

0から知った気になる **Algebraic Effects**

2019/09/30

びしょ〜じょ

本日の内容

Algebraic Effects

を

知った気にさせる

自己紹介



こんにちは、びしょ〜じょです

- 筑波大学大学院M2

Algebraic Effectsからコルーチンに変換する研究

- 株式会社HERPでエンジニア

We're hiring!

TSとかたまにHaskellを書いてる

まずはじめに

皆さんは**Algebraic Effects**知ってますか？ 

- 知ってるし書いたことがある
- 名前は聞いた
- 知らない

Algebraic Effectsを一言でいうと

限定継続が取得できる
例外およびハンドラ

限定継続🤔?

継続🤔

継続は分かりますか？👋

- わたしはschemerです
- はいはいコールバック関数ね
- 知らない

継続

コールバック関数です!!!!!! (完)

継続

例: ファイルを読み込み、結果をコールバック関数に渡す

```
readFile(file, res ⇒ { ..... })
```

継続

例: ファイルを読み込み、結果をコールバック関数に渡す

これ継続



```
readFile(file, res => { ..... })
```

継続

```
readFile(file, res ⇒ { ..... })
```

継続

```
readFile(file, res => { ..... })
```



```
const res = await promisify(readFile)(file);  
.....
```

繼續

```
readFile(file, res => { ..... })
```



```
const res = await promisify(readFile)(file);  
.....
```



```
promisify(readFile)(file)  
  .then(res => { ..... });
```

繼續

```
readFile(file, res => { ..... })
```

```
const res = await promisify(readFile)(file);  
.....
```

```
promisify(readFile)(file)  
  .then(res => { ..... });
```



The diagram illustrates the transformation of a callback function into an await statement and a promise chain. Three blue curved arrows show the mapping: the first arrow points from the `res => { }` callback in the top code block to the `res` variable in the middle code block; the second arrow points from the `readFile(file, ...)` call in the top code block to the `promisify(readFile)(file)` call in the bottom code block; the third arrow points from the `await` keyword in the middle code block to the `.then` method in the bottom code block.

継続

前の計算結果を使って実行する**残りの計算**

```
const t = f(10);  
const u = g(t);  
const v = h("aaa");  
.....
```

継続

前の計算結果を使って実行する残りの計算

$x \triangleright k \equiv k(x)$

```
const t = f(10);  
const u = g(t);  
const v = h("aaa");  
.....
```

継続渡しスタイル

```
                                f(10)  
▷ ((t) ⇒ g(t)  
▷ ((u) ⇒ h("aaa")  
▷ ((v) ⇒ .....  
)))
```


継続

前の計算結果を使って実行する残りの計算

$$x \triangleright k \equiv k(x)$$

```
const t = f(10);  
const u = g(t);  
const v = h("aaa");  
.....
```

継続渡しスタイル

```
f(10)  
▷ ((t) ⇒ g(t))  
▷ ((u) ⇒ h("aaa"))  
▷ ((v) ⇒ .....)  
)))
```

継続

前の計算結果を使って実行する残りの計算

$$x \triangleright k \equiv k(x)$$

```
const t = f(10);  
const u = g(t);  
const v = h("aaa");  
.....
```

継続渡しスタイル

```
                                f(10)  
▷ ((t) ⇒ g(t))  
▷ ((u) ⇒ h("aaa"))  
▷ ((v) ⇒ .....)  
)))
```

継続

前の計算結果を使って実行する残りの計算

$$x \triangleright k \equiv k(x)$$

```
const t = f(10);  
const u = g(t);  
const v = h("aaa");  
.....
```

継続渡しスタイル

```
                                f(10)  
▷ ((t) ⇒ g(t))  
▷ ((u) ⇒ h("aaa"))  
▷ ((v) ⇒ .....)  
)))
```

継続

継続が使えると...

継続

継続が使えると...

コントロールを扱う機能がユーザレベルで実装できる

- 大域脱出
- バックトラッキング
- マルチスレッディング

などなど

継続 - **call/cc** (call with current-continuation)

⚠️ しばらく[Racket](#)で行きます

継続 - `call/cc` (call with current-continuation)

⚠️ しばらく[Racket](#)で行きます

```
(call/cc fn)
```


呼ばれた位置からの**継続**を関数としてfnに渡す

継続が呼ばれたあとは`call/cc`には戻ってこない

継続 - **call/cc** (call with current-continuation)

```
(let [(x (call/cc (λ (k)
                    (+ 2 (k 4)))))]
  (displayln x))
```


継続 - **call/cc** (call with current-continuation)



```
(let [(x (call/cc (λ (k)
                    (+ 2 (k 4)))))]
  (displayln x))
```

☑ 呼ばれた位置からの継続を関数として渡す

継続 - **call/cc** (call with current-continuation)

```
(let [(x (call/cc (λ (k)
                   (+ 2 (k 4)))))]
  (displayln x))
```

```
⇒ (let [(x 4)] (displayln x))
⇒ displays "4"
```

- ☑ 呼ばれた位置からの継続を関数として渡す
- ☑ 継続が呼ばれたあとは**call/cc**には戻ってこない

継続 - call/ccと大域脱出

```
;; Racket  
(define (div-fail xs fallback)  
  (call/cc (λ (k)  
    (map (λ (e)  
      (if (= e 0)  
          (k fallback)  
          (/ e 2))))  
    xs))))
```

継続 - call/ccと大域脱出

```
;; Racket
(define (div-fail xs fallback)
  (call/cc (λ (k)
    (map (λ (e)
      (if (= e 0)
          (k fallback)
          (/ e 2))))
    xs))))
```



継続を取得

継続 - call/ccと大域脱出

```
;; Racket
(define (div-fail xs fallback)
  (call/cc (λ (k)
    (map (λ (e)
      (if (= e 0)
          (k fallback)
          (/ e 2)))
      xs))))
```



継続を取得



継続に
fallbackを
渡して脱出

継続 - call/ccと大域脱出

```
[let  
  [(x (div-fail '(3 4 5 6) '(1)))]  
  (displayln x)]
```

継続 - call/ccと大域脱出

```
[let  
  [(x (div-fail '(3 4 5 6) '(1)))]  
  (displayln x)]
```

⇒ *displays* "'(1 3/2 2 5/2)"

継続 - call/ccと大域脱出

```
[let  
  [(y (div-fail ' (1 2 0 3) ' (1))))]  
  (displayln y)]
```


継続 - call/ccと大域脱出

```
[let  
  [(y (div-fail '(1 2 0 3) '(1)))]  
  (displayln y)]
```

div-failから見た継続

継続 - call/ccと大域脱出

```
[let  
  [(y (div-fail '(1 2 0 3) '(1)))]  
  (displayln y)]
```

⇒ [let [(y '(1))] (displayln y)]

⇒ *displays* "'(1)'"

限定 継続

限定 継続

- **call/cc**

プログラムの残りすべてを
継続として利用

限定 継続

- **call/cc**

プログラムの残りすべてを
継続として利用

⇒ ちょっと使いづらい😅

限定 継続

- call/cc

プログラムの残りすべてを
継続として利用

⇒ ちょっと使いづらい 😅

- 限定継続

プログラムの残りの特定の範囲
を継続として利用

⇒ 取り回しが良い 🧑

限定 継続 - `shift/reset`

限定 継続 - `shift/reset`

```
(shift k e)
```

式eのスコープ内で継続`k`を利用する

限定 継続 - `shift/reset`

```
(shift k e)
```

式eのスコープ内で継続`k`を利用する

```
{reset e}
```

`new`

継続の範囲をe内に限定する

限定 継続 - `shift/reset`

`(shift k e)`

式eのスコープ内で継続kを利用する

`(reset e)`

new

継続の範囲をe内に限定する

その他の限定継続演算子: `control/prompt`, `cupto`, etc.

限定 継続 - **shift/reset**



```
;; Racket
(displayln [let [(f {reset
  (string-append
    (shift k (λ () (k "hello")))
    " world"})}]
  (f)])
```

浅井健一, [shift/resetプログラミング入門](#) を参考

限定 継続 - `shift/reset`



継続の範囲を限定



```
;; Racket  
(displayln [let [(f {reset  
  (string-append  
    (shift k (lambda () (k "hello")))  
    " world"})}]  
  (f)])]
```

浅井健一, `shift/reset`プログラミング入門 を参考

限定 継続 - shift/reset



```
;; Racket
(displayln [let [(f {reset
  (string-append
    (shift k (λ () (k "hello")))
    " world"})}]
  (f)])
⇒ (displayln
   [let [(f (λ () (k "hello")))] (f)])
```

shiftの結果が返る

浅井健一, shift/resetプログラミング入門 を参考

限定 継続 - `shift/reset`



```
;; Racket
(displayln [let [(f {reset
  (string-append
    (shift k (λ () (k "hello")))
    " world"})}]
  (f)])
⇒ (displayln
   [let [(f (λ () (k "hello")))] (f)])
⇒ displays "hello world"
```

浅井健一, [shift/resetプログラミング入門](#) を参考

限定 継続

限定継続が使えると...

- **call/cc !!**
- 型付きprintf
- Stateモナド

限定 継続

限定継続が使えると...

- `call/cc` モナド全般

A monadic framework for delimited continuations

例外+ハンドラ

例外+ハンドラ

皆さん**例外**は分かりますか？

- MonadError
- もちろん知ってる

例外+ハンドラ - try-catch

これは皆さんご存知try-catch (OCamlではtry-with)

```
:: OCaml  
try raise Not_found with  
| Not_found →  
  print_endline "not found"
```

例外+ハンドラ - try-catch

これは皆さんご存知try-catch (OCamlではtry-with)

- 例外が起きるとハンドラにジャンプする
- 例外発生位置からの**残りの計算は破棄**される

```
;; OCaml  
try raise Not_found with  
| Not_found →  
  print_endline "not found"
```

Algebraic Effectsを一言でいうと(再)

限定継続が取得できる
例外およびハンドラ

Algebraic Effectsを一言でいうと(再)

限定継続が取得できる
例外およびハンドラ

Algebraic Effectsを一言でいうと(再)

限定継続が取得できる
例外およびハンドラ

Algebraic Effects

歷史的經緯:

Algebraic Effects(2003) + Effect Handlers(2012)

Algebraic Effects = 限定継続 + 例外&ハンドラ

歴史的経緯:

Algebraic Effects + **Effect Handlers**

= *Algebraic Effects and Handlers*

略して "Algebraic Effects" または "Algebraic Effect Handlers"

Algebraic Effects = 限定継続 + 例外&ハンドラ

正しくは *Algebraic Effects and Handlers*

- 計算エフェクトを例外のthrowのように発生
- ハンドラにジャンプ
- ハンドラのスコープ内の継続を同時に取得してエフェクト発生位置から復帰できる

Algebraic Effects = 限定継続 + 例外&ハンドラ

- 計算エフェクトを統一的に扱える
 - モナドよりもcompositional
- インターフェースと実装の分離
 - エフェクトのシグネチャとハンドラ
 - モジュラーなプログラミング
(i.e. Dependency Injection)

Algebraic Effects - Option Monad

```
effect Option : 'a option → 'a
```

```
handle
```

```
  let ox : int option = lookup "key" assoc in  
  let x : int = perform (Option ox) in  
    Some (x + 5)
```

```
with
```

```
| effect (Option None) _k → None  
| effect (Option (Some v)) k → k v
```

Algebraic Effects - Option Monad

```
effect Option : 'a option → 'a
```



エフェクトを定義

```
handle
```

```
  let ox : int option = lookup "key" assoc in  
  let x : int = perform (Option ox) in  
    Some (x + 5)
```

```
with
```

```
| effect (Option None) _k → None  
| effect (Option (Some v)) k → k v
```

Algebraic Effects - Option Monad

```
effect Option : 'a option → 'a
```

エフェクトを定義

```
handle
```

エフェクトを発生

```
let ox : int option = lookup key assoc in
```

```
let x : int = perform (Option ox) in
```

```
Some (x + 5)
```

```
with
```

```
| effect (Option None) _k → None
```

```
| effect (Option (Some v)) k → k v
```

Algebraic Effects - Option Monad

```
effect Option : 'a option → 'a
```

エフェクトを定義

```
handle
```

エフェクトを発生

```
let ox : int option = lookup key assoc in
```

```
let x : int = perform (Option ox) in
```

```
Some (x + 5)
```

継続を破棄
(c.f.例外処理)

```
with
```

```
| effect (Option None) _k → None
```

```
| effect (Option (Some v)) k → k v
```

Algebraic Effects - Option Monad

```
effect Option : 'a option → 'a
```

エフェクトを定義

```
handle
```

エフェクトを発生

```
let ox : int option = lookup key assoc in
```

```
let x : int = perform (Option ox) in
```

```
Some (x + 5)
```

Someを剥がして
継続に値を渡す

```
with
```

```
| effect (Option None) _k → None
```

```
| effect (Option (Some v)) k → k v
```


Algebraic Effects - ハンドラのネストと合成性

ハンドルできないエフェクトはtry-catch同様に
外側のハンドラまでジャンプするので...

- ハンドラの**ネスト**で複数のエフェクトをハンドル
- リテラル的にも複数のエフェクトをハンドル可能

エフェクトのハンドリングが*compositional* におこなえる

Algebraic Effects - ハンドラのネストと合成性


```
effect ReadLine : () → string
```

```
let with_io_read th =  
  handle th () with  
  | effect (ReadLine ()) k →  
    k (read_line ())
```

```
let with_option th = .....
```

Algebraic Effects - ハンドラのネストと合成性

2種類のエフェクトを
発生



```
let read_int_opt () =  
  let line = perform (ReadLine ()) in  
  perform (Option (int_of_string_opt line))  
  
let int_of_string_opt  
  : string → int option  
= .....
```

Algebraic Effects - ハンドラのネストと合成性

```
let app () =  
  let a = read_int_opt () in  
  let b = read_int_opt () in  
  Some (a + b)
```

Algebraic Effects - ハンドラのネストと合成性

```
let app () =  
  let a = read_int_opt () in  
  let b = read_int_opt () in  
  Some (a + b)
```

```
let main () =  
  with_option (fun () →  
    with_io_read app)
```

Algebraic Effectsで何ができる？

- マルチプロセス
- Dependency Injection
- コールバック地獄から手続き的記述へ
(c.f. Promise ~~> async/await)
- Goのdefer

などなど

Algebraic Effectsを使おう

- 言語

- [Eff](#)
- [Koka](#)
- [Multicore OCaml](#)
- [Frank](#)

...などなど

- ライブラリ

- [eff.lua](#) for Lua (拙作)
- [ruff](#) for Ruby (拙作)
- [effective-rust](#) for Rust
- [gauche-effects](#) for Scheme

Algebraic Effectsを実装しよう

様々な実装方法

- コールスタックを直接触る
[libhandler](#), 言語処理系自体を実装
- 限定継続
gauche-effects, [effekt](#), 『Eff Directly in OCaml』
- コルーチン
eff.lua, ruff, effective-rust, 弊研究

まとめ

- Algebraic Effectsは**限定継続**の取れる**例外**
- Algebraic Effectsはなんか色々できて強い
- Algebraic Effectsは**実はすぐに触れる**

まとめ

話していないこと:

- value handler
- 型システム
- 『What is *algebraic* ?』
- 様々な流派と表現力

などなど

まとめ

話してないこと:

- value handler
- 型システム
- 『What is *algebraic* ?』
- 様々な流派と表現力

などなど



宿題です!!!

参考文献

- Andrej Bauer and Matija Pretnar. [Programming with algebraic effects and handlers](#). In: Journal of Logical and Algebraic Methods in Programming 84.1 (2015), pp. 108–12
- 浅井健一. [shift/resetプログラミング入門](#). In: ACM SIG-PLAN Continuation Workshop 2011 (2011)
- びしょ〜じょ. [How do you implement algebraic effects?](#). In: [effect system勉強会](#) (2019)
- R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. [A monadic framework for delimited continuations](#). In: Journal of Functional Programming 17.6 (2007), pp. 687-730.
- Andrej Bauer. [What is algebraic about algebraic effects and handlers?](#). In: arXiv preprint arXiv:1807.05923 (2018)
- Oleg Kiselyov and KC Sivaramakrishnan. [Eff directly in OCaml](#). In: ACM SIGPLAN Workshop on ML 2016 Sep. (2016)

計算エフェクト

a.k.a. 副作用 誤解を与えかねないのでしばしば言い換えられる
やりたい計算(`Num a ⇒ a`)に対して本道でないもの(`Maybe`)

$$\text{Num } a \Rightarrow a \quad \Leftrightarrow \quad \text{Num } a \Rightarrow \text{Maybe } a$$

モナドを使うと計算エフェクトの操作を隠蔽できる

```
class Applicative m ⇒ Monad m where
  (≫=) :: m a → (a → m b) → m b
  return :: a → m a
```

Extensible effectsとの関連性

Extensible Effectsは**type-directed**なAlgebraic Effectsの埋め込みだと思う

⇔ 限定継続、コルーチンはexpression-oriented

React Hooksとの関連性.....🤔

無さそう、AEで実装できるがメリットが少ない

- Reactが隠蔽していた実装を自分でやる必要がある
- 一般に、継続のランタイムコストは馬鹿にならない
- 継続を末尾位置で必ず呼ぶ(i.e. 複製、破棄などしない)ので旨味がない

React Hooks	→	Algebraic Effects
useHoge	→	Hogeエフェクトの発生
(React内部実装)	→	ハンドラ
次のレンダリング?	→	継続

Algebraic Effects - Dependency Injection 🧠💥

- Dependency Injection

インタフェースに対して実装をあとから入れる

- Algebraic Effects

エフェクトのシグネチャ(インタフェース)に対して

ハンドラ(実装)をあとから入れる

Algebraic Effects - Dependency Injection 🧠

```
effect GetUsers : () → user list
```

```
let mock_get_users th =  
  handle th () with  
    | effect (GetUsers ()) k →  
      k (List.create ~size:10 ~val:dummy_data)
```

```
let prod_get_users th =  
  handle th () with  
    | effect (GetUsers ()) k → k(DB.get_users ())
```

Algebraic Effects - Dependency Injection 🤖

```
effect GetUsers : () → user list
```

ユーザを取得する
エフェクトを定義

```
let mock_get_users th =  
  handle th () with
```

```
  | effect (GetUsers ()) k →  
    k (List.create ~size:10 ~val:dummy_data)
```

ダミーデータを渡す

```
let prod_get_users th =  
  handle th () with
```

実際のデータを渡す

```
  | effect (GetUsers ()) k → k(DB.get_users ())
```

Algebraic Effects - Dependency Injection 🤯

```
let app () =  
    let users = perform (GetUsers ()) in  
    .....  
  
let test_main () =  
    mock_get_users app  
  
let prod_main () =  
    prod_get_users app
```

ハンドラの切り替えで
実装を選べる