筑波大学大学院博士課程

システム情報工学研究科修士論文

コルーチンを用いた代数的効果の新しい実装方法の提案

河原 悟

修士(工学)

(コンピュータサイエンス専攻)

指導教員　亀山 幸義

2020年3月

概要

代数的効果およびハンドラは計算エフェクトの新たな言語機能であり、研究者のみならずプログラマからも注目を集めはじめている。代数的効果をライブラリとして実装するために、スタック操作、限定継続、 Free モナドを利用する方法がある。しかし、スタック操作は処理系に対する深い知識が必要になり、また、限定継続を操作できる言語は限られているため、多くの言語に適用できる方法ではない。本研究では新たに、ワンショットの代数的効果およびハンドラを、コルーチンを用いて直接埋め込む方法を提案する。多くの言語が持つコルーチンとワンショットの継続を対応付けることで、汎用性の高い埋め込みを実現できる。この埋め込みに基づき、 Lua 言語と Ruby 言語上にワンショット代数的効果のライブラリを実装した。マイクロベンチマークを実施し、 Free モナドによる代数的効果の埋め込み方法と比較して、本研究の埋め込みに基づいた実装がパフォーマンスの側面でも優れていることを示した。

目次

図目次

第 1 章

はじめに

代数的効果 [21] およびハンドラ [22] は計算エフェクトを定義、操作するための新たな言語機能であり、現在活発な研究分野であるのみならず、関心の分離をおこなうための強力な機能としてプログラマからも注目を集めはじめている。代数的効果およびハンドラは Eff [2]、 Multicore OCaml [8]、 Koka [18] などのプログラム言語の組み込み機能や、Haskell、 OCaml、 Scala や JVM バイトコード、 C などのライブラリとして提供されている。

代数的効果およびハンドラは主に、スタック操作、限定継続、および Free モナドによって実装されている。しかし、これらを用いてライブラリを実装する場合には、次に述べる欠点が存在する。スタック操作を利用した実装は、上記で述べた JVM バイトコードと C 言語上の実装で利用されているが [5] [19]、この実装方法はランタイムに強く依存し、言語処理系に関する深い知識を要求する。そのため実装に大きなコストがかかる上、汎用性が低い。限定継続 [7] を利用した実装は上に挙げた OCaml と Scala の実装に利用されている [4] [17]。限定継続の操作は低級な機能の知識を必要としないため、システマティックかつ簡潔に代数的効果が実装できる。その反面、限定継続を操作できる言語は非常に少ない。また、代数的効果を埋め込むためには複数のプロンプトを扱える限定継続操作が必要になるが [12] [17]、そのような限定継続の操作の埋め込み自体は非自明である。したがって、限定継続を持つ、あるいは限定継続を実装できる数少ない言語の上にしか実装することができない。 Free モナドによる実装は Haskell と Scala による実装で利用されており [1] [15] [16] [25]、計算エフェクトを実装するシステマティックな方法の一つとして知られている [24]。 Free モナドのデメリットは、モナディックプログラミングをプログラマに強制する点と、代数的効果と他の副作用を持つ計算を併せて利用する場合、モナド変換子や `forM`、 `replicateM`などの専用の演算子が必要になる点である。

本論文では、汎用性が高く効率の良い、簡潔な代数的効果の埋め込み方法を提案する。我々の埋め込み方法の鍵となるアイデアは、多くの言語が組み込みの機能として持つコルーチンを利用することである。継続を複数回実行することができる一般的な限定継続と比較し

て、コルーチンは表現力が低く、 (暗黙的な) 限定継続を 2 回以上呼び出すことができない。言い換えると、コルーチンは**ワンショット**の限定継続演算子、つまり継続の実行が高々 1 回に制限された限定継続演算子と同等の表現力を持つ [20]。ワンショットの限定継続の処理は継続のコピーが不要なため、複数回継続を実行できる一般的な限定継続と比較して効率的な実装を与えられることが知られている [6]。同様に、我々の対象とする代数的効果およびハンドラの継続を実行できる回数は高々 1 回となる。そして、継続をコルーチンスレッドの残りとして取り出すことができるので、 Free モナドを用いた実装と異なり、モナディックプログラミングを支援する機能のない言語でも、プログラマはエフェクトの発生位置からの継続を明示的に書く必要がなくなる。

我々は非対称コルーチンを用いて代数的効果およびハンドラを実装した。実装は実用性を指向し、本論文に基づき Lua 言語と Ruby 言語のライブラリとして実装し、 GitHub に公開している[*1][*2]。すでに利用されており、さらに興味を持ったユーザによって他の言語へ移植された[*3][*4]。コルーチンを用いた代数的効果の実装は本論文以前に存在するが[*5][*6][*7]、これらは第 1 級の継続をユーザが利用できない点や、実装や利用方法が複雑という点において問題がある。我々の提供するライブラリはいずれも、ユーザが継続を利用でき、シンプルで誰でも理解できる実装になっている。

本論文の主な貢献は次の 2 つである。

- 本論文では代数的効果およびハンドラの新しい埋め込み方法を提案する。埋め込みに利用するものは標準的な非対称コルーチンの機能のみであり、他の特別な機能は不要である。また、我々の埋め込みは継続渡し方式やユーザレベルのコントロール演算子を必要としない。そのため、 Free モナドによる代数的効果の埋め込みと比較して、多くのケースで高いパフォーマンスを発揮することができる。
- 我々の埋め込み方法は、代数的効果からコルーチンへの局所的、合成的な変換として簡潔に定義されている。そして、変数管理をホスト言語に任せることで、この変換に基づき、コルーチンを持つ Lua 言語と Ruby 言語上にライブラリとして実装することができた。実用性を指向しているため、実装独自の拡張がいくつかほどこされている。これらはすでに GitHub で公開されており、誰でも利用することができる。そして、このライブラリを参考にすることで、より多くの言語で代数的効果を実装することができる。

[*1] `https://github.com/Nymphium/eff.lua`
[*2] `https://github.com/Nymphium/ruff`
[*3] `https://github.com/MakeNowJust/eff.js`
[*4] `https://github.com/pandaman64/effective-rust`
[*5] `https://github.com/dry-rb/dry-effects`
[*6] `https://github.com/digital-fabric/affect`
[*7] `https://github.com/briancavalier/forgefx`

本論文の構成は次のようになる。第 2 章でワンショット代数的効果およびハンドラの例を、 Lua 言語上に実装した代数的効果ライブラリを用いて説明する。第 3 章ではワンショット代数的効果の埋め込み方法を示す。まず 2 つの言語、代数的効果およびハンドラを持つ λ_{eff} と非対称コルーチンを持つ言語 λ_{ac} を定義し、前者から後者への変換を定義する。第 4 章では、第 3 章で述べた変換がマクロ表現可能であることを示す。第 5 章は Lua 言語と Ruby 言語による実装で拡張された部分や実用面での問題について議論する。第 6 章では Lua 言語上に実装したライブラリを用いて、 Free モナドによる埋め込みとのパフォーマンスを比較する。第 7 章では関連研究について述べ、第 8 章では結論を述べる。

第 2 章

ワンショット代数的効果の利用例

本章では代数的効果およびハンドラについて、 3 つの例を用いて説明する。代数的効果を用いたプログラムの例の記述には、 Lua 言語および我々が Lua 上に実装したライブラリを用いる。このライブラリは第 3 章で述べる埋め込みに基づいており、実装に関する議論は第 5 章で後述する。さらに、第 6 章で述べるように、ライブラリを用いたこれらのプログラムは高いパフォーマンスで実行することができる。以下に示す例では、代数的効果およびハンドラを説明するために、ライブラリの提供する 3 つの関数 `inst`、 `perform`と `handler`を利用する。

2.1 例外

次に示す例のように、代数的効果は復帰可能な例外と考えることができる。 `inst`関数は新しいエフェクトを生成する。

```
local DivideByZero = inst()
```

そして `perform`を呼ぶことでエフェクトを発生させる。

```
local div = function(x, y)
  if y == 0 then
    return perform(DivideByZero, nil)
  else
    return x / y
  end
end
```

Lua 言語では、 `local`により変数を定義する[*1]。そして、 `function`ブロックで関数を定義

[*1] `local`修飾子を付けない場合、変数への代入またはグローバル変数の定義となる。

する。 divは xと yの商を得る。このとき yが 0の場合は商を計算せず、 DivideByZeroエフェクトを発生させる。エフェクトが発生すると、例外ハンドラ同様に、制御が最も近いハンドラに移る。

handler関数で新たにハンドラを生成する。

```
local with_nil = handler {
  val = function(_) return nil end,
  [DivideByZero] = function(_, _)
    return nil
  end
}
```

ハンドラはエフェクトを捕捉すると、その引数と、エフェクトの発生位置からハンドラで囲まれた部分までの現在の (限定) 継続を取得する。上記のコードの 2 行目の valは値ハンドラ (value handler) と呼ばれ、ハンドルされている式が値を返すときに利用される。[DivideByZero] =から続く DivideByZeroのハンドラ関数は 2 つの引数を持ち、 1 つ目は発生したエフェクトの引数、 2 つ目は限定継続にそれぞれ束縛される。この例では、エフェクトのハンドラの引数は無視され、計算全体は nilを返す。したがって、上記の例は単純な例外の実装と考えることができる。

次に、**復帰可能な例外**を示す。 DivideByZeroエフェクトとそこからの継続を捕捉し、“例外を発生した” 位置からの計算を再開させる。

```
local with_default_zero = handler {
  val = function(v) return v + 1 end,
  [DivideByZero] = function(_, k)
    return k(0)
  end
}
```

実装自体は単純であり、継続に 0を渡すだけである。値ハンドラは、受け取った値に 1を足す。 with_default_zeroは次のように利用する。

```
with_default_zero(function()
  local v = div(3, 0)
  return v + 20
end)
```

このコードでは、 3と 0が divに渡り、 DivideByZeroエフェクトが発生する。 with_default_zeroがこのエフェクトを捕捉し、ハンドラが限定継続を kに束縛す

る。このとき、 kが受け取る値は divの戻り値に相当する。したがって、ハンドラが kに 0を渡すと、 divは 0を返し、次のような計算に移る。

```
with_default_zero(function()
  local v = 0
  return v + 20
end
```

これより、ハンドルされている式は値として 20を返す。そして、値ハンドラ function(x) return x + 1 endが呼び出され、 20を受け取り 21を返す。

2.2 状態

パラメータ渡しのハンドラを用いて、参照セルを使わずに状態を実装することができる。まず、 2 つのエフェクトを定義する。

```
local Get = inst()
local Put = inst()
```

状態のある計算を実行するために、初期状態と計算をサンクとして受け取る run関数を定義する。

```
local run = function(init, task)
  local step = handler {
    val = function(_) return function() end end,
    [Get] = function(_, k)
      return function(s)
        return k(s)(s)
      end
    end,
    [Put] = function(s, k)
      return function(_)
        return k()(s)
      end
    end
  }

  return step(task)(init)
end
```

stepは Getと Putを捕捉するハンドラである。パラメータ渡しの手法に則るために、値ハンドラ自体は何も返さない関数を返す。ハンドラが Getを捕捉したとき、状態を受け取る関数 function(s) return k(s)(s) endを返す。ハンドラが Putを捕捉してその実引数 vを受け取ったとき、 vを変数 sに束縛し、それを新しい状態として継続 kに渡すサンクを返す。ハンドラを定義したのち、上記のコードは状態の初期値として initを与え、計算を実行する。

2.3 Defer

Go にある defer[*2]のような実行を遅延する関数の機能を、代数的効果を用いて実装することができる。 Go では defer文に関数呼出しを書くことができ、その親の処理が終わるまで関数呼出しの実行は遅延される[*3]。チュートリアル[*4]より引用すると、以下のように書くことができる。

```
package main

import "fmt"

func main() {
  defer fmt.Println("world")

  fmt.Println("hello")
}
```

main関数内で、関数呼出し fmt.Println("world")を登録する。そして mainを呼び出し helloが印字されてから、 deferに登録された関数が実行され worldが印字される。このような機構の一部を、我々のライブラリを用いて実装することができる。 Go は defer された関数呼出しを、例外の発生にかかわらず実行することができるが、ここでは省略する。 Defer エフェクトを定義し、それに対応するハンドラを定義する。

```
local Defer = inst()

local with_defer = handler {
  val = function(_) end,
  [Defer] = function(proc, k)
    k()
```

[*2] https://golang.org/ref/spec#Defer_statements

[*3] 構文の定義によればあらゆる式が書けるが、 Go のデファクトスタンダードのコンパイラである go は関数呼出し以外の式が書かれた defer文はコンパイルエラーとする。

[*4] https://tour.golang.org/flowcontrol/12

```
    proc()
    return nil
  end
}
```

with_deferは Deferエフェクトを捕捉し、その引数を受け取り procに束縛する。 Lua 言語は値呼びの評価戦略を採用しており、遅延評価に関する機能を持たないため、 defer される関数呼出しはサンクとして受け取る。 Deferエフェクトをハンドラが捕捉すると、まず継続を起動する。そして関数 procを呼び出し、継続の返す値を破棄して nilを返す。 with_deferは次のように使うことができる。

```
with_defer(function()
  perform(Defer, function() print("world") end)

  print("hello")
end)
```

このコードは helloと worldを印字してから nilを返す。 Go の deferのすべての機能を実装するためには、継続を例外ハンドラで囲み、 defer された関数呼出しを実行したのちに捕捉した例外を再送する必要がある。

これまでに挙げた例は、継続の実行回数に制限のない Eff 言語でも書くことができる。また、これまでのプログラムで示してきたように、我々のライブラリの提供する代数的効果は合成的であり、モジュラーなプログラミングや関心の分離ができるという、代数的効果およびハンドラの特徴をそのまま享受できる。

第 3 章

コルーチンを用いた代数的効果の埋め込み

この章では、非対称コルーチンを用いたワンショット代数的効果を実装するための、我々の埋め込み方法について説明する。この目的に沿って、型なしラムダ計算を拡張した 2 つの言語、λ_{eff} と λ_{ac} を定義する。λ_{eff} はワンショット代数的効果を持つ言語で、λ_{ac} は非対称コルーチンを持つ言語である。そして、λ_{eff} から λ_{ac} へのプログラム変換を定義する。

3.1 λ_{eff}

λ_{eff} は Effy [23] に基づいて設計された、代数的効果およびハンドラを持つ、型なしの言語である。

3.1.1 構文

λ_{eff} の構文を図 3.1に示す。通常の `let` 式は、$\texttt{let}\, x = c_1\, \texttt{in}\, c_2$ で表される。`perform` *eff* v は引数に v が渡されたエフェクト *eff* を発生する式である。

`handler` *eff* $(\texttt{val}\, x \to e_1)\,((y, k) \to e_2)$ はハンドラを生成する式である。生成されるハンドラはエフェクト *eff* を捕捉し、エフェクト発生時に渡される値を y に、限定継続を k に束縛して e_2 を評価する。$\texttt{val}\, x \to e_1$ は値ハンドラを表し、ハンドルしている式が返す値を処理する。

Effy は 1 つのハンドラにつき複数のエフェクトを捕捉できるが、λ_{eff} は、簡単のため、1 つのハンドラにつき 1 つのエフェクトまでしか捕捉できない。この簡易化は λ_{eff} の利便性を損なうが、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拡張するのは容易である。実際に、第 5 章で述べるように、本論文に基づいて実装されたライブラリは複数のエフェクトをハンドルできるように拡張されている。

`with` h `handle` e はハンドル式であり、ハンドラとして h を設定し、設定したハンドラの

$$
\begin{array}{rcl}
x & \in & \textit{Variables} \\
\textit{eff} & \in & \textit{Effects} \\
v & ::= & x \mid h \mid \lambda x.\, e \\
e & ::= & v \mid v\ v \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & \mid & \texttt{perform}\ \textit{eff}\ v \mid \texttt{with}\ v\ \texttt{handle}\ e \\
h & ::= & \texttt{handler}\ \textit{eff}\ (\texttt{val}\ x \to e)\ ((x, x) \to e) \\
 & & \\
w & ::= & \texttt{clos}\,(\lambda x.e, E) \mid \texttt{closh}\,(h, E) \\
F & ::= & (\square\ e, E) \mid w\ \square \\
 & \mid & (\texttt{let}\ x = \square\ \texttt{in}\ e, E) \\
 & \mid & (\texttt{with}\ w\ \texttt{handle}\ \square)^{\textit{eff}} \\
 & \mid & (\texttt{with}\ \square\ \texttt{handle}\ e, E) \\
C & ::= & e \mid w \\
E & ::= & [] \mid (x = w) :: E \\
K & ::= & [] \mid F :: K
\end{array}
$$

図 3.1　$\lambda_{\textit{eff}}$ の構文およびランタイム表現

もとで e を評価する。

構文の分類 w は、 $\texttt{clos}\,(\lambda x.e, E)$ と $\texttt{closh}\,(h, E)$ から成るランタイムの値を表す。 $\texttt{clos}\,(\lambda x.e, E)$ は環境として E を持つ関数クロージャ、 $\texttt{closh}\,(h, E)$ は環境 E を持つハンドラクロージャである。

フレーム F は、関数適用、 `let` 束縛またはハンドル式から成る。 C は CEK マシンにおける “コード” を表す。 K は継続に相当し、フレームにより構成されるスタックである。

3.1.2　意味論

補助関数

意味論のための補助関数を図 3.2に示す。 K // *eff* は、エフェクト *eff* をハンドルするフレーム F、 $K = K_1 :: [F] :: K_2$ を満たす K_1 と K_2 から成る三組 (K_1, F, K_2) を返す。複数のフレームが *eff* をハンドルしている場合、最初の 1 つが選択される。いずれのフレームもエフェクトをハンドルしない場合の動作は未定義である。 $(\!|K|\!)$ はスタック K を関数形式の継続に変換する。スタックが空でない場合、先頭のフレーム F に値を渡し、その結果をスタックの残り K が作る関数 $(\!|K|\!)$ に渡す関数を返す。スタックが空の場合、恒等関数を返す。 $K * E$ は、 $(\!|-|\!)$ を利用してスタック K と環境 E からクロージャを作る補助関数である。

$$\left((\texttt{with}\ w\ \texttt{handle}\ \square)^{\mathit{eff}} :: K\right) \mathbin{//} \mathit{eff} = \left([]\,, (\texttt{with}\ w\ \texttt{handle}\ \square)^{\mathit{eff}}\,, K\right)$$

$$(F :: K) \mathbin{//} \mathit{eff} = (F :: K', F', K'')$$
$$\text{where } F \neq (\texttt{with}\ w\ \texttt{handle}\ \square)^{\mathit{eff}}$$
$$\text{and } (K', F', K'') = K \mathbin{//} \mathit{eff}$$

$$(\!|F :: K|\!) = \lambda x.\ (\!|K|\!)\ F\ [x]$$
$$(\!|\ []\ |\!) = \lambda x.\ x$$
$$K * E = \texttt{clos}\,(\lambda x.(\!|K|\!)\ x, E)$$

図 3.2　λ_{eff} の意味論の補助関数

小ステップ意味論

CEK マシン方式の値呼びの小ステップ意味論 ($\longrightarrow_{\text{eff}}$) を図 3.3および図 3.4に示す。

$$\boxed{\langle C;\ E;\ K\rangle \longrightarrow_{\text{eff}} \langle C';\ E';\ K'\rangle}$$

$$\langle x;\ E;\ K\rangle \longrightarrow_{\text{eff}} \langle E\,(x)\,;\ E;\ K\rangle \qquad \text{(Lookup)}$$

$$\langle \texttt{let}\ x = e\ \texttt{in}\ e';\ E;\ K\rangle \longrightarrow_{\text{eff}} \langle e;\ E;\ (\texttt{let}\ x = \square\ \texttt{in}\ e', E) :: K\rangle \qquad \text{(PushLet)}$$

$$\langle w;\ E;\ (\texttt{let}\ x = \square\ \texttt{in}\ e, E') :: K\rangle \longrightarrow_{\text{eff}} \langle e;\ (x = w) :: E';\ K\rangle \qquad \text{(Bind)}$$

$$\langle \lambda x.\ e;\ E;\ K\rangle \longrightarrow_{\text{eff}} \langle \texttt{clos}\,(\lambda x.e, E)\,;\ E;\ K\rangle \qquad \text{(Close)}$$

$$\langle e\ e';\ E;\ K\rangle \longrightarrow_{\text{eff}} \langle e;\ E;\ (\square\ e', E) :: K\rangle \qquad \text{(PushApp)}$$

$$\langle w;\ E;\ (\square\ e, E') :: K\rangle \longrightarrow_{\text{eff}} \langle e;\ E';\ (w\ \square) :: K\rangle \qquad \text{(PushArg)}$$

$$\left\langle w;\ E;\ \left(\texttt{clos}\,(\lambda x.e, E)'\ \square\right) :: K\right\rangle \longrightarrow_{\text{eff}} \langle e;\ (x = w) :: E';\ K\rangle \qquad \text{(App)}$$

図 3.3　λ_{eff} の意味論 (1)

Lookup 規則にある $E\,(x)$ は環境 E 内で x に関連付けられている値を指す。 PushLet と Bind は `let` 式を表す。 Closure 規則はクロージャを生成する。 PushApp、 PushArg と App は関数適用を表すための規則である。

$$\langle \texttt{with}\ h\ \texttt{handle}\ e;\ E;\ K\rangle \longrightarrow_{\text{eff}} \langle h;\ E;\ (\texttt{with}\ \square\ \texttt{handle}\ e, E) :: K\rangle$$
(PushWithHandle)

$$\frac{\text{where } h = \texttt{handler}\ \mathit{eff}\ \ (\texttt{val}\ \ x \to e_v)\ \ ((x,k) \to e_{\mathit{eff}})}{\langle h;\ E;\ K\rangle \longrightarrow_{\text{eff}} \langle \texttt{closh}\,(h, E)\,;\ E;\ K\rangle}$$
(CloseHandler)

$$\left\langle \begin{array}{c} w_h; \\ E'; \\ (\texttt{with}\ \square\ \texttt{handle}\ e, E) :: K \end{array} \right\rangle \longrightarrow_{\text{eff}} \left\langle \begin{array}{c} e; \\ E; \\ \left((\texttt{with}\ w_h\ \texttt{handle}\ \square)^{\mathit{eff}}\right) :: K \end{array} \right\rangle$$
$$\text{where } w_h = \texttt{closh}\,(\texttt{handler}\ \mathit{eff}\ (\texttt{val}\ x \to e_v)\ \ ((x,k) \to e_{\mathit{eff}})\,, E)$$
(Handle)

$$\langle \texttt{perform}\ \mathit{eff}\ v;\ E;\ K\rangle \longrightarrow_{\text{eff}} \langle v;\ E;\ (\texttt{perform}\ \mathit{eff}\ \square) :: K\rangle$$ (PushPerform)

$$\frac{K\ //\ \mathit{eff}\ = \left(K', (\texttt{with}\ w_h\ \texttt{handle}\ \square)^{\mathit{eff}}\,, K''\right) \quad \text{where } w_h = \texttt{closh}\,(\texttt{handler}\ \mathit{eff}\ \ (\texttt{val}\ x \to e_v)\ \ ((y,k) \to e_{\mathit{eff}})\,, E')}{\langle w;\ E;\ (\texttt{perform}\ \mathit{eff}\ \square) :: K\rangle \longrightarrow_{\text{eff}} \left\langle \begin{array}{c} e_{\mathit{eff}}; \\ (y = w) :: (k = K' * E) :: E'; \\ (\texttt{with}\ w_h\ \texttt{handle}\ \square)^{\mathit{eff}}\ :: K'' \end{array} \right\rangle}$$
(HandlePerform)

$$\frac{F = (\texttt{with}\ w_h\ \texttt{handle}\ \square)^{\mathit{eff}} \quad \text{where } w_h = \texttt{closh}\,(\texttt{handler}\ \mathit{eff}\ \ (\texttt{val}\ x \to e_v)\ \ ((y,k) \to e_{\mathit{eff}})\,, E')}{\langle w;\ E;\ F :: K\rangle \longrightarrow_{\text{eff}} \langle e_v;\ (x = w) :: E';\ K\rangle}$$
(HandleValue)

図 3.4　λ_{eff} の意味論 (2)

残りの規則が代数的効果およびハンドラに関する操作を表している。 PushWithHandle はスタック K に評価文脈を積み、 CloseHandler はハンドラと環境のペアを作る。 Handle 規則は with 式 $\texttt{with}\ h\ \texttt{handle}\ e$ の操作をおこなう。 h をハンドラの値に評価し、 e をそのハンドラのもとで評価する。 PushPerform はエフェクト *eff* を発生させる perform 式のフレームをスタックに積む。 HandlePerform と HandleValue はハンドル操作における重要な規則である。 HandlePerForm 規則において、遷移前のコードは値 w である。したがって、スタックフレームの先頭 $\texttt{perform}\ \mathit{eff}\ \square$ は PushPerform 規則で取り込まれ評価された perform 式である。そして、エフェクト *eff* のハンドラを K から探す。ハンドラが見つかった場合、エフェクトの引数 w を y に、スタック K' と環境 E から成る限定継続を k に束縛し、ハンドラのボディを評価する。 λ_{eff} は**深い** (deep) ハンドラを採用し

ているため、ハンドラ w_h でハンドルしている式 $(\texttt{with}\ w_h\ \texttt{handle}\ \Box)^{\mathit{eff}}$ はこの状態遷移のあともスタックの先頭にあり続ける。 HandleValue 規則はハンドルされている式がエフェクトを発生させず、値 w を返すときに用いられる。値ハンドラ $(\texttt{val}\ x \to e_v)$ が x に w を束縛し、 e_v を評価し、スタックからハンドル式を取り除く。

3.2 λ_{ac}

λ_{ac} は de Moura と Ierusalimschy により定義された言語 [20] に基づいた非対称コルーチンを持つ言語である。 3.4 節で述べる変換のため、コルーチンに加え、再帰を含む `let` 式、パターンマッチ、比較演算子を言語機能に含む。

3.2.1 構文

図 3.5に λ_{ac} の構文を示す。 λ_{eff} からのプログラム変換のターゲット言語として λ_{ac}

$$
\begin{array}{rcl}
x & \in & \textit{Variables} \\
K & \in & \{\textit{Eff}, \textit{Resend}, \textit{True}, \textit{False}\} \\
l & \in & \textit{Labels} \\
\mathit{eff} & \in & \textit{Effects} \\
v & ::= & \texttt{nil} \mid \mathit{eff} \mid K\ \overrightarrow{v^\star} \mid l \mid x \mid \lambda x.e \\
e & ::= & v \mid K\ \overrightarrow{e^\star} \mid l : e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & \mid & \texttt{match}\ e\ \texttt{with}\ \mathit{cases} \\
 & \mid & \texttt{create}\ e \mid \texttt{resume}\ e\ e \mid \texttt{yield}\ e \\
\mathit{letrec} & ::= & \texttt{let rec}\ x\ \overrightarrow{x} = e\ \left[\overrightarrow{\texttt{and}\ x\ \overrightarrow{x} = e^\star}\right]\ \texttt{in}\ e \\
\mathit{cases} & ::= & \overrightarrow{\mathit{pat}\ [\mathit{cond}] \to e;} \\
\mathit{cond} & ::= & \texttt{when}\ x = x \\
\mathit{pat} & ::= & K\ \overrightarrow{\mathit{pat}^\star} \mid x \\
C & ::= & \Box \mid C\ e \mid v\ C \mid \texttt{let}\ x = C\ \texttt{in}\ e \mid \texttt{let}\ x = v\ \texttt{in}\ C \\
 & \mid & \texttt{match}\ C\ \texttt{with}\ \mathit{cases} \mid \texttt{let rec}\ f\ \overrightarrow{x} = e\ \texttt{in}\ C \\
 & \mid & C = e \mid \mathit{eff} = C \\
 & \mid & \texttt{let rec}\ f\ \overrightarrow{x} = e\ \overrightarrow{\texttt{and}\ f\ \overrightarrow{x} = e^\star}\ \texttt{in}\ C \\
 & \mid & \texttt{create}\ C \mid \texttt{resume}\ C\ e \mid \texttt{resume}\ l\ C \mid \texttt{yield}\ C \mid l : C
\end{array}
$$

図 3.5 λ_{ac} の構文

を用いるため、そのための要素が言語にいくつか組み込まれている。 *Effects* とその要素は λ_{eff} のエフェクトに対応する。 de Moura と Ierusalimschy の体系に加え、条件分岐、パターンマッチと (相互) 再帰を追加した。 $f\ \overrightarrow{x}$ は $f\ x_0\ x_1\ \cdots\cdots x_n$、 $\overrightarrow{\texttt{and}\ g\ \overrightarrow{y} = e}$ は

$g_0\ \overrightarrow{y_0} = e_0$ and $g_1\ \overrightarrow{y_1} = e_1$ and $\cdots\cdots$ and $g_m\ \overrightarrow{y_m} = e_m$ の略記である。また、これに類似する略記をコンストラクタおよびパターンマッチにも用いる。

非対称コルーチンの構成要素は、ラベル、ラベル式 $l : e$、 `create`、 `resume` と `yield` である。ラベルはコルーチンを指す参照であり、ラベル式 $l : e$ はコルーチン l 中の式 e である。

K はコンストラクタを表す。特に、 *True* と *False* は真理値を表す定数である。 match e $withK\,cases$ はパターンマッチを表す。 λ_{ac} はさらに、制限されたガード句 $K\ \overrightarrow{x}$ when $y = z \to e$ をパターンマッチに使うことができる。ガードは二項演算子 $(=)$ と変数のみから成るという制限は、変換のターゲット言語としての利用目的にとっては充分である。

C は評価文脈を表し、項 e が文脈 C 中にあるのを $C[e]$ と書ける。

3.3 意味論

3.3.1 補助関数

λ_{ac} の意味論を定義するため、パターンマッチに関して導入する補助関数について説明する (図 3.6)。 $FV_p(pat)$ は *pat* 中の自由変数の集合を表す。 $matchable(v, pat)$ は与えら

$$FV_p\left(K\ \overrightarrow{pat}\right) = \bigcup p \in \overrightarrow{pat}.FV_p(p)$$
$$FV_p(x) = \{x\}$$

$$matchable\left(K\ \overrightarrow{v}, K'\ \overrightarrow{pat}\right) = K =_K K' \wedge \forall v \in \overrightarrow{v}, p \in \overrightarrow{pat}.matchable(v, p)$$

$$\theta_1 \oplus \theta_2 = \emptyset \left[\begin{array}{l} \forall x \in dom(\theta_1).x \leftarrow \theta_1(x), \\ \forall y \in dom(\theta_2).y \leftarrow \theta_2(y) \end{array}\right]$$

$$genstore\left(K\ \overrightarrow{v}, K\ \overrightarrow{pat}\right) = \bigoplus_{v \in \overrightarrow{v}, p \in \overrightarrow{pat}} genstore(v, p)$$
$$genstore(v, x) = \emptyset[x \leftarrow v]$$

図 3.6 λ_{ac} の意味論の補助関数

れた値 v とパターン *pat* について、値がパターンにマッチするかどうかを判定する述語で

ある。 $\oplus$ は 2 つのストア σ_1 と σ_2 を結合する演算子である。ストアは変数またはラベルから値への部分写像である。 $\emptyset$ は空のストアである。 *genstore* 関数は、与えられたコンストラクタの値とコンストラクタのパターンから新たにストアを作る関数である。例えば、適当な値 w、 v、 u に対して $genstore\left(\left(Resend\ \left(Eff\ w\ v\right)\ u\right), \left(Resend\ \left(Eff\ y\ y\right)\ k\right)\right)$ は $\emptyset\left[y \leftarrow w, x \leftarrow v, k \leftarrow u\right]$ を生成する。

3.3.2 小ステップ意味論

式とストアの 2 つ組 $\langle e, \theta \rangle$ の状態遷移 ($\longrightarrow_{\mathrm{ac}}$) で表される λ_{ac} の小ステップ意味論を図 3.7 および図 3.8に示す。

$$\langle C[x], \theta \rangle \longrightarrow_{\mathrm{ac}} \langle C[\theta(x)], \theta \rangle \qquad \text{(Lookup)}$$

$$\frac{x \notin dom\left(\theta\right)}{\langle C\left[\left(\lambda x.e\right) v\right], \theta \rangle \longrightarrow_{\mathrm{ac}} \langle C\left[e\right], \theta\left[x \leftarrow v\right] \rangle} \qquad \text{(App)}$$

$$\frac{x \notin dom\left(\theta\right)}{\langle C[\texttt{let}\ x = v\ \texttt{in}\ e'], \theta \rangle \longrightarrow_{\mathrm{ac}} \langle C[e], \theta[x \leftarrow v] \rangle} \qquad \text{(Let)}$$

$$\frac{\forall z \in \left\{f, \overrightarrow{x}, g, \overrightarrow{\overrightarrow{y}}\right\}.z \notin dom\left(\theta\right)}{\left\langle C\left[\begin{array}{l}\texttt{let rec}\ f\ \overrightarrow{x} = e_f \\ \overrightarrow{\texttt{and}\ g\ \overrightarrow{y} = e_g} \\ \texttt{in}\ e\end{array}\right], \theta\right\rangle \longrightarrow_{\mathrm{ac}} \left\langle C[e], \theta\left[\begin{array}{l} f \leftarrow \lambda\overrightarrow{x}.e_f, \\ \overrightarrow{g \leftarrow \lambda\overrightarrow{y}.e_g}\end{array}\right]\right\rangle} \qquad \text{(LetRec)}$$

$$\frac{l \notin dom\left(\theta\right)}{\langle C\left[\texttt{create}\ v\right], \theta \rangle \longrightarrow_{\mathrm{ac}} \langle C\left[l\right], \theta\left[l \leftarrow v\right] \rangle} \qquad \text{(Create)}$$

$$\langle C\left[\texttt{resume}\ l\ v\right], \theta \rangle \longrightarrow_{\mathrm{ac}} \langle C\left[l : \theta\left(l\right)\ v\right], \theta\left[l \leftarrow \texttt{nil}\right] \rangle \qquad \text{(Resume)}$$

$$\langle C_1\left[l : C_2\left[\texttt{yield}\ v\right]\right], \theta \rangle \longrightarrow_{\mathrm{ac}} \langle C_1\left[v\right], \theta\left[l \leftarrow \lambda x.C_2\left[x\right]\right] \rangle \qquad \text{(Yield)}$$

$$\langle C\left[l : v\right], \theta \rangle \longrightarrow_{\mathrm{ac}} \langle C\left[v\right], \theta \rangle \qquad \text{(LabelledReturn)}$$

図 3.7　λ_{ac} の意味論 (1)

$dom\left(\rho\right)$ は θ の始域を表し、 $\theta\left(x\right)$ は x の指す値を表す。 $\theta\left(l\right)$ が `nil` を指す場合も l を $dom\left(\rho\right)$ に含める。変数またはラベルを追加していく規則 (App、 Let、 LetRec、 Create、 Match と MatchWhen) では、 α 同値な項を区別し、被りのないように適宜変数名を変更する。パターンにおける_は変数と同じように扱われるが、パターンマッチにおいて変数束縛を

$$\frac{op =_{op} op'}{\langle C\,[op = op']\,, \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C\,[True]\,, \theta\rangle} \quad \text{(EqT)}$$

$$\frac{op \neq_{op} op'}{\langle C\,[op = op']\,, \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C\,[False]\,, \theta\rangle} \quad \text{(EqF)}$$

$$\frac{\neg matchable\,(K\ \overrightarrow{v}, pat)}{\left\langle C\left[\begin{array}{l}\texttt{match}\ K\ \overrightarrow{v}\ \texttt{with}\\ pat\ \ [cond] \rightarrow e;\\ cases\end{array}\right], \theta\right\rangle \longrightarrow_{\mathrm{ac}} \langle C\,[\texttt{match}\ K\ \overrightarrow{v}\ \texttt{with}\ cases]\,, \theta\rangle} \quad \text{(MatchNext)}$$

$$\frac{\begin{array}{c}\forall x \in FV_p\,(pat)\,.x \notin \mathrm{dom}\,(\theta) \qquad matchable\,(K\ \overrightarrow{v}, pat)\\ \theta' = \theta \oplus genstore\,(K\ \overrightarrow{v}, pat)\end{array}}{\langle C\,[\texttt{match}\ K\ \overrightarrow{v}\ \texttt{with}\ pat \rightarrow e; cases]\,, \theta\rangle \longrightarrow_{\mathrm{ac}} \langle C[e], \theta'\rangle} \quad \text{(Match)}$$

$$\frac{\begin{array}{c}\forall x \in FV_p\,(pat)\,.x \notin \mathrm{dom}\,(\theta) \qquad matchable\,(K\ \overrightarrow{v}, pat)\\ \theta' = \theta \oplus genstore\,(K\ \overrightarrow{v}, pat)\end{array}}{\left\langle C\left[\begin{array}{l}\texttt{match}\ K\ \overrightarrow{v}\ \texttt{with}\\ pat\ \texttt{when}\ c \rightarrow e;\\ cases\end{array}\right], \theta\right\rangle \longrightarrow_{\mathrm{ac}} \left\langle C\left[\begin{array}{l}\texttt{match}\ c\ \texttt{with}\\ True \rightarrow e;\\ False \rightarrow\\ \quad \texttt{match}\ K\ \overrightarrow{v}\ \texttt{with}\\ \quad cases\end{array}\right], \theta'\right\rangle} \quad \text{(MatchWhen)}$$

図 3.8　λ_{ac} の意味論 (2)

おこなわず、また_は上書きすることができる。

規則には、まず変数の指す値を参照する Lookup、関数適用を表す App、 `let` と `let rec` を表す Let および LetRec がある。 Create はコルーチンを生成するための規則である。新たなラベル l を作り、コルーチンを l に束縛し、文脈 C にそのラベルを返す。 Resume 規則は、ラベル l と関数適用 $\theta\,(l)\ v$ から成るラベル式を作る。 $\theta\,(l)\ v$ はラベル l に対応する値を θ から探し、それに v を適用する。作られたラベル式 $l : \theta\,(l)\ v$ は l でラベル付けされた計算を表している。コルーチンの残りの計算が参照されるのを防ぐため、 Resume 規則では参照している値を `nil` に設定して、無効な値が参照されるようにする。 Yield は現在のコルーチンの計算を停止し、 `yield` の引数を親のスレッドに渡す。対象の体系が持っているのは非対称コルーチンなので、コルーチンは `resume` を使うことで他のコルーチンの親スレッドとして振る舞う。 LabelledReturn はコルーチン l の計算結果 v を呼び出し元に渡す。

EqT と EqF は 2 つのエフェクトを比較する規則である。 $=_{eff}$ 演算子は与えられた 2 つのエフェクトが同じかどうかを判別する。 Match と MatchWehen はパターンマッチを表す規則である。後者は $K\ \vec{v}$ がパターンにマッチし、句がガード c を持っている場合に適用される。この規則は、 *genstore* を利用して $\vec{v}$ を対応するパターンの変数に代入し、ガードを新たなマッチ式に変形する。ガードが `True` を返す場合、パターンマッチが成功して `True` のボディを評価し、 `False` を返す場合には残りのパターンを走査していく。

3.4 λ_{eff} から λ_{ac} への変換

本節では λ_{eff} から λ_{ac} へのプログラム変換について説明する。図 3.9にその変換を示す。 λ_{eff} の項 e に対し、 $[\![e]\!]\,\eta$ は λ_{eff} の変数から λ_{ac} への有限写像 η を用いて変換された λ_{ac} の項である。 η は e 中に現れる自由変数を変換するのに用いられる。 $\eta\,[x \to x']$ で x を x' にマップし、 $\eta\,(y)$ で λ_{eff} の変数 y が指す λ_{ac} の変数を返す。変換は、変数の指す値の参照、 λ 抽象、関数適用、 `let` 式を含むそれぞれの式に対して準同型となる。この変換は、次の観点から `perform` を `yield` に写像する。代数的効果では、エフェクトが発生すると、コントロールがそのエフェクトに対応するハンドラに移る。一方コルーチンでは、 `yield` がコルーチンの中で呼ばれた場合、そのコルーチンを呼び出している `resume` にコントロールが移る。したがって、この動作を用いて `perform` を `yield` でエミュレーションしている。この変換では、 `perform` の引数も変換して *Eff* というタグを付ける。このタグは、後述するハンドラの変換で用いられる。

ハンドル式 `with` h `handle` e は、ハンドラが関数に変換されるため、関数適用の形に変換される。

ハンドラの変換は他の変換と比較して複雑である。はじめに λ_{ac} 上に *handler* 関数を定義し、そして変換されたハンドラの要素を関数に渡す。

handler 関数は、エフェクトを表す *eff*、値ハンドラを表す *vh*、エフェクトハンドラを表す *effh*、そしてハンドルされる計算のサンクである *th* の 4 つを引数にとる。内部では *th* から成るコルーチン *co* と、 3 つの関数 *continue*、 *rehandle* と *handle* を定義する。 *continue* は 1 つの引数を受け取って `resume` *co* にそれを渡し、コルーチン *co* の残りの計算を実行する。 *continue* は**継続**を実行すると考えることができる。そして、継続の計算結果を *handle* に渡す。継続の結果をハンドルすることで、ハンドラは**深い**ハンドラとして定義される。

rehandle は k *arg* をハンドルするハンドラを新たに生成する。 *continue* を値ハンドラに設定することで、生成されたハンドラは最終的に *co* の残りの計算を実行する。

handle はハンドルされる式から渡る値を r として受け取る。 r のタグでパターンマッチをおこない、各句のボディに実行を移す。パターンマッチの最初のガード付きパターン (*Eff* *eff*$'$ v `when` *eff*$' = $ *eff*) はエフェクトの発生を捕捉する。ハンドルされている式の中でエフェクト *eff*$'$ が引数 v をともなって発生したとき、 r は *Eff* *eff*$'$ v という形になる。 *eff*$'$

$$
\begin{aligned}
\llbracket x \rrbracket \eta &= \eta\,(x) \\
\llbracket \lambda x.e \rrbracket \eta &= \lambda x'.\llbracket e \rrbracket \eta\,[x \mapsto x'] \\
\llbracket v_1\ v_2 \rrbracket \eta &= (\llbracket v_1 \rrbracket \eta)\ (\llbracket v_2 \rrbracket \eta) \\
\llbracket \texttt{let}\ x = e\ \texttt{in}\ e' \rrbracket \eta &= \texttt{let}\ x' = \llbracket e \rrbracket \eta\ \texttt{in}\ \llbracket e' \rrbracket \eta\,[x \mapsto x'] \\
\llbracket \texttt{perform}\ \mathit{eff}\ v \rrbracket \eta &= \texttt{yield}\ (\mathit{Eff}\ \mathit{eff}\ (\llbracket v \rrbracket \eta)) \\
\llbracket \texttt{with}\ h\ \texttt{handle}\ e \rrbracket \eta &= \llbracket h \rrbracket \eta\ (\lambda\_.\llbracket e \rrbracket \eta) \\
\llbracket \texttt{handler}\ \mathit{eff}\ (\texttt{val}\ x \to e_v)\ ((y,k) \to e_{\mathit{eff}}) \rrbracket \eta &= \\
&\quad \texttt{let}\ \mathit{vh} = \lambda x'.\llbracket e_v \rrbracket \eta\,[x \mapsto x']\ \texttt{in} \\
&\quad \texttt{let}\ \mathit{effh} = \lambda y'\ k'.\llbracket e_{\mathit{eff}} \rrbracket \eta\,[y \mapsto y', k \mapsto k']\ \texttt{in} \\
&\quad \mathit{handler}\ \mathit{eff}\ \mathit{vh}\ \mathit{effh}
\end{aligned}
$$

where $\mathit{handler} =$

```
let rec handler eff vh effh th =
  let co = create th in
  let rec continue arg = handle (resume co arg)
  and rehandle k arg = handler eff continue effh (λ_.k arg)
  and handle r =
    match r with
    | Eff eff' v                when eff' = eff   → effh v continue
    | Eff _ _                                     → yield (Resend r continue)
    | Resend (Eff eff' v) k     when eff' = eff   → effh v (rehandle k)
    | Resend effv k                               → yield (Resend effv (rehandle k))
    | _                                           → vh r
  in continue nil
in handler
```

図 3.9　λ_{eff} から λ_{ac} への変換

が *eff* と一致する場合、ハンドラによってハンドルでき、エフェクトハンドラ *effh* に v と *continue* が渡される。 2 つ目の句はハンドルできないエフェクトがハンドルされる式から渡った場合である。この場合、 `yield` に *Resend r continue* を渡してエフェクトを上位のハンドラに再送する。 3 つ目の句と 4 つ目の句は上記で述べた 2 つ目の句で再送されたエフェクトの解決をおこなう。 3 つ目の句は *eff*′ が *eff* と同値であり、ハンドラによってハンドルできる場合を表している。 1 つ目の句と同様に *effh* に v を渡すが、第 2 引数は *continue* ではなく *rehandle k* を継続として渡す。 k は内部から渡ってきた継続であり、 *continue* は現在のハンドラが参照できる継続である。前者の継続を実行したあとに後者の継続を実行する

ため、 k を、**現在の**継続を持つ *rehandle* でラップする。

rehandle はコルーチンのレイヤを調整する役割を担っている。 2 つ目の句で *handle* が `yield` を呼ぶため、コルーチン 1 層から制御を抜けることになる。 3 つ目と 4 つ目の句で、継続の戻り値を操作するためだけなら $rehandle\ k$ ではなく $\lambda arg.handle\ (k\ arg)$ と書くこともできる。そうした場合、コルーチンの層が “薄く” なっていき、最終的にコルーチンの外側で `yield` を呼びエラーとなってしまう。そのため、 *rehandle* では内部でハンドルされる式をコルーチンでカプセル化し、コルーチンの層が薄くなるのを回避している。

4 つ目の句ではハンドルできないエフェクトを再送する。 2 つ目の句とほとんど同じだが、上記で述べた 3 つ目の句と同様の理由で、継続を *rehandle* を使って変更している。最後の句は他の全ての値にマッチし、値ハンドラにそれが渡される。内部で利用する関数を定義したのち、 *handler* は *continue* に `nil` を渡して実行する。

この変換は複雑に見えるが、改めて、この変換は局所的かつ合成的におこなわれる構文指向の変換であり、高階のストアや複雑でほとんどのプログラム言語が持ってないような機能は使用せず、非対称コルーチンの標準的な機能だけに依存している。この簡潔さのため、我々の変換に基づいたライブラリの他言語への移植が第三者によってすでにおこなわれている。

第 4 章

変換のマクロ表現可能性

第 3 章では λ_{eff} の項から λ_{ac} の項への変換を定義した。 Felleisen はプログラム言語の表現力を比較するために、マクロ表現可能性という性質を定義した [10]。この性質により、チューリング完全よりも粒度の細かい計算の複雑さに関する議論が可能となる。 Forster らは call-by-push-value 計算体系に基づき、代数的効果、 monadic reflection、 $\mathrm{shift}_0/\mathrm{reset}_0$ のそれぞれについて表現力の比較をおこなった [11]。本章では、型なしラムダ計算を拡張した 2 つの言語 λ_{eff} と λ_{ac} について、前者から後者への変換がマクロ表現可能であることを示す。まず、第 3 章で示した変換を、変数を置き換えない変換に変えたものを図 4.1に示す。 この変換を用いて、次の定理を示す。

定理 1. **λ_{eff} の項から λ_{ac} の項への変換 $[\![-]\!]$ は *Felleisen* の意味でマクロ表現可能な変換である。**

証明. 変換の定義のそれぞれの場合において、 Felleisen の意味でのマクロ表現可能性を満たすことを確認する。

- $[\![x]\!]$ のとき:
  $[\![x]\!] = x$ となり、明らかにマクロ表現可能な変換である。
- $[\![\lambda x.e]\!]$ のとき:
  $[\![\lambda x.e]\!] = \lambda x.[\![e]\!]$ となる。右辺は変換される項の部分項に対する変換を組み合わせた形をしているので、マクロ表現可な変換である。
- $[\![\mathtt{let}\ x = e\ \mathtt{in}\ e']\!]$ のとき:
  $[\![\mathtt{let}\ x = e\ \mathtt{in}\ e']\!] = \mathtt{let}\ x = [\![e]\!]\ \mathtt{in}\ [\![e']\!]$ となる。右辺は変換される項の部分項に対する変換を組み合わせた形をしているので、マクロ表現可な変換である。
- $[\![v_1\ v_2]\!]$ のとき:
  $[\![v_1\ v_2]\!] = [\![v_1]\!]\ [\![v_2]\!]$ となる。右辺は変換される項の部分項に対する変換を組み合わせた形をしているので、マクロ表現可な変換である。
- $[\![\mathtt{perform}\ \mathit{eff}\ v]\!]$ のとき:

$$
\begin{aligned}
\llbracket x \rrbracket &= x \\
\llbracket \lambda x.e \rrbracket &= \lambda x.\llbracket e \rrbracket x \\
\llbracket v_1\ v_2 \rrbracket &= (\llbracket v_1 \rrbracket)\ (\llbracket v_2 \rrbracket) \\
\llbracket \texttt{let}\ x = e\ \texttt{in}\ e' \rrbracket &= \texttt{let}\ x = \llbracket e \rrbracket\ \texttt{in}\ \llbracket e' \rrbracket \\
\llbracket \texttt{perform}\ \mathit{eff}\ v \rrbracket &= \texttt{yield}\ (\mathit{Eff}\ \mathit{eff}\ (\llbracket v \rrbracket)) \\
\llbracket \texttt{with}\ h\ \texttt{handle}\ e \rrbracket &= \llbracket h \rrbracket\ (\lambda\_.\llbracket e \rrbracket) \\
\llbracket \texttt{handler}\ \mathit{eff}\ (\texttt{val}\ x \to e_v)\ ((y,k) \to e_{\mathit{eff}}) \rrbracket &= \texttt{let}\ \mathit{vh} = \lambda x.\llbracket e_v \rrbracket\ \texttt{in} \\
&\quad\ \texttt{let}\ \mathit{effh} = \lambda y\ k.\llbracket e_{\mathit{eff}} \rrbracket\ \texttt{in} \\
&\quad\ \mathit{handler}\ \mathit{eff}\ \mathit{vh}\ \mathit{effh}
\end{aligned}
$$

where $\mathit{handler} =$

```
let rec handler eff vh effh th =
  let co = create th in
  let rec continue arg = handle (resume co arg)
  and rehandle k arg = handler eff continue effh (λ_.k arg)
  and handle r =
    match r with
    | Eff eff' v              when eff' = eff  → effh v continue
    | Eff _ _                                  → yield (Resend r continue)
    | Resend (Eff eff' v) k   when eff' = eff  → effh v (rehandle k)
    | Resend effv k                            → yield (Resend effv (rehandle k))
    | _                                        → vh r
  in continue nil
in handler
```

図 4.1　変数を置き換えない λ_{eff} から λ_{ac} へのマクロ変換

$\llbracket \texttt{perform}\ \mathit{eff}\ v \rrbracket = \texttt{yield}\ (\mathit{Eff}\ \mathit{eff}\ \llbracket v \rrbracket)$ となる。右辺は変換される項の部分項に対する変換を組み合わせた形をしているので、マクロ表現可な変換である。

- $\llbracket \texttt{with}\ h\ \texttt{handle}\ e \rrbracket$ のとき:
  $\llbracket \texttt{with}\ h\ \texttt{handle}\ e \rrbracket = \llbracket h \rrbracket\ (\lambda\_.\llbracket e \rrbracket)$ となる。右辺は変換される項の部分項に対する変換を組み合わせた形をしているので、マクロ表現可な変換である。
- $\llbracket \texttt{handler}\ \mathit{eff}\ (\texttt{val}\ x \to e_v)\ ((y,k) \to e_{\mathit{eff}}) \rrbracket$ のとき:
  $\llbracket \texttt{handler}\ \mathit{eff}\ (\texttt{val}\ x \to e_v)\ ((y,k) \to e_{\mathit{eff}}) \rrbracket = \texttt{let}\ \mathit{vh} = \lambda x.\llbracket e_v \rrbracket\ \texttt{in}\ \texttt{let}\ \mathit{effh} = \lambda y\ k.\llbracket e_{\mathit{eff}} \rrbracket\ \texttt{in}\ \mathit{handler}\ \mathit{eff}\ \mathit{vh}\ \mathit{effh}$ となる。また、 $\mathit{handler}$ は λ_{ac} の項として定義されており、右辺は変換される項の部分項に対する変換を組み合わせた形をしているの

で、マクロ表現可な変換である。

したがって、変換 $[\![-]\!]$ は Felleisen の意味でマクロ表現可能な変換である。 □

定理 1から、本研究の変換は局所的な変換であることが言える。また、代数的効果に関連しない部分項は変換されずにそのままであることが分かる。以上から、本研究で示した変換により、代数的効果を単純なライブラリとして実装可能であることが示される。次章では、この変換をライブラリとして実装し、様々なプログラムに適用する場合について議論する。

第 5 章

ライブラリにおける拡張と実用

第 3 章で述べた変換に基づき、ワンショットの代数的効果およびハンドラを Lua 言語と Ruby 言語上に実装した。変換は局所的かつ合成的に定義されており、変数管理はホスト言語によりおこなわれるため、ライブラリとして実装することができた。実装は単純で理解しやすいものとなっている一方、実用面等の都合により変換とは異なる部分がある。また、実際のプログラミングで本ライブラリを利用する際に発生する問題もある。本章では、実装段階で拡張された点と、実際のプログラミングで発生する問題とその対策について述べる。

5.1 複数のエフェクトをハンドルするハンドラ

本論文では、 1 つのハンドラにつき 1 つのエフェクトだけをハンドルできるという制限を λ_{eff} に与えた。しかしこれは定義を簡単にするためであり、この制限を緩和することができる。実際に、我々の提供するライブラリは 1 つのハンドラが複数のエフェクトをハンドルできるようになっており、本論文中にある、複数のエフェクトをハンドルするハンドラも問題なく動作している。複数のエフェクトをハンドルするハンドラは簡単に実装でき、 Lua 言語では table を利用し、 Ruby 言語では Hash を利用している。これらの実装において、ハンドラが複数のエフェクトをハンドルすることによる致命的なパフォーマンスの低下はおこらない。

5.2 動的なエフェクト生成

λ_{eff} はエフェクトの集合は前もって定義されていることを前提にしており、新たにエフェクトのインスタンスを生成する方法はない。しかしこれもまた言語の定義を簡単にするためであり、実際の実装ではこの制約は取り払われている。エフェクトインスタンスが動的に生成できるため、 Kiselyov らの例 [17] にあるような、エフェクトインスタンスが一意性を必要とするような例を実装することができる。

エフェクトインスタンスの動的な生成を許すことで、様々なプログラムが書けるようにな

る一方、エフェクトの推論の基礎理論を複雑化させる可能性がある。エフェクトの動的生成の体系の定式化と理論展開は今後の課題となる。

5.3 既存のエフェクトとの競合

本研究で述べる変換に基づいた代数的効果の埋め込みを利用することで、プログラマは内部でどのようなデータ構造が使われているかを意識する必要がないことを第 1 章で述べた。しかしいくつか例外的なケースが存在する。我々の定義した変換は、すべてのエフェクトは代数的効果およびハンドラの機能によって定義されているという仮定に基づいている。もしソースのプログラムが、代数的効果以外の副作用を使用している場合、その副作用が内部で使われているコルーチンと競合し、予期しないエラーが発生する可能性がある。ここでは例として (言語組み込みの) コルーチンを挙げる。我々の提供する Lua のライブラリを利用し、かつ代数的効果と Lua 組み込みのコルーチンを同時に使用した場合、プログラム内でエフェクトを発生させると意図せずコルーチンによって捕捉される可能性がある。そのため、組み込みのコルーチンと (我々の提供する) 代数的効果およびハンドラを同時に使うべきではない。この問題は、代数的効果およびハンドラの表現力を以て解決することができる。以下に Lua のコードを示す。

```
local Yield = inst()

local yield = function(v)
  return perform(Yield, v)
end

local create = function(f)
  return { it = f, handled = false }
end

local resume = function(co, v)
  if co.handled then
    return co.it(v)
  else
    co.handled = true
    return handler({
      val = function(x) return x end,
      [Yield] = function(u, k)
        co.it = k
```

```
      return u
    end
  })(function()
    return co.it(v)
  end)
 end
end
```

このコードの下半分は、代数的効果によるコルーチンの実装である。 yield関数は resume に値を投げるような実装になるべきなので、 yieldはエフェクトの発生、 resumeはハンドラのようになる必要がある。この対応関係は図 3.9で述べた変換と反対になる。 create は table により参照セルを作る。コルーチンは参照セルで表現され、関数 fと、後述する handledというフラグで初期化される。 resumeのハンドラは yieldの発生を捕捉し、その引数と継続を取得する。この継続はコルーチンの残りの計算部分に相当し、ハンドラはこの継続をセルに保存し、引数 uを返す。我々の提供するハンドラは *deep* であるため、ハンドラを 2 回以上設定する必要はない。 handledフラグはその関数がすでにハンドルされているか否かを判別するために使われる。 resume関数はこのフラグをチェックし、フラグが立っていなければ、フラグを立て、ハンドラのもとで関数を実行する。フラグが立っていれば、関数をそのまま実行する。

コルーチンに関する問題は上で述べたとおりだが、 1 つのプログラム中に様々な副作用が含まれうる場合は、上記の対応を更に拡張する必要あがある。

第 6 章

評価

本章では我々のおこなったベンチマークについて述べる。我々の実装した Lua 言語上のライブラリと、 Free モナドを利用した代数的効果 [23] の Lua 実装を用いてマイクロベンチマークを取り、パフォーマンスを比較した。ベンチマークは GitHub 上で公開している[*1]。

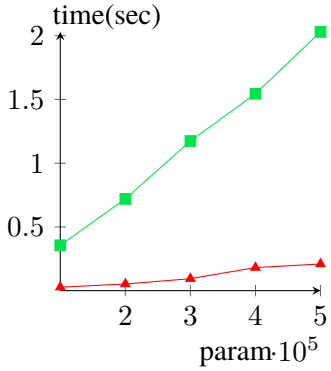
ベンチマークの結果を表す図中に現れる記号は、▲は我々のライブラリの結果を表し、■は Free モナドに基づいた実装の結果を表す。ベンチマークのうち 1 つでは Lua 組み込みのコルーチンとの比較をおこない、その結果は ⋆ で表される。ベンチマークは表 6.1に示す環境のもとでおこなわれた。

表 6.1　ベンチマーク環境

OS	Arch Linux
CPU	Intel Core i7-8565U
メモリ	16GB DDR4
Lua 処理系	LuaJIT 2.05

図 6.1は State モナドのエミュレーションのベンチマークである。 このベンチマークでは [15] から引用した `count`を、 Free モナドと我々のライブラリが使えるように調整して利用している。 `count`は 1 つのレイヤの 1 つのエフェクトハンドラと協調し、入力として受け取った整数の回数だけ再帰的に実行される。結果は、我々のライブラリが Free モナド実装の約 10 倍の実行速度となった。 Free モナドが遅かった理由は、 `bind` 演算子が次の計算ステップとして継続を要求するが、 Lua のような手続き的言語においては関数クロージャを作るコストが高いためと考えられる。また、 Haskell のような関数型言語の処理系は (Free) モナドに対する最適化をおこなう場合が多いが、このベンチマークで利用した Lua 処理系はそういった最適化はおこなわない。少なくともそのような最適化をおこなわない処理系においては、我々のライブラリ実装は Free モナドを用いる場合よりも高速であることがベンチマー

[*1] `https://github.com/nymphium/effs-benchmark`

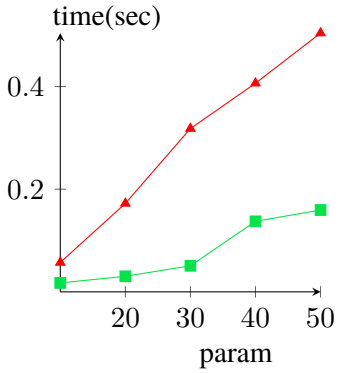


param ($10^5\times$)	1	2	3	4	5
free	0.3560	0.7199	1.1741	1.5449	2.0303
ours	0.0273	0.0521	0.0939	0.1811	0.2090

図 6.1　`onestate` のベンチマーク結果

ク結果より裏付けられる。

次の実験結果を図 6.2に示す。 このベンチマークでは `count`関数を入れ子になったハンド

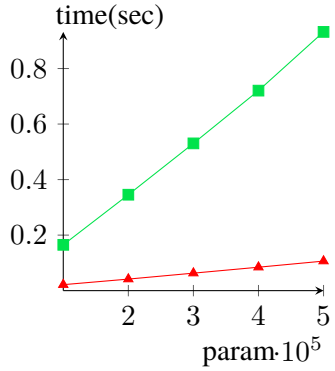


param ($10\times$)	1	2	3	4	5
free	0.0174	0.0303	0.0508	0.1374	0.1594
ours	0.0572	0.1722	0.3181	0.4064	0.5041

図 6.2　`multistate` のベンチマーク結果

ラの中で 3000 回実行する。表中のパラメータは入れ子になるハンドラの数を表し、 50 個の入れ子のハンドラは現実的ではないが、ベンチマーク用の極端な例として追加した。図が示すように、 Free モナドの場合と比較して我々のライブラリは約 3 倍の遅さとなった。図 3.9 に示すように、 *rehandle* が他のハンドラから再送されてきたエフェクトを捕捉するごとに呼ばれ、新たにコルーチンを生成していることがパフォーマンス低下の原因となったと考えられる。一方で、 Free モナドの場合も入れ子になるハンドラの数に比例してパフォーマンスが低下している。代数的効果の実装において、エフェクトを再送するためにかかるオーバーヘッドを回避するために、入れ子になるハンドラの数には注意して使用する必要がある。

次のベンチマークでは、 `looper`関数が図 6.3の表に示されるパラメータをイテレーションを回す回数として受け取り、 `for` ループ 1 回ごとにエフェクトを発生させる。 ハンドラはループの外側に設定され、ループ内で発生するエフェクトを捕捉する。我々のライブラリは Free モナドに基づいた実装の 9 倍高速に動作した。 Free モナドは単純な `for` オペレー

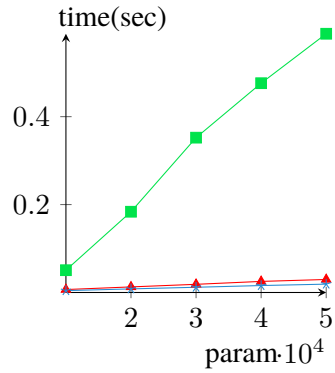


param ($10^5\times$)	1	2	3	4	5
free	0.1650	0.3455	0.5304	0.7204	0.9319
ours	0.0215	0.0416	0.0630	0.0844	0.1064

図 6.3　`looper` のベンチマーク結果

タではなく `forM` オペレータが必要になり、この演算子の違いがオーバーヘッドとなる。`onestate` のベンチマークで述べたことと同様に、このオーバーヘッドは言語処理系によっては最適化により消すことができる場合がある。

図 6.4は same-fringe 問題を代数的効果またはコルーチンを用いて解くベンチマーク結果を示している。 このベンチマークでは、パラメータとして与えられた数値を葉の数とした同じ



param ($10^4\times$)	1	2	3	4	5
free	0.0507	0.1837	0.3522	0.4761	0.5886
ours	0.0067	0.0127	0.0186	0.0252	0.0296
coroutines	0.0042	0.0082	0.0119	0.0158	0.0190

図 6.4　`same_fringe` のベンチマーク結果

木を 2 本作り、それに対する same-fringe 問題を解き、そのパフォーマンスを測定する。問題を解くために、第 5 章で述べた方法でコルーチンを代数的効果で、それぞれ我々のライブラリと Free モナドを用いて実装した。また、 Lua の組み込みのコルーチンでもソルバを実装した。我々のライブラリを用いた実装は Free モナドの実装よりも 18 倍高速となった。また、特筆すべき部分として、 Lua 組み込みのコルーチンを用いた場合と比較して、我々のライブラリを用いた実装のパフォーマンスは 1.6 倍の低下に収まった。

評価実験から、我々の示した代数的効果およびハンドラの実装方法は、様々な言語に適用するにあたり、パフォーマンスの観点において有利と考えられる。また、コルーチンを用いて計算エフェクトのあるプログラムを書くのは、高レベルな抽象を提供する代数的効果を用

いて書くよりも非常に困難である。我々の提供する代数的効果の埋め込みを用いれば、コルーチンをラップして、計算エフェクトのあるプログラムを、簡潔に、少ないオーバーヘッドで書くことができる。

第 7 章

関連研究

7.1 浅いハンドラ

本論文で示した埋め込みは、ハンドラが取得する限定継続の中で発生するエフェクトも、同じハンドラによって捕捉される**深い** (deep) ハンドラを採用した。一方、深いハンドラに対して**浅い** (shallow) ハンドラが存在し [13]、それ独自のメリットがある。第 5 章で述べた、代数的効果によるコルーチンの実装は、浅いハンドラを利用すればハンドラによりハンドルされているかどうかのフラグを使う必要がない。

我々は浅いハンドラ `handler`† もコルーチンへの変換として定義した (図 7.1)。 アイデアは単純で、ハンドラが一度エフェクトを捕捉したら、それ以降は常にエフェクトを外側のハンドラに再送する。図 3.9に示した *rehandle* の役割は、コルーチンの層を調節するために継続をコルーチンでカプセル化するのに加え、継続の中で発生するエフェクトの発生を再びハンドルすることである。浅いハンドラの場合も同様にコルーチンの層を調節する必要はあるが、継続の実行中に発生するエフェクトを再びハンドルする必要はない。

7.2 ワンショットの継続

ワンショットの継続に関する研究はすでに、理論や実行効率の側面から研究されている。Bruggeman らは、多くの継続は一度しか起動されないことに着目し、ワンショットの継続演算子 *call/1cc* を導入した [6]。継続の実行回数が高々 1 回であることを処理系 (コンパイラ) が分かっている場合、効率的なコード生成が可能になる。 Bruggeman らは、一般的な (複数回継続を実行できる) 継続演算子 *call/cc* が使用されているプログラムのうち、継続の実行が高々 1 回の部分を *call/1cc* に書き換えた場合、メモリ使用量が抑えられ、実行速度が向上することをベンチマークにより示した。 Berdine らは線形型を用いて、様々なコントロールがワンショットの継続を含む CPS 変換により表現できることを示した [3]。

James らはジェネレータが持つ *yield* オペレータを定式化した [14]。ジェネレータは制限されたコルーチンであり、多くの言語が持つワンショットの限定継続である。 James らは複

$$
\begin{aligned}
&[\![\texttt{handler}^{\dagger}\ \mathit{eff}\ (\texttt{val}\ x \to e_v)\ ((x,k) \to e_{\mathit{eff}})]\!]\eta = \\
&\qquad\qquad \texttt{let}\ \mathit{vh} = \lambda x'.[\![e_v]\!]\eta\,[x \mapsto x']\ \texttt{in} \\
&\qquad\qquad \texttt{let}\ \mathit{effh} = \lambda x'\ k'.[\![e_{\mathit{eff}}]\!]\eta\,[x \mapsto x', k \mapsto k']\ \texttt{in} \\
&\qquad\qquad \mathit{handler}^{\dagger}\ \mathit{eff}\ \mathit{vh}\ \mathit{effh}
\end{aligned}
$$

where $\mathit{handler}^{\dagger} =$

```
let rec handler eff vh effh th =
  let co = create th in
  let rec continue arg = handle (resume co arg)
  and rehandle k arg = handler eff continue effh (λ_.k arg)
  and continue' = resume co
  and rehandle' k = resume (create k)
  and handle r =
    match r with
    | Eff eff' v              when eff' = eff  → effh v continue'
    | Eff _ _                                  → yield (Resend r continue)
    | Resend (Eff eff' v) k   when eff' = eff  → effh v (rehandle k)
    | Resend effv k                            → yield (Resend effv (rehandle' k))
    | _                                        → vh r
  in continue nil
in handler
```

図 7.1　浅いハンドラからコルーチンへの変換

数回起動できる継続を持つ、一般化された *yield* を定義し、ジェネレータの *yield* 演算子との関係について述べている。本研究の埋め込みのワンショットの制限は、コルーチンスレッドをコピーできない、つまり特定の状態のコルーチンを複数回起動できないことに由来している。本研究の埋め込みを、一般化された *yield* を利用することで、ワンショットの制限のない実装ができると予想される。一方、 James らの *yield* はモナドを用いるため、後述するように、簡潔な構文という本研究の利点が失われる。

Multicore OCaml は、ランタイムのスタック操作により代数的効果をネイティブに持つ OCaml 言語の方言である。 Multicore OCaml のモチベーションは並行プログラミングを直接方式で書くことにある [8]。 Multicore OCaml はパフォーマンスを考慮して継続の実行をワンショットに制限しており、継続を複数回実行するために明示的な継続のコピー操作を提供している。 Multicore OCaml は、 OCaml を拡張した言語として定義されており、ワンショットの代数的効果をライブラリとして実装できる点において本研究と異なる。

第 2 章や [9] にあるように、ハンドラの取得できる継続がワンショットに制限されてもなお、代数的効果を用いて様々なアプリケーションを実装することができる。現在広く使われている JavaSctript 向け Web フレームワーク React には Hooks という機能[*1]があり、副作用のあるコンポーネントをモジュラーに定義できる。 React の開発者の一人である Abramov は Hooks と代数的効果の関連性について述べている[*2]。 Hooks の提供する機能はワンショットの代数的効果で実現できると考えられる。

7.3 Free モナド

モナドは主に `return` 演算子と `bind` 演算子から成る。 `bind` は、簡単に述べると、第 1 引数の値と、それ以降の計算、つまり継続として受け取った第 2 引数の関数で計算を進める。そのため、モナドを用いたプログラミングは、本質的には CPS をプログラマが書くことになる。 Free モナドとそれを用いた代数的効果の埋め込みも同様である。 CPS が書ければよいので、 Free モナドを用いた代数的効果の埋め込みで主に必要になるのは第一級の関数であり、スタック操作や限定継続、そして本研究で示したコルーチンを利用する方法と比較して、より多くの言語に適用できる。その一方、 CPS をプログラマに書かせることによる問題もある。 Haskell や Scala、 OCaml などには、モナディックプログラミングを手続き的に記述できる糖衣構文や演算子があるが、多くの言語はこれを持たない。特殊な機能のないプログラム言語では継続を明示的に書く必要があり、“コールバック地獄” などで知られるように煩雑でメンテナンスのしづらいコードになる。我々のコルーチンを用いた埋め込みは、継続の起動が高々 1 回という制限を持ちつつも、特別な糖衣構文などを必要としない、読み書きしやすい埋め込みである。また、 Free モナドを用いた埋め込みは第 6 章で述べたように、多くの関数を生成するため、強力な最適化をおこなう処理系でないと実行時に大きなオーバーヘッドがかかる。我々の埋め込みでは、コルーチンが継続を管理するため、特に処理系による最適化が無い場合は、 Free モナドによる実装と比較して、実行効率のよいプログラムが書ける。

[*1] `https://reactjs.org/docs/hooks-reference.html`
[*2] `https://overreacted.io/algebraic-effects-for-the-rest-of-us/`

第 8 章

結論

本論文では、代数的効果を実装する新たな方法として、コルーチンを用いた埋め込みを提案した。まず、代数的効果を持つ言語とコルーチンを持つ言語を定義し、そして前者から後者への簡潔な構文指向の変換を定義した。他の埋め込み手法で必要となるプログラムの機能と比較して、コルーチンは現在多くの言語が持っているため、代数的効果およびハンドラを広範囲に埋め込むことができる。 Lua 言語と Ruby 言語上にライブラリとして代数的効果を実装することで、我々の埋め込み手法の適用しやすさを示した。このライブラリは GitHub 上に公開されており、一部の興味を持った利用者によって、 JavaScript と Rust に移植された。我々の実装の簡潔さはより多くの利用者、言語、アプリケーションにとってより有利に働くと考えられる。

本研究の中心となる考えは、継続の利用をワンショットに制限することである。我々の埋め込みでは、コルーチンスレッドの残りの計算部分を継続としって利用しており、そしてコルーチンの状態はコピーできない。そのため、継続の実行は高々 1 回に制限される。ワンショットは動的な性質であり、この性質の静的な近似として、線形 (限定) 継続や線形継続渡し方式が従来より研究の対象となっている。本研究における基礎部分をより深く研究し、コルーチンと他のコントロールオペレータとの関係の基礎理論に貢献することが今後の課題である。

謝辞

本研究の進行および本論文の執筆にあたり、様々な助言および指導をしてくださった亀山幸義先生、並びに海野広志先生、およびプログラム論理研究室の皆様に深く感謝致します。計算エフェクトの勉強会を開催してくださった Oleg Kiselyov 先生、および Gordon Plotkin 先生を始めとする発表者、参加者の皆様に深く感謝致します。計算エフェクトに関して実用面等から議論してくださった亀岡亮太様、および株式会社 HERP のエンジニアの皆様に深く感謝致します。

参考文献

[1] Atnos-org. *eff*. https://github.com/atnos-org/eff. Accessed: 2020-01-10.

[2] Andrej Bauer and Matija Pretnar. "Programming with Algebraic Effects and Handlers". In: *Journal of Logical and Algebraic Methods in Programming* 84 (Mar. 2012), DOI: 10.1016/j.jlamp.2014.02.001.

[3] Josh Berdine et al. "Linear Continuation-Passing". In: *Higher-Order and Symbolic Computation* 15 (Sept. 2002), pp. 181–208. DOI: 10.1023/A:1020891112409.

[4] Jonathan Immanuel Brachthäuser and Philipp Schuster. "Effekt: Extensible Algebraic Effects in Scala (Short Paper)". In: *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*. Scala 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 67–72. ISBN: 9781450355292. DOI: 10.1145/3136000.3136007.

[5] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. "Effect Handlers for the Masses". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276481.

[6] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. "Representing Control in the Presence of One-Shot Continuations". In: *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. PLDI 1996. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 99–107. ISBN: 0897917952. DOI: 10.1145/231379.231395.

[7] Olivier Danvy and Andrzej Filinski. "Abstracting Control". In: (1990), pp. 151–160. DOI: 10.1145/91556.91622.

[8] Stephen Dolan, Leo White, and Anil Madhavapeddy. "Multicore OCaml". In: *OCaml Users and Developers Workshop*. 2014.

[9] Stephen Dolan et al. "Concurrent System Programming with Effect Handlers". In: *Trends in Functional Programming*. Apr. 2018, 98–117. ISBN: 9783319897189. DOI: 10.1007/978-3-319-89719-6_6.

[10] Matthias Felleisen. "On the Expressive Power of Programming Languages". In: *Selected Papers from the Symposium on 3rd European Symposium on Programming*. ESOP '90. USA: Elsevier North-Holland, Inc., 1991, pp. 35–75.

[11] Yannick Forster et al. “On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control”. In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017). DOI: 10.1145/3110257. URL: https://doi.org/10.1145/3110257.

[12] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. “A Generalization of Exceptions and Control in ML-like Languages”. In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA ’95. La Jolla, California, USA, 1995, pp. 12–23. ISBN: 0897917197. DOI: 10.1145/224164.224173.

[13] Daniel Hillerström and Sam Lindley. “Shallow Effect Handlers”. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2018, 415–435.

[14] Roshan James and Amr Sabry. “Yield: Mainstream Delimited Continuations”. In: Jan. 2011,

[15] Ohad Kammar, Sam Lindley, and Nicolas Oury. “Handlers in Action”. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2013. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 145–158. ISBN: 9781450323260. DOI: 10.1145/2500365.2500590.

[16] Oleg Kiselyov and Hiromi Ishii. “Freer Monads, More Extensible Effects”. In: Haskell 2015 (2015), pp. 94–105. DOI: 10.1145/2804302.2804319.

[17] Oleg Kiselyov and KC Sivaramakrishnan. “Eff Directly in OCaml”. In: *Electronic Proceedings in Theoretical Computer Science* 285 (Dec. 2018), 23–58. DOI: 10.4204/EPTCS.285.2.

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

[19] Daan Leijen. “Implementing Algebraic Effects in C”. In: *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*. 2017, pp. 339–363. DOI: 10.1007/978-3-319-71237-6_17.

[20] Ana Lúcia de Moura and Roberto Ierusalimschy. “Revisiting Coroutines”. In: *ACM Trans. Program. Lang. Syst.* 31.2 (Feb. 2009). ISSN: 0164-0925. DOI: 10.1145/1462166.1462167.

[21] Gordon Plotkin and John Power. “Algebraic Operations and Generic Effects”. In: *Applied Categorical Structures* 11 (Feb. 2003), 69–94. DOI: 10.1023/A:1023064908962.

[22] Gordon Plotkin and Matija Pretnar. “Handling Algebraic Effects”. In: *Logical Methods in Computer Science* 9 (Dec. 2013), DOI: 10.2168/LMCS-9(4:23)2013.

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

[24] Wouter Swierstra. “Data types á la carte”. In: *Journal of Functional Programming* 18 (July 2008), pp. 423–436. DOI: 10.1017/S0956796808006758.

[25] Typelevel.scala. *cats-effect*. `https://github.com/typelevel/cats-effect`. Accessed: 2020-01-10.