

びしょ〜じょ

■ やること



を伝道

何



が使える言語
の活用事例

研究のご紹介

先日 でポスター発表した内容を紹介シマス

■ 目次

■ 自己紹介



こんにちは、びしょ〜じょ
です。

- ▶ 大学大学院で をやっている
プログラム言語や型とか検証などの研究室
- ▶ 少し前は にお熱だった





イメージとしては**継続**を持てる**例外**

イメージとしては**継続**を持てる**例外** 例

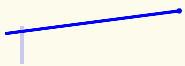
```
effect Say      : string → unit
effect Twice    : unit    → unit

handle (
  perform (Say "Hello");
  perform (Twice ());
  perform (Say "world")
) with
| x → x
| effect (Twice ()) k → k (); k ()
| effect (Say msg) k →
  print_endline msg; k ()
```


イメージとしては**継続**を持てる**例外** 例

```
effect Say      : string → unit
effect Twice    : unit    → unit

handle (
  perform (Say "Hello");
  perform (Twice ());
  perform (Say "world")
) with
| x → x
| effect (Twice ()) k → k (); k ()
| effect (Say msg) k →
  print_endline msg; k ()
```



イメージとしては**継続**を持てる**例外** 例

```
effect Say      : string → unit
effect Twice    : unit    → unit

handle (
  perform (Say "Hello");
  perform (Twice ());
  perform (Say "world")
) with
| x → x
| effect (Twice ()) k → k (); k ()
| effect (Say msg) k →
  print_endline msg; k ()
```

イメージとしては**継続**を持てる**例外** 例

```
effect Say      : string → unit
effect Twice    : unit    → unit

handle (
  perform (Say "Hello");
  perform (Twice ());
  perform (Say "world")
) with
| x → x
| effect (Twice ()) k → k (); k ()
| effect (Say msg) k →
  print_endline msg; k ()
```

イメージとしては**継続**を持てる**例外** 例

```
effect Say      : string → unit
effect Twice    : unit    → unit

handle (
  perform (Say "Hello");
  perform (Twice ());
  perform (Say "world")
) with
| x → x
| effect (Twice ()) k → k (); k ()
| effect (Say msg) k →
  print_endline msg; k ()
```

● の型

なにこれ

```
effect Say : string → unit
```

● の型

なにこれ

```
effect Say : string → unit
```

例

```
E[perform (Say "Hello")]
```

● の型

なにこれ

```
effect Say : string → unit
```

例

```
E[perform (Say "Hello")]
```

```
: string
```

● の型

なにこれ

```
effect Say : string → unit
```

例

```
E[perform (Say "Hello")]
```

```
: string
```

```
: unit
```


● の型

なにこれ

```
effect Say : string → unit
```

例



```
E[perform (Say "Hello")]  
: unit → 'a
```

継続の型

```
: string
```

```
: unit
```

● の型

なにこれ

```
effect Say : string → unit
```

例



```
E[perform (Say "Hello")]  
: unit → 'a
```

継続の型



```
: string
```

: unit

完全に理解した



● デモ


```
effect Say      : string → unit
effect Twice    : unit    → unit


handle (
  perform (Say "Hello");
  perform (Twice ());
  perform (Say "world")
) with
| x → x
| effect (Twice ()) k → k (); k ()
| effect (Say msg) k →
  print_endline msg; k ()
```

● ポイント

- 定義と実装の**分離**
 - ハンドラの**合成**
 - **限定継続**が使える
- } モジュラーなプログラミングを支援
- } 例外より強力

- 
-
- 
- ベースの、型推論
 - 処理系やライブラリ など実装多数

- 
- 製
 - が型で表される
- ```
println : a -> console ()
```
- 以外にもおもしろ機能

- 
- が を
  - 継続が の を持つ

---

<https://www.eff-lang.org/>

<https://koka-lang.github.io/koka/>

<http://ocaml-labs.io/doc/multicore.html>

## モナド風

```
module State(S : sig type t end)
= struct
 type t = S.t

 effect Put : t → unit;;
 effect Get : t

 let run init f =
 init |> match f () with
 | x → (fun s → (s, x))
 | effect (Put s') k →
 (fun s → continue k () s')
 | effect Get k →
 (fun s → continue k s s)
 end

 effect Log : int → unit
 let log msg = perform @@ Log msg
```

```
try begin
 let module S1 =
 State(struct
 type t = int
 end) in
 let open S1 in
 let incr () =
 perform (Put (perform Get + 1))
 in
 run 0 @@ fun () →
 incr ();
 log @@ perform Get;
 incr ();
 log @@ perform Get
end
with effect (Log msg) k →
 print_int msg; continue k ()
```

## も実装できるぞ

```
module DelimccOne = struct
 type 'a prompt = {
 take : 'b. (('b → 'a) → 'a) → 'b;
 push : (unit → 'a) → 'a;
 }

 let new_prompt (type a) : unit → a prompt = fun () →
 let module M = struct
 effect Prompt : (('b → a) → a) → 'b
 end in
 let take f = perform (M.Prompt f) in
 let push th =
 try th () with effect (M.Prompt f) k → f @@ continue k
 in
 {take; push}

 let push_prompt {push} = push
 let take_subcont {take} = take

 let shift0 p f = take_subcont p @@ fun k → f k
end
```

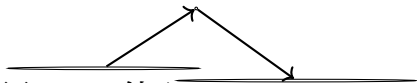




# ■ 研究紹介

---

による の実装



のある言語で が使える

👉 の類似の変換、の既存の変換を参考にする



に基づき先程の実装の逆をやる  
直感としては→

→→

```
module Translate(D: DELIMCC) : sig
 type ('a, 'b) free
 type 'a thunk = unit → 'a
 val newi : unit → 'a D.prompt
 val op : ('a, 'b) free D.prompt → 'a → 'b
 val handler : ('g, 'g) free D.prompt →
 ('g → 'o) → ('g * ('g → 'o) → 'o) → 'g thunk → 'o
 val handle : ('a thunk → 'b) → 'a thunk → 'b
end = struct

end
```

継続の を が引き継いでることの証明は  
まだ無い。で型によって示す予定



## による変換を使う による実装

の `coroutine` 現在の を表す `current`

※ただし はない が持つ限定継続は直前の

```
function sr.reset(th)
 local l = coro.create(
 function(_)
 return (function(y)
 return function(_)
 return y
 end
 end)(th())
 end)

 return coro.resume(l)()
end
```

```
function sr.shift(f)
 local k = coro.current

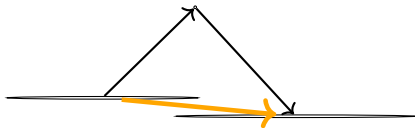
 return coro.yield(function(_)
 return sr.reset(function()
 return f(k)
 end)
 end)
end

function sr.throw(k, e)
 return coro.resume(k, e)()
end
```

## ● 課題、今後の予定

---

- ▶ の保証 先述
- ▶ による変換の対象の を で再定義
- ▶ を経由しないダイレクトな変換を考える





## ■ まとめ

---

が楽しい  
や にも  
関連のトピック

☒ 使おう  
インプリいろいろ  
なければ自作も可  
研究やってます

---

<https://icfp18.sigplan.org/event/mlfamilyworkshop-2018-papers-programming-with-abstract-algebraic-effects>

<https://icfp18.sigplan.org/event/icfp-2018-papers-versatile-event-correlation-with-algebraic-effects>

おわり



## ■ 参考文献

---