ワンショット代数的効果から非対称コルーチンへの変換

河原 悟, 亀山 幸義

PRO2019-01

June 6, 2019

ワンショット代数的効果から非対称コルーチンへの変換

背景
コルーチン
　コルーチンの合成性
代数的効果
研究
　操作の対応
実装
　応用
まとめと課題
変換
関連研究
対称コルーチンと非対称コルーチン
ワンショット継続とコルーチンの複製

背景

コルーチン

- 😄 様々な言語が持っている
  Lua, Ruby, C#, Kotlin, etc.
- 😄 強力なコントロール抽象
  async/await、イテレーション、etc.

合成性 (composability) に欠ける

背景

代数的効果

- インターフェースと実装を分離できる
  モジュール性、合成性が高い
- いくつかのコントロール抽象との関係が知られている
  代数的効果 ↦ 限定継続
  代数的効果 ↦ Free モナド

コルーチンにも落とし込めないだろうか？

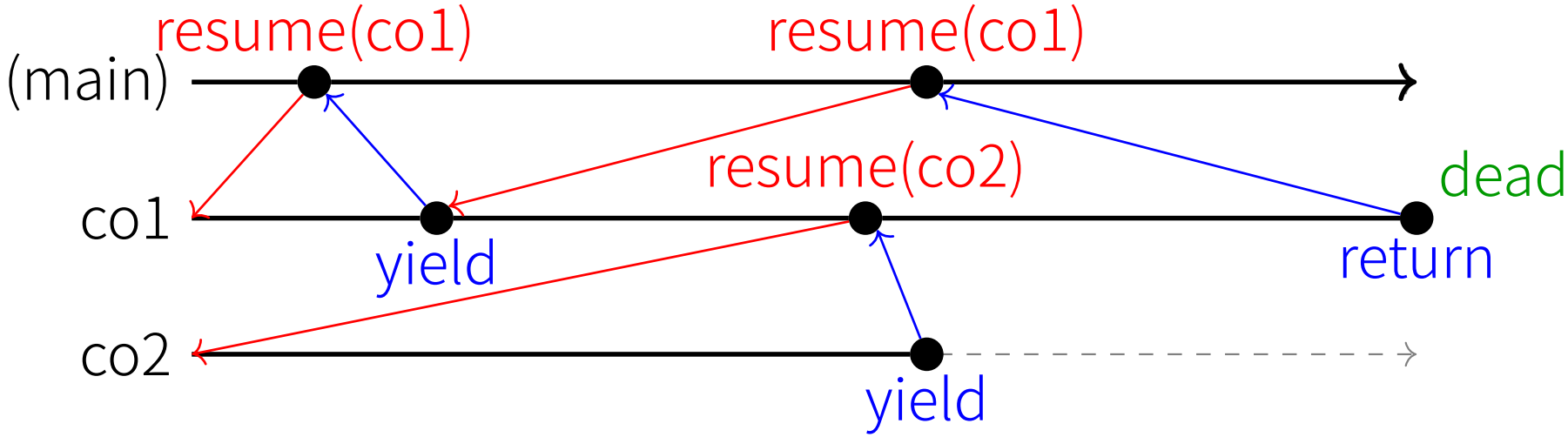
研究

目的: コルーチンを持つ言語でも
もっと合成性の高いコントロール抽象を使いたい
手段: 代数的効果をコルーチンに落とし込む
内容: 代数的効果からコルーチンへの変換
代数的効果とコルーチンの操作を対応づける

コルーチン

一時停止、リジュームのできるサブルーチン

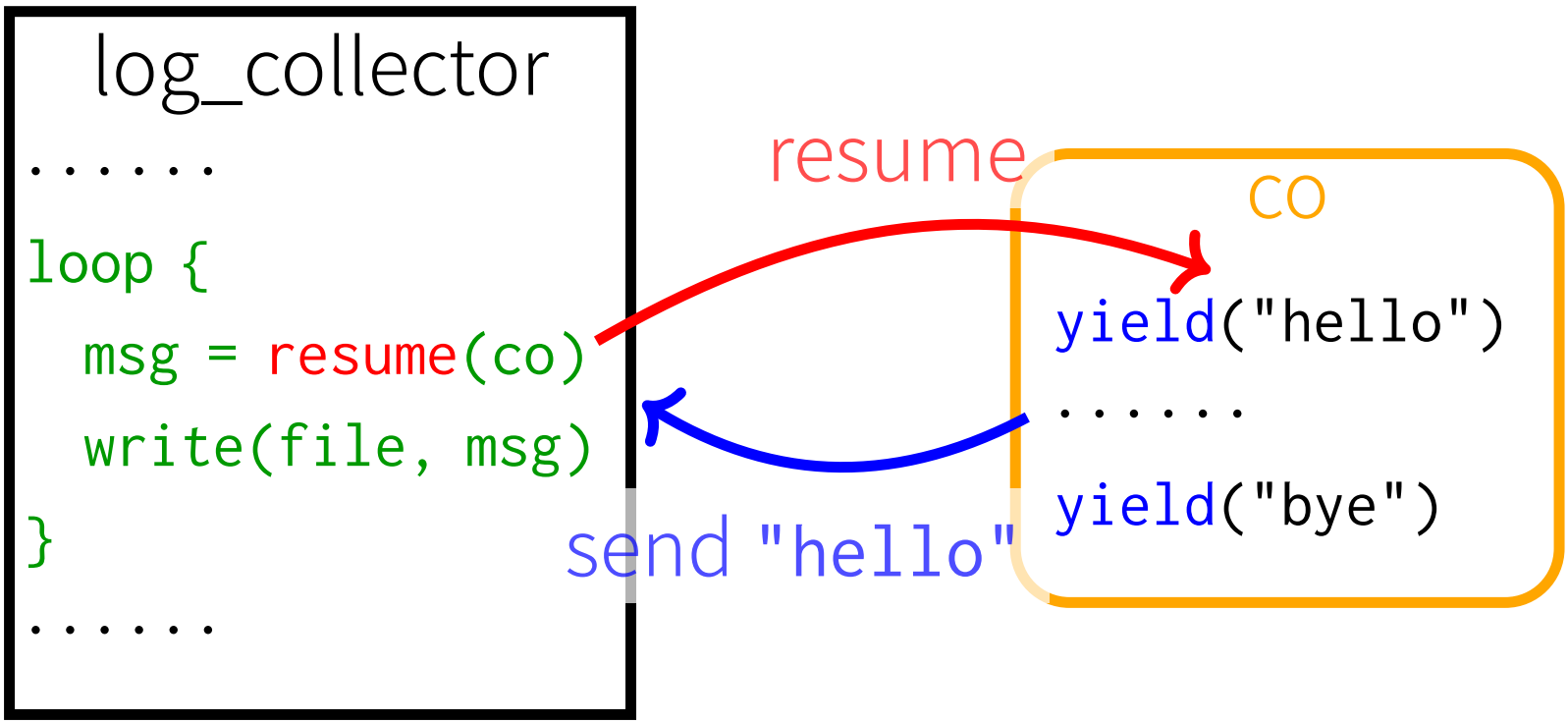
- **非対称コルーチン**
  呼び出し (resume)、リジューム元に戻る (yield)



- 対称コルーチン
  resume のみ

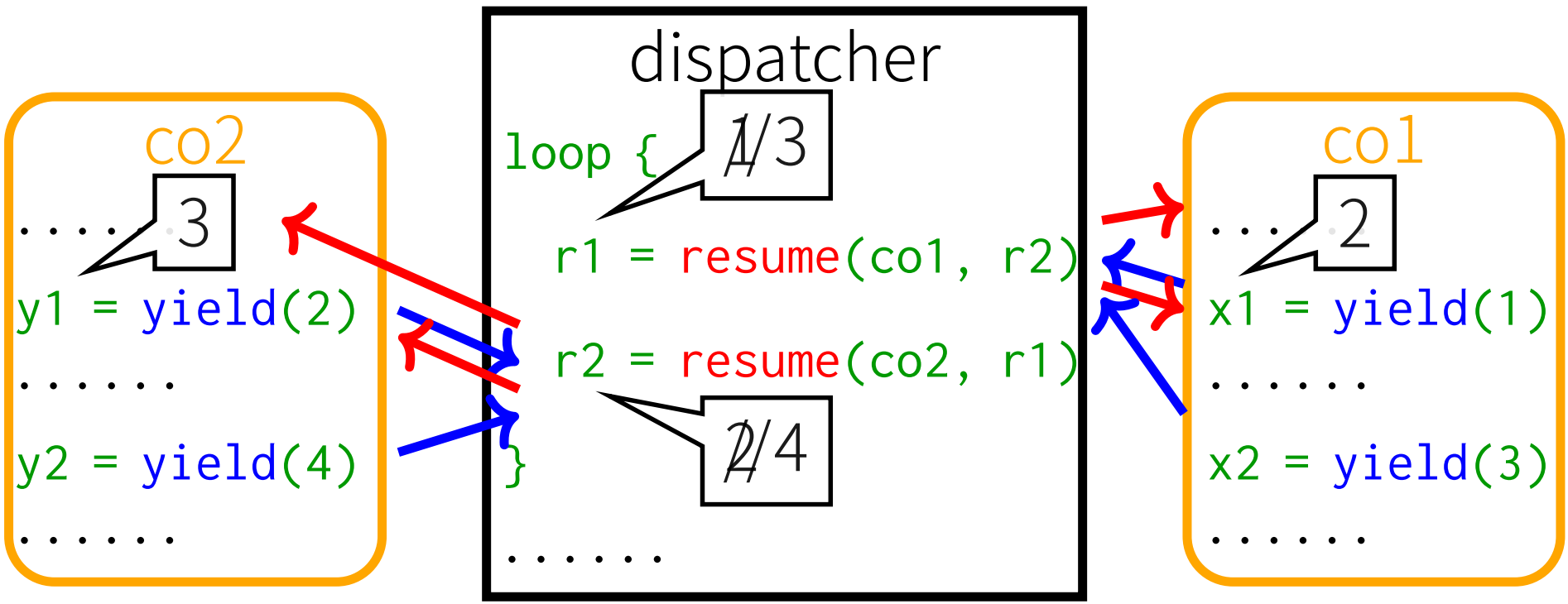
コルーチン

e.g.) ログの送信/収集



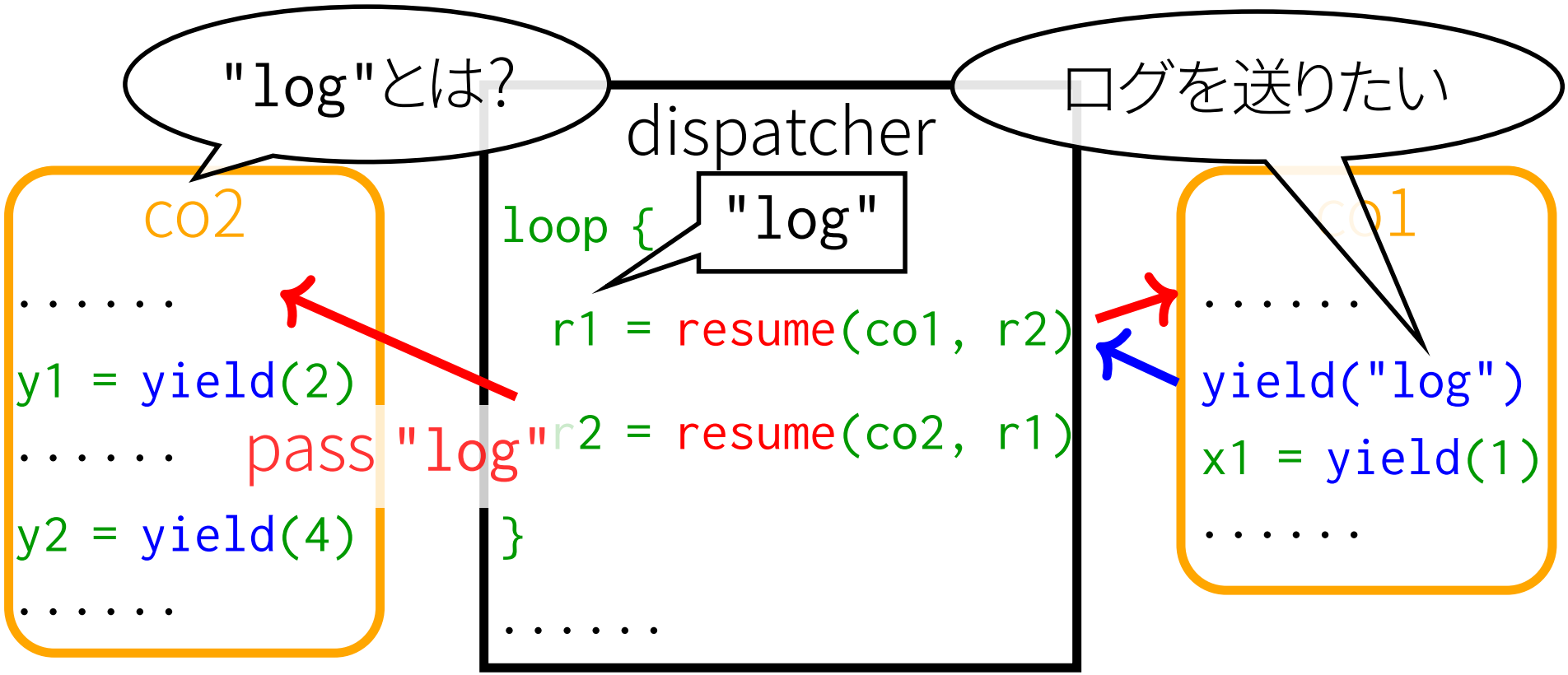
コルーチン

e.g.) スレッド間の値の受け渡し



コルーチンの合成性

ログの送受信 + スレッド間の値の受け渡し……？



コルーチンの合成性

😟 すぐに合成できない!!

異なる使い方の yield と resume を
1 つのコルーチンで利用できない

- yield の引数にタグを付けて
- resume が受け取ってタグで処理を分岐して
- etc.

工夫が必要

代数的効果

計算エフェクト (I/O, 例外, etc.) を扱う言語機能

- 圏論に基づき発展、理論や実用面でホットな分野
- 高レベルなエフェクト抽象
- 例外処理のようにエフェクトの発生をハンドル
- 継続 (コントロールの状態) を同時に取得

⇒ エフェクト抽象+ハンドラにより、
モジュール性+合成性の高いプログラミングが可能

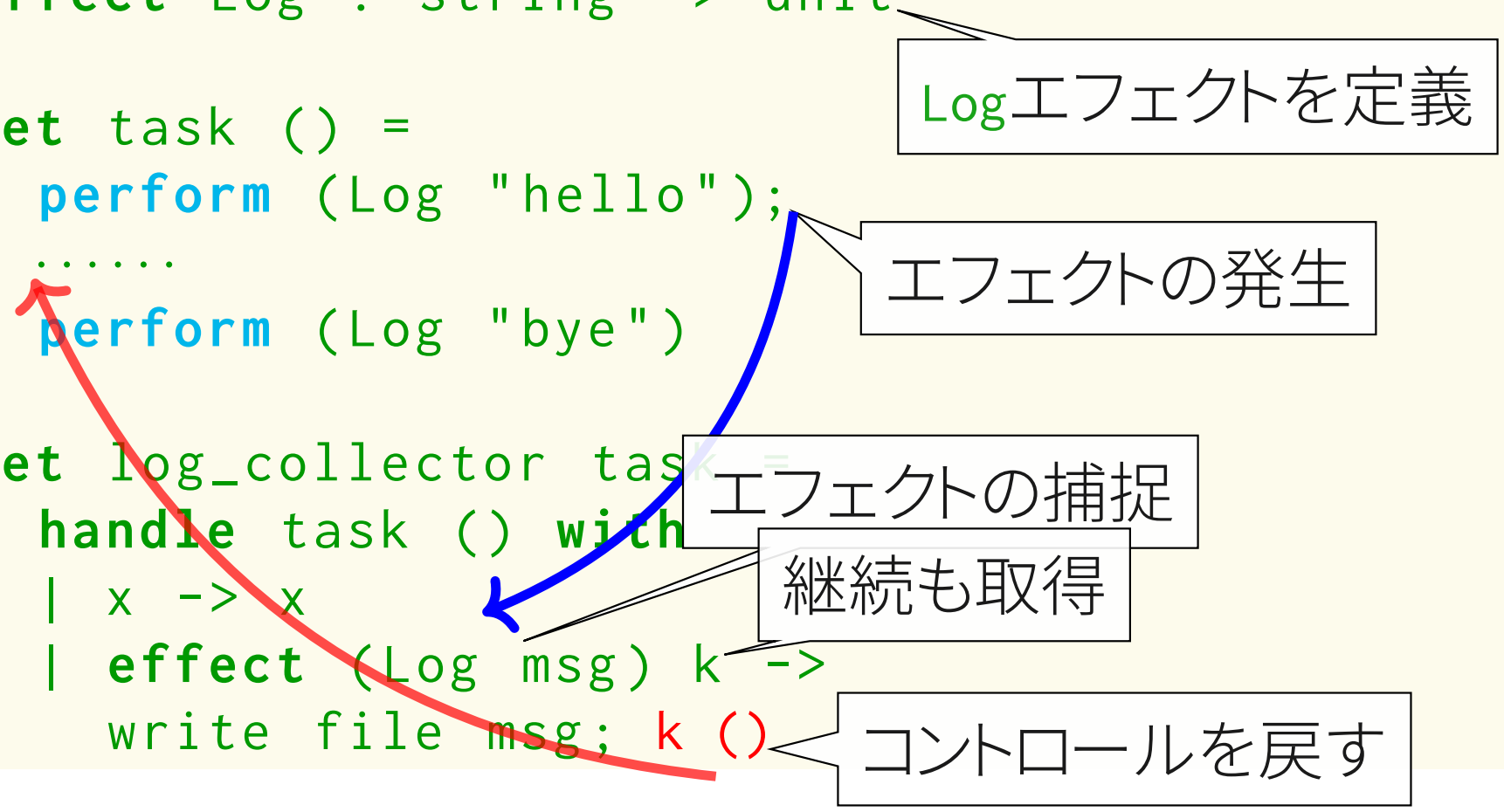
代数的効果

e.g.) ログの送信/収集

```
effect Log : string -> unit

let task () =
  perform (Log "hello");
  ......
  perform (Log "bye")

let log_collector task =
  handle task () with
  | x -> x
  | effect (Log msg) k ->
    write file msg; k ()
```



代数的効果

e.g.) スレッド間の値の受け渡し

```
effect Send : (int * int) -> int

let dispatcher init task1 task2 =
  let rec tasks = [| (fun _ -> go task1);
                     (fun _ -> go task2) |]
  and go task =
    handle task init with
    | x -> x
    | effect (Send (idx, x)) k ->
      tasks[idx] := k;
      tasks[(idx + 1) % 2] x
  in go task1
```

Sendエフェクトを定義

taskをハンドル

インデックス、値、継続を取得

継続を保存

他方のタスクを再開

代数的効果

ログの送受信 + スレッド間の値の受け渡し

```
let co1 init =
  ......
  perform (Log "msg");
  let x1 = perform (Send(0, 1)) in
  ......

let co2 init =
  ......
  let y1 = perform (Send(1, 2)) in
  ......
  let y2 = perform (Send(1, 4)) in
  ......
```

ログも送る、値も渡す

代数的効果

ログの送受信 + スレッド間の値の受け渡し😎

dispatcherにハンドルされた状態でコントロールを戻せる

```
log_collector (fun () ->
  dispatcher 0 co1 co2)
```

Logエフェクトは素通し

😄 合成が非常に簡単!

研究目的

非対称コルーチンを持つ言語でも、
もっと合成性の高いコントロール抽象を使いたい

操作の対応

代数的効果	$\mapsto$	非対称コルーチン
エフェクトの発生	$\mapsto$	yield
エフェクトのハンドル	$\mapsto$	resume
ハンドルされる式	$\mapsto$	コルーチン
継続	$\mapsto$	コルーチン

操作の対応

代数的効果	↦	非対称コルーチン
エフェクトの発生	↦	yield
ハンドルされる式	↦	コルーチン
継続	↦	コルーチン

何回も呼び出せる

コピーできない

継続の実行を**ワンショットに限定**することで
操作を正しく対応付けられる!

研究

✅変換を定義

$$\lambda_{\text{eff}} \mapsto \lambda_{\text{ac}}$$

ワンショット 代数的効果	$\mapsto$	非対称コルーチン
エフェクトの発生	$\mapsto$	yield
エフェクトのハンドル	$\mapsto$	resume
ハンドルされる式	$\mapsto$	コルーチン
ワンショットの継続	$\mapsto$	コルーチン

実装

変換結果に基づきワンショット代数的効果ライブラリを実装

- https://github.com/Nymphium/eff.lua
- Lua 言語で実装
  第一級関数、非対称コルーチンが標準ライブラリ
- 代数的効果の応用例で動作を確認
  Multicore OCaml のチュートリアル [*1] を実装
  - Same Fringe 問題
  - generator
  - async/await
  - state

[*1] https://github.com/ocamllabs/ocaml-effects-tutorial

応用

ライブラリを読み込むだけで利用可能

```
local Send = inst()
local Log = inst()

local co1 = function(init)
  ......
  perform(Log("msg"))
  local x1 = perform(Send(1, 10))
  ......
end
```

応用

Lua の table を用いてハンドラを直感的に定義可能

```
local log_collector = function(task)
  return handlers({
    function(x) return x end,
    [Log] = function(k, msg)
      write_to(log_file, msg)
      return k()
    end
  })(task)
end
```

まとめと課題

- 🤯 **ワンショット代数的効果から非対称コルーチンへの変換を定義した**
  合成性の高いコントロール抽象が使える!
- 🤯 **変換の結果に基づき、ワンショット代数的効果のライブラリを実装した**
  Lua で代数的効果が使える!
- ❓ **継続のワンショット性**
  - λ_{eff} が継続を 2 回以上実行しないことを *affine* 型を用いた型システムで静的に検査
  - 変換が継続の実行回数を変えないことの証明

変換

$$
\begin{array}{rcl}
x & \in & \mathit{Variables} \\
\mathit{eff} & \in & \mathit{Effects} \\
v & ::= & x \mid h \mid \lambda x.e \mid \texttt{perform}\ \mathit{eff}\ v \\
e & ::= & v \mid v\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & \mid & \texttt{inst}\ () \mid \texttt{with}\ v\ \texttt{handle}\ e \\
h & ::= & \texttt{handler}\ \ v\ \ (\texttt{val}\ x \to e)\ \ ((x, k) \to e)
\end{array}
$$

Figure: λ_{eff} の構文

$$
\begin{array}{rcl}
x & \in & \mathit{Variables} \\
K & \in & \{\mathit{Eff}, \mathit{Resend}\} \\
\mathit{eff} & \in & \mathit{Effects} \\
v & ::= & x \mid \lambda x.e \\
e & ::= & v \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{inst}\ () \\
 & \mid & \texttt{match}\ e\ \texttt{with}\ \overrightarrow{\mathit{case}} \\
 & \mid & \texttt{create}\ e \mid \texttt{resume}\ e\ e \mid \texttt{yield}\ e \\
\mathit{case} & ::= & K\ \overrightarrow{x} \to e \mid K\ \overrightarrow{x}\ \texttt{when}\ e = e \to e \\
\mathit{letrec} & ::= & \texttt{let rec}\ f\ \overrightarrow{x} = e\ \mathit{mutrec} \\
\mathit{mutrec} & ::= & \texttt{and}\ \ f\ \overrightarrow{x} = e \mid \texttt{in}\ e
\end{array}
$$

Figure: λ_{ac} の構文

変換

$$
\begin{aligned}
\llbracket x \rrbracket \eta &= \eta\,(x) \\
\llbracket \mathit{eff} \rrbracket \eta &= \mathit{eff} \\
\llbracket \lambda x.e \rrbracket \eta &= \lambda x'.\llbracket e \rrbracket \eta\,[x \mapsto x'] \\
\llbracket \texttt{let}\ x = e\ \texttt{in}\ e' \rrbracket \eta &= \texttt{let}\ x' = \llbracket e \rrbracket \eta\ \texttt{in}\ \llbracket e' \rrbracket \eta\,[x \mapsto x'] \\
\llbracket v_1\ v_2 \rrbracket \eta &= (\llbracket v_1 \rrbracket \eta)\ (\llbracket v_2 \rrbracket \eta) \\
\llbracket \texttt{inst}\ () \rrbracket \eta &= \texttt{inst}\ () \\
\llbracket \texttt{perform}\ \mathit{eff}\ v \rrbracket \eta &= \texttt{yield}\ (\mathit{Eff}\ (\llbracket \mathit{eff} \rrbracket \eta)\ (\llbracket v \rrbracket \eta)) \\
\llbracket \texttt{with}\ h\ \texttt{handle}\ e \rrbracket \eta &= \llbracket h \rrbracket \eta\ (\lambda\_.\llbracket e \rrbracket \eta)
\end{aligned}
$$

$\llbracket \texttt{handler}\ \mathit{eff}\ (\texttt{val}\ x \to e_v)\ ((x,k) \to e_{\mathit{eff}}) \rrbracket \eta =$

```
let rec handler eff vh effh th =
  let co = create th in
  let rec handle r =
    match r with
(1) | Eff eff' v              when eff' = eff  -> effh v continue
(2) | Eff _ _                                  -> yield (Resend r continue)
(3) | Resend (Eff eff' v) k   when eff' = eff  -> effh v (rehandle k)
(4) | Resend effv k                            -> yield (Resend effv (rehandle k))
(5) | _                                        -> vh r
  and continue arg = handle (resume co arg)
  and rehandle k arg = handler eff continue effh (λ_.k arg)
  in continue c // c is anything to run coroutine at first, like nil, (), etc.
in
let eff = ⟦eff⟧η in
let vh = λx'.⟦e_v⟧η[x ↦ x'] in
let effh = λx' k'.⟦e_eff⟧η[x ↦ x', k ↦ k'] in
handler eff vh effh
```

Figure: λ_{eff} から λ_{ac} への変換

関連研究

変換対象	実装	変換の定義
限定継続	o	o
Free モナド	o	o
非対称コルーチン	o	x

- 非対称コルーチンによる実装の利点
  - さまざまな言語で実装可能
    非対称コルーチンを持つ言語は多い、増えている
  - 代数的効果のイディオムをそのまま使える

対称コルーチンと非対称コルーチン

- 対称コルーチン
  - Modula-2, Win32API, Ruby, etc.
  - コントロールを移す先を常に明示
  - 構造化プログラミングと相性が悪い
- 非対称コルーチン
  - 現在“コルーチン”と呼ばれる機能は主にこちら
  - 対称コルーチンをエミュレーション可能
  - 非同期プログラミングやコールバック地獄の解消などでも活躍

ワンショット継続とコルーチンの複製

- **ワンショット継続は軽い**
  いくつかのアプリケーションでの継続の実行は高々１回
  コールスタックなどの複製を抑えることができ、実行効率のアドバンテージ
- **コルーチンの複製操作は重い**
  スタック、レジスタ、ip、etc. を複製
  軽量スレッドとしてのコルーチンとは相性が悪い
  内部では非対称コルーチンを使用している
  Multicore OCaml は継続の複製操作を持つ