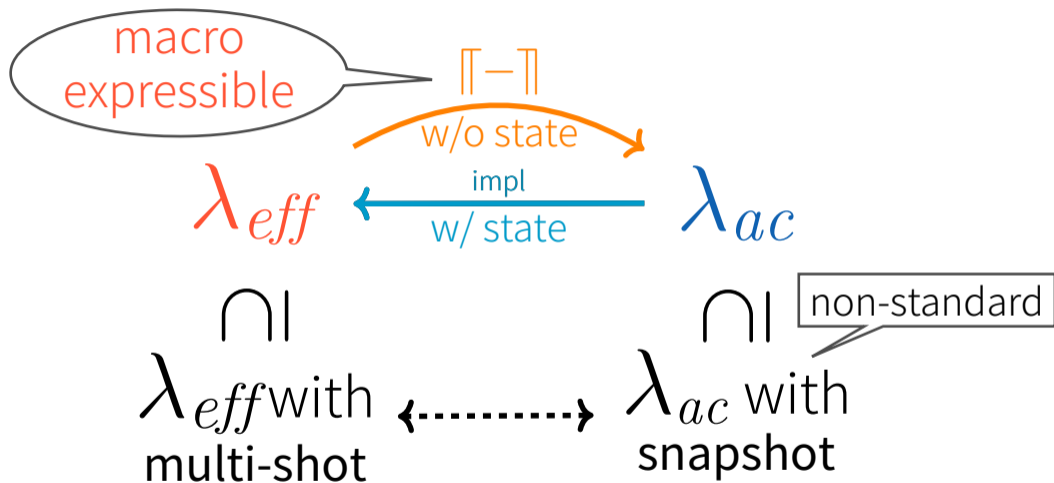
- ▶ relate **one-shot** AE with Coroutines
- ▶ define **macro-expressible** translation
- ▶ implement the translation as a **library**
- ▶ resolve existing problems
  - ✓ widely available in those which have coroutines
  - ✓ shallow embedding: without large overhead
  - ✓ direct-style without special syntax or transpilation

# One-shot continuation/AE

|                         | expressive power | copying stack      |
|-------------------------|------------------|--------------------|
| multi-shot continuation | strong           | necessary          |
| one-shot continuation   | weak             | <b>unnecessary</b> |
| coroutines              | weak             | <b>unnecessary</b> |

- ▶ No copying makes it run efficiently
- Exception, State, Concurrent, DI
- × Nondet, Backtracking

# Translation



## ⚠ Conflicts with existing effects

Our translation  $\llbracket - \rrbracket$  is defined  
*disregard* for the effects of host language  
coroutines, exceptions, etc.

💡 **re-implement** them with AE  
to resolve the conflict  
enables to mix in their effects