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[1,a)]　亀山 幸義[1,b)]

概要：非対称コルーチンは，多くの言語に搭載されている機能である．非対称コルーチンにより，async/await やイテレータなどを実装できるだけでなく，限定継続や call/1cc との等価性も知られており，非常に強力なコントロール抽象である．その一方で，合成性に欠けるという問題点がある．
代数的効果は，エフェクトとハンドラの分離により，合成性とモジュール性の高いプログラミングができる機能である．代数的効果は限定継続や Free モナドなどのコントロール抽象への変換が知られている．
本研究では，ワンショットの代数的効果から非対称コルーチンへの変換を考える．この変換にもとづいて非対称コルーチンを用いて代数的効果を実装することで，より合成性の高いコントロール抽象をもたらすことができる．我々は実際に非対称コルーチンを標準ライブラリに持つ Lua 言語によるワンショットの代数的効果のライブラリを実装した．既存の代数的効果を用いたプログラムのいくつかを本ライブラリを用いて実装，テストをおこない，代数的効果としての正しい動作を確認した．

キーワード：代数的効果, 非対称コルーチン, プログラム変換, 線形性

Satoru Kawahara[1,a)]　Yukiyoshi Kameyama[1,b)]

Abstract: Asymmetric coroutines are the utility that many languages have. Not only async/await or iterator can be implemented with asymmetric coroutines, but also the qualities of the expressive power between asymmetric coroutines and symmetric one, and asymmetric coroutines and cal/1cc is known. On the other hand, asymmetric coroutine has the problem that it is less composability.
Algebraic effects can conduct highly composable and modular programming thanks to the separation between effects and handlers. And it is known that there are conversions from algebraic effects to delimited continuations, and from algebraic effects to free monad.
We define the conversion from one-shot algebraic effects to asymmetric coroutines. Implementing algebraic effects using asymmetric coroutines based on the conversion, we can gain more composable control abstraction. We have implemented algebraic effects library in Lua language, which has asymmetric coroutines as a standard library. We have used the library to implement the existing programs containing algebraic effects, tested them, and confirmed the library satisfies the right behavior of algebraic effects.

Keywords: Algebraic Effects, Asymmetric Coroutines, Program Conversion, Linearity

1. はじめに

非対称コルーチンは Lua や Ruby，C#や Kotlin などの様々な言語に搭載されている機能である．Same-Fringe のような古典的な問題の解法から，今日では async/await のような非同期プログラミングや並行計算，イテレータなどとして幅広く活用されている．それだけでなく，対称コルーチンや call/1cc などとの等価性も知られている非常に強力なコントロール抽象である [1].

コルーチンは 1960 年代から使われてきた伝統的なプログラム言語の機能である．その一方で，de Moura と Ierusalimschy による分類 [1] が行われるまでは理論的な意味付けがなされてこなかった．de Moura と Ierusalimschy による分類では，コントロールを移す操作を 1 つだけ持つ対称コルーチンと，コルーチンを呼び出す操作と中断する操作の 2 つを持つ非対称コルーチンという 2 種類のコルーチンに大別できる．さらに非対称コルーチンは，関数呼び

[1] 筑波大学
University of Tsukuba
[a)] sat@logic.cs.tsukuba.ac.jp
[b)] kam@cs.tsukuba.ac.jp

```lua
local logger = coroutine.yield

local task = function()
  ......
  logger("hello")
  ......
  logger("world")
  ......
end

local log_handler = function(task)
  local logs = {}
  local co = coroutine.create(task)

  while true do
    local resumable, msg =
      coroutine.resume(co)

    if not resumable then
      break
    end

    table.insert(logs, msg)
  end

  return logs
end

log_handler(task)
-- returns {"hello", "world"}
```

図 1　非対称コルーチンによるロガーの実装
Fig. 1　an implementation of a logger with asymmetric coroutines

出しをまたいで中断がおこなえる *stackful* コルーチンと，またぐことのできない *stackless* コルーチンに分けられる．

本研究では，de Moura と Ierusalimschy により定式化された非対称コルーチンを用いる．また，我々の扱うコルーチンは第 1 級オブジェクトとして扱うことができ，かつ stackful なコルーチンなので，彼らの言葉によれば *full* 非対称コルーチンに該当する．あるスレッドから呼び出されたコルーチンは，実行が終了するか中断した場合は，そのスレッドに戻る．そのため，コルーチン（スレッド）間には，“呼び出す”，“呼び出される” という関係が生まれる．

非対称コルーチンを扱うことができる Lua 言語による例を以下に示す（図 1）．コルーチンを作る `create`，コルーチンを実行または再開する `resume` と，コルーチンの実行を中断し，コントロールを呼び出し側に戻す `yield` から成る．

Lua 言語は関数を第一級オブジェクトとして持つ動的型付けの値呼びの言語である [2]．バージョン 5.0 からは，非

```lua
local task = function()
  local ok, file = file_open(path_to_config)

  if not ok then
    coroutine.yield(false, "failed to read config")
  else
    loadconfig(file)
    logger("loaded config")
  end

  ......
end
```

図 2　`task` の拡張
Fig. 2　the extension of `task`

対称コルーチンを扱う `coroutine`ライブラリが追加された [3]．`create`，`resume`，`yield` は `coroutine`モジュールより関数として提供されている．

1 行目では，`logger`を `yield` のエイリアスとして定義している．

`task`関数内で `logger`を複数回呼び出しており，この関数をコルーチンとして呼び出したときに，`resume` にログメッセージが返るようになっている．`log_handler`関数がログメッセージを収集するハンドラとしての役割を担う．Lua 言語の `resume` 関数は，コルーチンが返した値と同時に，コルーチンが終了状態かどうかを表す真偽値を返す．コルーチンが終了状態になるまで `resume` で `co`を繰り返し呼び出すことで，ログメッセージを収集することができる．29 行目で実際にログ収集をおこない，`{"hello", "world"}`というログを収集している．

さらに `task`関数を拡張し，設定ファイルを読み込み，失敗すれば `yield` で呼び出し元のスレッドに通知することを考える（図 2）．この例では失敗の通知として `yield` に `false`を同時に渡している．一方 `log_handler`はこれに対応してないため，修正が必要になる．コルーチンを使って非同期に設定ファイルを読み込む場合や，ファイルの読み込みに失敗したらデフォルト設定を適用したい場合などにも，`log_handler`も拡張していかなければならなくなる．このように，様々な機能をあとから追加していく場合などに，非対称コルーチンの合成性の弱さが問題となる．

代数的効果（Algebraic effects）は Plotkin と Power により提案され，Plotkin と Pretnar によりハンドラが加えられたことで，計算効果を代数的に扱う機構として発展してきた [4] [5]．様々な計算エフェクトを抽象化し，エフェクトの発生をハンドラが捕捉し，そのハンドラが実際のエフェクトの意味を付与する．そして，エフェクトの発生位置以降の処理を継続としてハンドラに渡すことにより，計算を再開することができる．エフェクトと，エフェクトに

```
1 effect Choice : (int * int) -> int
2
3 let f () =
4   let x = 5 in
5   let y = perform (Choice (8, 6)) in
6   x + y
```

図 3 Choice エフェクト
Fig. 3 Choice effect

対して意味を与えるハンドラが分離していることにより，合成性の高くモジュラーなプログラミングが可能である．

代数的効果は，限定継続や選択的 CPS，Free モナドなどのコントロール抽象への変換が知られている [6] [7] [8]．本研究では，代数的効果から非対称コルーチンへの変換を与えることで，合成性の高い代数的効果を非対称コルーチンを持つ言語の上に実装することを考える．まず代数的効果を持つ言語と非対称コルーチンを持つ言語を定義し，前者から後者への変換を考える．代数的効果における継続はコルーチンに対応する．一方でコルーチンの状態をコピーすることはできないため，本研究が対象とする代数的効果は継続がワンショットの体系となる．この制限は，[1] で等価性について論じられている限定継続や call/cc の継続の実行が高々 1 回に制限されていることと本質的に同じである．そして，変換の結果により得られた非対称コルーチンを持つ言語のプログラムに対し，実装を与える．本研究では，非対称コルーチンを扱うライブラリを持つ Lua 言語により実装をおこなった．

本資料の構成は以下のようになる．第 2 章で代数的効果について説明する．非対称コルーチンの抱える合成性の問題が代数的効果によって解決できることを，上記のプログラムを代数的効果で実装することで示す．第 4 章で本研究の変換について述べる．代数的効果を持つ言語 λ_{eff} と非対称コルーチンを持つ言語 λ_{ac} を定義し，λ_{eff} から λ_{ac} への変換を定義する．第 5 章では，変換より得られた結果に対して与えた Lua 言語による実装と，変換の対応について述べる．

2. 代数的効果

代数的効果はエフェクトの定義，エフェクトの発生とエフェクトのハンドラから成る．まずエフェクトを定義し，エフェクトを含む式とそのハンドラを記述する．代数的効果をプリミティブに持つ，ML 風の構文の言語 Eff [9] によるプログラムの例は以下のようになる （図 3）．1 行目では，Choiceというエフェクトを定義している．intのタプルを受け取り，intを返すという型情報以外は，エフェクトの処理内容などの意味を持っていない．3 行目で定義される関数 fは Choiceエフェクトを 2 回実行している．エフェクトの実行には，performというキーワードを使う．次に，エフェクトを捕捉し，意味を付与するハンドラを実装する （図 4）．minハンドラは Choiceエフェクトを捕捉すると，Choiceエフェクトの引数を x，yに，ハンドラによって区切られる限定継続を kに束縛する．そして x，yのうち小さい方を継続 kに渡し，コントロールをエフェクトの発生した位置に戻す．5 行目の| x -> print_int xは値ハンドラ（value handler）と呼ばれ，ハンドルされた式が最終的に返す値を受け取り，ハンドリング全体の戻り値を返す．minでは，xに値を束縛し，printに渡して最終的な値を印字する．

```
1  let min = handler
2    | effect (Choice (x, y)) k ->
3      if x < y then k x
4      else k y
5    | x -> print x
6
7  with min handle f () (* prints 11 *)
8
9  let ave = handler
10   | effect (Choice (x, y)) k ->
11     k ((x + y) / 2)
12   | x -> print x
13
14 with ave handle f () (* prints 12 *)
```

図 4 Choice エフェクトのハンドラの定義
Fig. 4 the definition of the handler for Choice effect

7 行目では minハンドラでハンドルしながら関数 fを実行している．fのエフェクトを実行すると，minハンドラの取る継続 kには，with min handle let y = □ in x +↩
↪ yが束縛される．□ には，kの引数が入ってくる．また，再び minハンドラによってハンドルされるため，Choiceエフェクトが発生する場合や，値が返ってくる場合は引き続き minハンドラによってハンドルされる．したがってこの場合は with min handle let y = 6 in x + yが実行され，値ハンドラに渡ることで，11が印字される．このように継続がハンドルされ続けるハンドリング処理は *deep* ハンドラと呼ばれる [10]．本研究の対象とする代数的効果では，deep ハンドラを採用する．

aveハンドラは，Choiceエフェクトの 2 つの引数の平均値を継続に渡している．14 行目では aveハンドラで関数 fをハンドルしながら実行している．同様の動作で，12が印字される．

このように，同様のエフェクトでも，ハンドルによって異なる振る舞いをさせることができる．

また，複数のエフェクトを定義，ハンドルすることも可能である．新たに Flipと Exitというエフェクトを定義し

```
1  effect Flip : unit -> bool
2  effect Exit : unit -> int
3
4  let g () =
5    if perform (Flip ()) then
6      let x = perform (Choice (3, 10)) in
7      x + 5
8    else
9      perform (Exit ())
10
11 let h = handler
12   | effect (Flip ()) k -> k false
13   | effect (Exit ()) _ -> ()
14   | x -> print x
15
16 with h handle
17   with min handle g ()
```

図 5　複数のエフェクト
Fig. 5　multiple effects

た．関数 gでは，Flipエフェクトの戻り値によって分岐し，Chooseまたは Exitエフェクトのいずれかが発生する．

hハンドラでは 2 つのエフェクトをハンドルしている．Flipエフェクトには，常に falseを返す．Exitエフェクトは継続を破棄することで，エフェクトの発生により大域脱出をおこなうことができる．

16 行目では式のハンドリングが入れ子になっている．内側では minハンドラが gの呼び出しをハンドルする．外側で hハンドラによってハンドルされており，Flipや Exitは minハンドラを突き抜けて hハンドラに捕捉される．

3. 合成性

第 1 章で述べた非対称コルーチンの合成性に関する問題を，実際に代数的効果で実装することで，解消できることを示す （図 6）．まず Loggerエフェクトと LoadErrorエフェクトを定義する．Loggerエフェクトの発生は，図 1 の logger関数の呼び出しに対応する．設定ファイルの読み込み時のエラーは LoadErrorエフェクトの発生で表現している．log_handlerは Loggerエフェクトを捕捉し，リストの参照セルに msgを追加し，値ハンドラで収集したログのリストを返している．load_error_handlerで LoadError↩
↪ エフェクトを捕捉する．標準エラー出力にエラーメッセージを出力し，継続を破棄することで taskの実行を停止する．

代数的効果を用いて，Loggerエフェクトと LoadErrorエフェクトという 2 つのエフェクトを定義することで，それぞれを別のエフェクトとして独立してハンドリングをおこなうことができ，合成性の高いプログラムになる．

```
1  effect Logger : string -> unit
2  effect LoadError : string -> unit
3
4  let task () =
5    let ok, file = file_open path_to_config in
6    let () =
7      if not ok then
8        perform (LoadError "failed to read config")
9      else
10       let () = loadconfig file in
11       perform (Logger "loaded config")
12   in
13   ......
14
15 let log_handler task =
16   let logs = ref [] in
17   let h = handler
18     | effect (Logger msg) k ->
19       ref := msg :: !ref;
20       k ()
21     | x -> !logs
22   in
23   with h handle task ()
24
25 let load_error_handler = handler
26   | effect (LoadError msg) _ ->
27     print_stderr msg
28   | x -> x
29
30 with load_error_handler handle
31   log_handler task
```

図 6　代数的効果による task の実装
Fig. 6　the reimplementation of task with algebraic effects

3.1 ワンショット代数的効果

本研究の対象とする代数的効果は，ハンドラで扱える継続の実行を高々 1 回に制限している．我々の変換では，代数的効果における継続が非対称コルーチンにおけるコルーチンスレッドに対応する．一方，de Moura らにより定式化された非対称コルーチンを持つ計算体系には複製の操作が無い．また，非対称コルーチンを持つプログラム言語のほとんどは，特定の状態のコルーチンを複製する操作を持たない．このため，特定の状態のコルーチンの実行，つまり継続の実行は高々 1 回に制限される．

この制限により，本研究の実装で記述できるプログラムは，制限のない代数的効果を用いる場合と比較して範囲の狭いものとなる．しかし，継続の実行が高々 1 回に制限された場合でも，依然として有用であることが知られている [11]．

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

図 7 λ_{eff} の構文
Fig. 7 the syntax of λ_{eff}

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

図 8 λ_{ac} の構文
Fig. 8 the syntax of λ_{ac}

4. λ_{eff} から λ_{ac} への変換

本章では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について論じる．

4.1 λ_{eff}

変換のソースとなる，代数的効果を持つ言語 λ_{eff} を定義する （図 7）．λ_{eff} は，ラムダ計算に `let` 式に加え，代数的効果に関する構文を持つ．エフェクトの生成 `inst()`，エフェクトの発生 `perform` eff v，ハンドラ `handler` v `(val` $x \to e$`)` $((x,k) \to e)$ とハンドル処理 `with` e `handle` e を持つ．実際に広く使われている代数的効果と比較して，簡単のために 1 つのハンドラにより 1 つのエフェクトだけハンドルできるように制限されている．しかし，[6] で用いられる代数的効果を持つ計算体系も同様に 1 ハンドラ 1 エフェクトの制限を持つが，この制限は本質ではなく，1 つのハンドラで複数のエフェクトをハンドルできるように自然に拡張できることが述べられている．

4.2 λ_{ac}

変換の対象となる，非対称コルーチンを持つ言語 λ_{ac} を定義する （図 8）．λ_{ac} はラムダ計算に `let` 式と非対称コルーチンの操作を持つ．また λ_{eff} のエフェクト定義の操作を互換するための `inst ()` を持つ．そして変換のため必要となる代数的データ型とパターンマッチング，および相互再帰を持つ体系となっている．

表 1 ワンショットの代数的効果と非対称コルーチンの直感的な対応
Table 1 the intuitive correspondence between oneshot algebraic effects and asymmetric coroutines

代数的効果	↦	非対称コルーチン
エフェクトの発生	↦	yield
エフェクトのハンドル	↦	create & resume
継続	↦	コルーチン

4.3 λ_{eff} から λ_{ac} への変換

4.3.1 直感的な対応

変換の直感的な対応は**表 1** のようになる．

エフェクトの発生によりコントロールがハンドラに移ることを考えると，`yield` が対応することが考えられる．したがって，ハンドラは `yield` の戻り先である `resume` となり，ハンドルしたい式を `create` でコルーチンにする必要がある．これらの対応から，継続はコルーチンに自然に対応する．

4.3.2 実際の変換

表 1 を参考に，実際の変換は**図 9** のようになる．η は変数のストアである．λ_{eff} の変数は，対応する λ_{ac} の変数に変換される．変数，エフェクトおよびその定義，ラムダ抽象，`let` 式，関数適用は変数名の付け替えをおこなうだけである．`perform` は直感の通り `yield` を用いる．また後述するハンドラでの処理のため，*Eff* というタグを付けている．直感的な対応ではハンドルされる式をコルーチンにする必要がある．そのため式のハンドリング `with` h`handle` e は式の代わりにサンクを受け取るように変更する．また，ハンドラは関数に変換されるため，ハンドリングは関数適用に変換される．

ハンドラの変換は複雑である．ハンドラは 4 引数の関数 *handler* になる．λ_{eff} の項のエフェクト *eff* は λ_{eff} にそのまま対応する．*vh* は値ハンドラを λ_{ac} の関数に変換したものである．*effh* も同様にエフェクトハンドラを λ_{ac} の関数に変換する．*th* はハンドルされる式を λ_{ac} にサンクとして渡されるものである．まず，*th* からコルーチン *co* を生成する．*handle* がハンドルされる式から戻ってきたもの（エフェクトの発生または最終的な値）を処理している．r をパターンマッチで分解し，それぞれの場合について処理を分岐する．(1) ハンドルすべきエフェクトが来た場合，エフェクトハンドラ *effh* に v と *continue* を渡す．*continue* は *co* に *arg* を渡して再開し，戻り値を *handle* に渡す．コルーチンスレッドが継続に対応するため，*co* を `resume` することで継続の実行に当たる．また *handle* に戻り値が渡ることで，deep ハンドラの動作を模倣している．

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

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

```
  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
```

図 9 λ_{eff} から λ_{ac} への変換
Fig. 9 the conversion from λ_{eff} terms to λ_{ac} ones

(2) ハンドルできないエフェクトが来た場合，引数 r と継続を *Resend* でタグ付けし，`yield` に渡す．これによって 1 つ外側のハンドラにエフェクトの処理を任せ，継続を実行させることができる．(3) 他のハンドラからハンドル可能なエフェクトが飛んできた場合，エフェクトの引数 v と，継続として *rehandle* k を渡す．*rehandle* は新しくハンドラを作り，引数 k に *arg* を渡して実行するのをハンドルしている．ハンドラを作り直し，値ハンドラに相当する関数に *conitinue* を渡すことで，deep ハンドラの動作に合わせ，ハンドルの終了後に現在の継続にコントロールを戻すことができる．(4) 他のハンドラからハンドルできないエフェクトが飛んできた場合，`yield` でハンドルできないエフェクトを再送する．このとき *Resend* の持ってきた継続をコルーチンに包み *continue* に渡すことで，今のハンドラによる処理を持続させることができる．(5)*Eff*，*Resend* のタグが付いてない値がきた場合，値ハンドラに値を渡したものを返してハンドリングを終了する．最後に *continue* を実行することで，ハンドルされる式が評価される．*co* はサンクから作られているため，最初に渡される引数は使用されない．そのため最初 *continue* に渡す値は *nil* や *()* などの適当な値でよい．

```
handler(Choose,
  function(v) return v end,
  function(k, x, y)
    return k((x + y) / 2)
  end)
```

図 10　複数の引数を取るハンドラ
Fig. 10　handler with multiple arguments

```
handlers({
  function(v) return v end,
  [Flip] = function(k)
    return k(false)
  end,
  [Exit] = function()
    return
  end
})
```

図 11　複数のエフェクトハンドリング
Fig. 11　single handler for multiple effects

5. 実装

第 4 章で示した変換により得られた結果に対し，Lua 言語による実装を与えた．本章では，変換の結果と実際のプログラム言語のライブラリとしての間の変更を交えて議論する．本実装は GitHub にて MIT ライセンスの元で公開されている [12].

5.1 実装におけるポリフィル

Lua は代数的データ型を持たないため，tableという連想配列や配列として振る舞うデータ構造を代わりに用いる．代数的データ型の代用となる tableにタグの種類を示す clsというフィールドを追加し，フィールドの値で処理を分けることでパターンマッチをおこなう．また，エフェクトを定義する inst() は，一意な値を生成するように，inst() を呼び出すごとに内部でオブジェクトを生成，破棄し，そのオブジェクトの id を用いた．

5.2 実装における拡張

エフェクトハンドラの引数の位置を，エフェクトの引数と継続を入れ替え，多値と tableを展開することにより，複数の値をエフェクトハンドラへ渡せるようにした（図 **10**）．

また，複数のエフェクトを一つのハンドラでハンドルする拡張もおこなった （図 **11**）．複数のエフェクトハンドラを，エフェクトと関数の連想配列として管理することで実装している．

本ライブラリを用いて図 6 を実装すると，図 **12** ようになる．Eff 言語に近い構文で代数的効果を記述できることがわかる．

6. おわりに

本資料では，継続の実行をワンショットに制限した代数的効果から非対称コルーチンへの変換を示した．これにより，非対称コルーチンを用いて代数的効果を実装することができるようになり，合成性の高いコントロール抽象を多くの言語で使うことができるようになる．実際に，変換の結果に基づき，Lua 言語の coroutineモジュールを用いて実装をおこない，テストをおこなうことで，変換の確から

```
local Logger = inst()
local LoadError = inst()

local task = function()
  local ok, file = file_open(path_to_config)

  if not ok then
    perform(LoadError("failed to read config"))
  else
    loadconfig(file)
    perform(Logger("loaded config"))
  end
  ......
end

local log_handler = function(task)
  local logs = {}
  local h = handlers({
    function(x) return x end,
    [Logger] = function(k, msg)
      table.insert(logs, msg)
      return k()
    end
  })

  return h(task)
end

local load_error_handler = handlers({
  function(x) return x end,
  [LoadError] = function(k, msg)
    return
  end
})

load_error_handler(log_handler(task))
```

図 12　本ライブラリによる図 6 の実装
Fig. 12　the reimplementation of 図 6 with the library

しさを確認した．本実装に倣うことで，非対称コルーチンを扱えるプログラム言語によりワンショット代数的効果を実装することができる．

6.1 *stackfulness*

本研究で対象とした非対称コルーチンは関数呼び出しをまたいで中断することができる *stackful* コルーチンである．その一方で，関数呼び出しをまたぐことのできない *stackless* コルーチンを持つ JavaScript 言語による代数的効果の実装もおこなわれている [13]．このライブラリは，本研究で実装した [13] に基づいて実装されている．関数もコルーチンにし，関数呼び出しを `yield` でおこなうことでエフェクトによるコントロールの移動を実現している．この方法を用いることで，stackless コルーチンを持つ言語にも本研究の成果が適用できるようになることが期待できる．

6.2 今後の課題

本研究では，ワンショットの代数的効果から非対称コルーチンへの変換をおこなう．この変換が満たすべき性質の一つに，継続の実行が高々 1 回であることを保持していることが挙げられる．まず，λ_{eff} において継続の実行が高々 1 回に制限されていることを静的に保証することを考える．現在の取り組みとして，affine types を用いた型システムを λ_{eff} に与えることで，継続が 2 回以上実行される場合を静的に検出することを検討している．そして，型のついた λ_{eff} の項を λ_{ac} に変換する過程で，継続の実行回数が増減しないことの証明をおこなう．

参考文献

[1] Moura, A. L. D. and Ierusalimschy, R.: Revisiting coroutines, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 31, No. 2, p. 6 (2009).

[2] Ierusalimschy, R., de Figueiredo, L. H. and Celes, W.: The evolution of Lua, *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ACM, pp. 2–1 (2007).

[3] De Moura, A. L., Rodriguez, N. and Ierusalimschy, R.: Coroutines in lua, *Journal of Universal Computer Science*, Vol. 10, No. 7, pp. 910–925 (2004).

[4] Plotkin, G. D. and Pretnar, M.: Handling Algebraic Effects, *Logical Methods in Computer Science*, Vol. 9, No. 4:23, pp. 1–36 (2013).

[5] Plotkin, G. and Power, J.: Algebraic operations and generic effects, *Applied Categorical Structures*, Vol. 11, No. 1, pp. 69–94 (2003).

[6] Kiselyov, O. and Sivaramakrishnan, K.: Eff directly in OCaml, *arXiv preprint arXiv:1812.11664* (2018).

[7] Leijen, D.: Algebraic Effects for Functional Programming, Technical report, Technical Report. 15 pages. `https://www.microsoft.com/en-us/research/publication/algebraic-effects-for-functional-programming/` (2016).

[8] Pretnar, M., Saleh, A. H., Faes, A. and Schrijvers, T.: Efficient compilation of algebraic effects and handlers, *CW Reports, volume CW708*, Vol. 32 (2017).

[9] Bauer, A. and Pretnar, M.: Programming with algebraic effects and handlers, *Journal of Logical and Algebraic Methods in Programming*, Vol. 84, No. 1, pp. 108–123 (2015).

[10] Kammar, O., Lindley, S. and Oury, N.: Handlers in action, *ACM SIGPLAN Notices*, Vol. 48, No. 9, ACM, pp. 145–158 (2013).

[11] Dolan, S., Eliopoulos, S., Hillerström, D., Madhavapeddy, A., Sivaramakrishnan, K. and White, L.: Concurrent system programming with effect handlers, *International Symposium on Trends in Functional Programming*, Springer, pp. 98–117 (2017).

[12] Kawahara, S.: eff.lua, `https://github.com/Nymphium/eff.lua` (2018). Accessed 2019, 13, March.

[13] MakeNowJust: eff.js, `https://github.com/MakeNowJust/eff.js` (2019). Accessed 2019, 29, May.