How do you *implement* Algebraic Effects?

びしょ〜じょ

effect system 勉強会

May 26, 2019

やること

Algebraic Effects and Handlers

の

さまざまなインプリ方法

について考える。

Table of Contents

Who talks

こんにちは、びしょ〜じょです。

- **筑波大学大学院 M2**
  プログラム言語や型とか検証などの研究室で
  プログラム変換の研究

Nymphium

Introduction

Algebraic Effects の様々な実装

Introduction

Algebraic Effects の様々な実装

- ライブラリ
  - eff.lua
  - Effekt
  - libhandler
  - etc.

Introduction

Algebraic Effects の様々な実装

- ライブラリ
  - eff.lua
  - Effekt
  - libhandler
  - etc.
- 言語 (処理系)
  - Eff
  - Multicore OCaml
  - Koka
  - etc.

Introduction

Algebraic Effects の様々な実装

- ライブラリ
  - eff.lua
  - Effekt
  - libhandler
  - etc.
- 言語 (処理系)
  - Eff
  - Multicore OCaml
  - Koka
  - etc.

どうやって実装されているの??

Low-level Manipulations

e.g.) 📎 libhandler, implemented in C

Low-level Manipulations

e.g.) 📎 libhandler, implemented in C

Algebraic Effects	$\mapsto$	libhandler
effect invocation	$\mapsto$	longjmp
effect handler	$\mapsto$	stack frame + ip
continuation	$\mapsto$	stack frame + ip

pros/cons

pros/cons

😄 低レベル実装

- FFI で様々な言語から呼び出せる
- 速い

pros/cons

- 😁 低レベル実装
  - FFI で様々な言語から呼び出せる
  - 速い
- 😓 実装が大変かつ限定的

pros/cons

- 😁 低レベル実装
  - FFI で様々な言語から呼び出せる
  - 速い
- 😓 実装が大変かつ限定的

$\Downarrow$

{処理系, ライブラリ}バックエンド向けか

Coroutines

e.g.)

- Multicore OCaml, implemented with fiber, in C [DWS+15]
- eff.lua , implemented with coroutine, in Lua

Coroutines

e.g.)

- Multicore OCaml, implemented with fiber, in C [DWS+15]
- eff.lua , implemented with coroutine, in Lua

Algebraic Effects		Coroutines
effect invocation	$\mapsto$	yield
effect handler	$\mapsto$	create & resume
continuation	$\mapsto$	coroutine

Coroutines

Algebraic Effects		Coroutines
effect invocation	$\mapsto$	yield
effect handler	$\mapsto$	create & resume
continuation	$\mapsto$	coroutine

Coroutines

Algebraic Effects		Coroutines
effect invocation	$\mapsto$	yield
effect handler	$\mapsto$	create & resume
continuation	$\mapsto$	coroutine

Coroutines

Algebraic Effects	$\mapsto$	Coroutines
	$\mapsto$	
effect handler	$\mapsto$	create & resume
continuation	$\mapsto$	coroutine

何回も呼び出したいけど

状態をコピーできない

Coroutines

Algebraic Effects $\mapsto$ Coroutines

effect handler $\mapsto$ create & resume

continuation $\mapsto$ coroutine

何回も呼び出したいけど

状態をコピーできない

coroutines をコピーするような操作がなければ
継続はワンショットに限定される

pros/cons

pros/cons

😁 さまざまな言語で実装可能
Coroutinesを持ってる言語は多い😉
Lua, Ruby, JS, Kotlin, Python, etc.

pros/cons

- 😁 **さまざまな言語で実装可能**
  Coroutinesを持ってる言語は多い😉
  Lua, Ruby, JS, Kotlin, Python, etc.
- 😅 **継続はワンショット**
  非決定計算とかは書けない

pros/cons

- 😁 さまざまな言語で実装可能
  Coroutinesを持ってる言語は多い😉
  Lua, Ruby, JS, Kotlin, Python, etc.
- 😅 継続はワンショット
  非決定計算とかは書けない
- 🤔 coroutineを複製する操作があれば……
  Multicore OCamlの `Obj.clone_continuation` の実装

Multiprompt shift/reset

Multiprompt shift/reset

e.g.) `racket/control` in Racket

```
(reset
  (+ 2 (reset
    (+ 3 (shift _k 4)))))

;; →* (reset (+ 2 4)) →* 6
```

Multiprompt shift/reset

e.g.) `racket/control` in Racket

```
(let ((p (make-continuation-prompt-tag))
      (q (make-continuation-prompt-tag)))
  (reset-at p
    (+ 2 (reset-at q
      (+ 3 (shift-at p _k 4))))))

;; →* (reset-at p (+ 2 (shift-at p _k 4)))
;; →* 4
```

Multiprompt shift/reset

e.g.) `racket/control` in Racket

対応するpromptまで飛んでいく

```
(let ((p (make-continuation-prompt-tag))
      (q (make-continuation-prompt-tag)))
  (reset-at p
    (+ 2 (reset-at q
      (+ 3 (shift-at p _k 4))))))

;; →* (reset-at p (+ 2 (shift-at p _k 4)))
;; →* 4
```

Multiprompt shift/reset

e.g.) 📎 Effekt, implemented in Scala

Multiprompt shift/reset

e.g.) 📎 Effekt, implemented in Scala

Algebraic Effects	$\mapsto$	Multiprompt shift/reset
effect operation	$\mapsto$	prompt tag
effect invocation	$\mapsto$	`shift-at`
effect handler	$\mapsto$	`reset-at`
continuation	$\mapsto$	continuation

pros/cons

pros/cons

- 😁 直感的で素直な対応
  実装しやすい
- 😁 effect の dynamic instantiation も対応
  multi-state など

pros/cons

- 😄 直感的で素直な対応
  実装しやすい
- 😄 effect の dynamic instantiation も対応
  multi-state など
- 😅 あまり一般的でない
  実装が少ない

Free Monad

Free Monad

```
type 'a free =
  | Pure: 'a → 'a free
  | Impure: 'arg * ('res → 'a free) → 'a free

let rec (≫=) op f =
  match op with
  | Pure x → f x
  | Impure (x, k) →
    Impure (x, fun y → k y ≫= f)
```

Free Monad

e.g.) 📎 Eff, implemented in OCaml

Free Monad

e.g.) 📎 Eff, implemented in OCaml

Algebraic Effects	$\mapsto$	Free Monad
effect invocation	$\mapsto$	Impure
effect handler	$\mapsto$	run
continuation	$\mapsto$	rhs of $(\gg=)$

pros

- 😁 Free の資産が使える
  [PSF+17] では equation rules や type-directed optimisation などを駆使して実行効率の良いコードを生成

cons

- 😅 ただの Free Monad
  monadic な書き方ができないとちょっとつらい
  Haskell の **`do`**, F#の computation expression, Scala の **`for`**, etc.

N-Barrelled CPS

Double-Barrelled CPS [Thi02] を拡張するといい感じに使えるのでは？

N-Barrelled CPS

Double-Barrelled CPS [Thi02] を拡張するといい感じに使えるのでは?

sort	number of continuations
(pure) CPS	1
👉 CPS + Exception	2

N-Barrelled CPS

Double-Barrelled CPS [Thi02] を拡張するといい感じに使えるのでは?

sort	number of continuations
(pure) CPS	1
CPS + Exception	2
CPS + Algebraic Effects	1 + number of effect handlers

N-Barrelled CPS

```
handler
| effect (Foo x) k -> k (x + x)
| effect (Bar y) k -> k (y * y)
| v -> v * 20
```

N-Barrelled CPS

```
handler
| effect (Foo x) k -> k (x + x)
| effect (Bar y) k -> k (y * y)
| v -> v * 20
```

$$\Downarrow$$

```
[
  (Value, fun v k -> k (v * 20))
  (Foo, fun x k -> k (x + x));
  (Bar, fun b k -> k (b * b))
]
```

N-Barrelled CPS

Algebraic Effects	$\mapsto$	N-Barrelled CPS
effect handler	$\mapsto$	(effect-id * handler) list
effect invocation	$\mapsto$	lookup &run
handler nesting	$\mapsto$	list concatenation

N-Barrelled CPS

N Barrels

Algebraic Effects	$\mapsto$	N-Barrelled CPS
effect handler	$\mapsto$	(effect-id * handler) list
effect invocation	$\mapsto$	lookup &run
handler nesting	$\mapsto$	list concatenation

N-Barrelled CPS

例

```
handle (perform (Foo 5)) with
| effect (Foo x) k → k (x + x)
| effect (Bar b) k → k (b * b)
| (*  value   *) v → v * 20
```

N-Barrelled CPS

例

```
handle (perform (Foo 5)) with
| effect (Foo x) k → k (x + x)
| effect (Bar b) k → k (b * b)
| (*  value   *) v → v * 20
```

⇓雰囲気で変換

```
(fun k_0 h_0 →
  (fun k_1 h_1 → k_1 5) (fun v_1 →
  (lookup_eff h_0 Foo v_1) (fun res_Foo →
  (lookup_val h_0 res_Foo k_0))) h_0
) (fun x → x)
  [ (VALUE, fun v k → k (v * 20));
    (Foo, fun x k → k (x + x));
    (Bar, fun b k → k (b - b)) ]
```

pros/cons

pros/cons

- 😁 CPS の資産が得られるかも

pros/cons

- 😁 CPS の資産が得られるかも
- 😕 グローバルな変換が必要

pros/cons

- 😁 CPS の資産が得られるかも
- 😕 グローバルな変換が必要

$\Downarrow$

処理系の中間表現向けかな

pros/cons

- 😁 CPS の資産が得られるかも
- 😕 グローバルな変換が必要

$$\Downarrow$$

処理系の中間表現向けかな

- ⭐ related) 📎 Koka
  compilililing to JS or C# via type-directed
  selective CPS [Lei16]

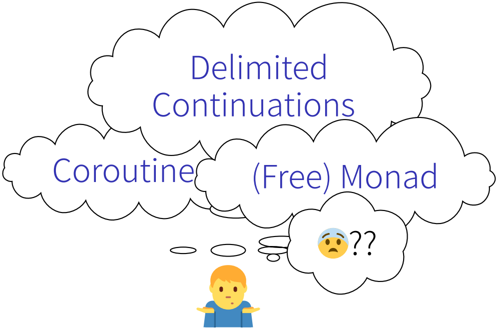
Good Point as Libraries

Good Point as Libraries

様々なコントロール抽象を Algebraic Effects という１つのインターフェースに落とし込める

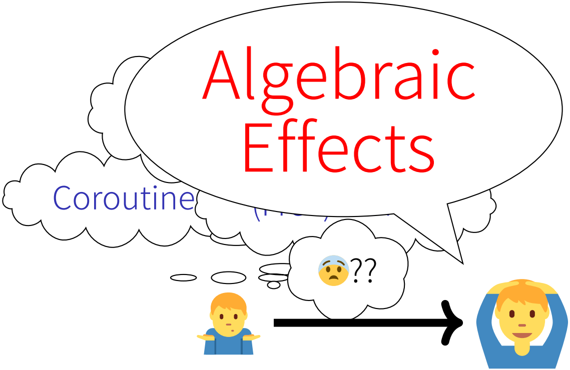
Good Point as Libraries

様々なコントロール抽象を Algebraic Effects という１つのインターフェースに落とし込める



Good Point as Libraries

様々なコントロール抽象を Algebraic Effects という１つのインターフェースに落とし込める



Conclusion

Conclusion

- Algebraic Effects の実装方法は様々

Conclusion

- Algebraic Effects の実装方法は様々
- 実装言語などに応じて使い分ける

Conclusion

- Algebraic Effects の実装方法は様々
- 実装言語などに応じて使い分ける
- ライブラリ化することでコントロール抽象の抽象化

Conclusion

- Algebraic Effects の実装方法は様々
- 実装言語などに応じて使い分ける
- ライブラリ化することでコントロール抽象の抽象化

もっといろいろな
言語で
Algebraic Effects
を!!

参考文献

[DWS+15] Stephen Dolan et al. Effective concurrency through algebraic effects. In: *OCaml Workshop*. 2015, p. 13.

[Lei16] Daan Leijen. Algebraic Effects for Functional Programming. Tech. rep. MSR-TR-2016-29, 2016, p. 15.

[PSF+17] Matija Pretnar et al. Efficient compilation of algebraic effects and handlers. In: *CW Reports, volume CW708* 32 (2017).

[Thi02] Hayo Thielecke. Comparing control constructs by double-barrelled CPS. In: *Higher-Order and Symbolic Computation*