代数的エフェクトとハンドラのための CPS インタプリタと型システム

藤井 舞花, 浅井 健一

お茶の水女子大学

fujii.maika@is.ocha.ac.jp, asai@is.ocha.ac.jp

概 要 例外処理を一般化し継続を明示的に扱える代数的エフェクトとハンドラは様々な言語に導入されている。他にも、代数的エフェクトとハンドラと継続の扱い方が近しいとされる 4 種類の限定継続演算子が存在するが、両者はこれまで全く別の定式化で議論されてきた。これらを統一的に議論するための土台として、代数的エフェクトとハンドラを 4 種類の限定継続演算子と同じ枠組みで CPS インタプリタを再定義し、両者を繋ぐ足掛かりを作る。さらに、4 種類の限定継続演算子ではインタプリタから機械的に抽象機械や型システムが導けており、それらとインタプリタとの間に規則的な繋がりがある。本稿でも新たに定義した代数的エフェクトとハンドラの CPS インタプリタから、別途証明を与えずとも正当性が保証された、型システムや抽象機械を導出する。

1 はじめに

継続とは、プログラムの実行中のある時点における残りの計算のことである。例えば `1 + 2 * 3` という計算があり「`2` と `3` をかける」の時点での残りの計算は「その結果に `1` を足す」である。この「`1` を足す」のような残りの計算を継続と呼ぶ。

本稿ではこの継続を明示的に扱える言語機構である代数的エフェクトとハンドラ [1, 20] を取り上げる。代数的エフェクトとハンドラは例外処理を一般化したものとなっている。従来の例外処理は副作用を発生するとハンドラに事後処理を任せ、オペレーションが発生した時点での継続は捨てられる。一方で代数的エフェクトとハンドラはオペレーションが発生した時点における限定された範囲までの継続 (限定継続) も用いて処理を行うことができる。つまりユーザー自身が自由に定義できる大域ジャンプが可能となる。代数的エフェクトとハンドラのハンドラには深いもの (deep handler) と浅いもの (shallow handler) [16] が存在する。この 2 種類のハンドラは限定継続の扱い方に違いがあるため、同じプログラムでも異なる場所に移動することが可能となる。

他にも継続を扱う演算子として shift/reset [6, 7]、control/prompt [10]、shift0/reset0 [18]、control0/prompt0 [13] の 4 種類の限定継続演算子なども存在する。この 4 つの限定継続演算子の中で shift0/reset0 は deep handler と、control0/prompt0 は shallow handler と限定継続の扱い方に関してとても近い関係性がある [11, 19]。しかし代数的エフェクトとハンドラと 4 種類の限定継続演算子は全く異なる定式化でそれぞれが議論されてきた。これまでの研究では、4 種類の限定継続演算子は、trail [2] とメタ継続 [6] があれば各種の演算子のインタプリタが表現できている [23]。そのため本稿では代数的エフェクトとハンドラのためのインタプリタも、trail やメタ継続を用いて再定義することで、両者を同じ枠組みで統合するための第一歩としていく。また trail をリストではなく高階関数を用いて実装することによって、高階関数の形で shallow handler のインタプリタを定義することを可能にした。高階関数の形で定義できたことによって、trail の型が変化するようなプログラムの例でも型をつけることができるようになる。今回は 4 種類の限定継続演算子[1]に合わ

[1] マルチプロンプトや [8, 9] 多段の CPS 階層 [15] を扱わない基本的なものの場合

せたため、例外に名前を付けることができないという制約を設けているが、条件文さえ追加すればdeep handler については名前付きオペレーションが模倣可能となる。

4 種類の限定継続演算子は正当性が保証された非関数化などの変換を施せば、インタプリタから抽象機械や仮想機械 [12] が導出できている。正当性が保証された変換を使うことで、別途証明を与えることなく抽象機械や仮想機械が正しいことが保証されている。さらに型システム [24] もインタプリタから機械的に導けており、停止性や健全性が示された。それをもとにして 代数的エフェクトとハンドラも 4 種類の限定継続演算子と同じ枠組みでインタプリタで定義することで、同様にしてインタプリタから機械的に仮想機械や型システムを導出し別途証明を与えずとも正当性が保証された代数的エフェクトとハンドラの直接実装への先駆けとしていく。

本稿の貢献は以下のようにまとめられる。

- 代数的エフェクトとハンドラのための新たな CPS インタプリタを定義する。これを 4 種類の限定継続演算子の実装に必要となる trail とメタ継続の概念のみで実現する。そのため代数的エフェクトとハンドラと 4 種類の限定継続演算子を統一的に議論できる土台を提供する。
- CPS インタプリタから代数的エフェクトとハンドラのための型システムを導出する。この型システムに基づく言語と、それに対する CPS インタプリタを定理証明系言語 Agda で実装し、停止性や健全性も示している。
- CPS インタプリタから半機械的に導出した抽象機械を示す。

Agda で実装したコードは `http://pllab.is.ocha.ac.jp/~asai/jpapers/ppl/ppl22.agda` で入手可能である。

論文の構成

次節で代数的エフェクトとハンドラについて紹介をした後、3 節でインタプリタを紹介する。4 節では新たに定義したインタプリタとこれまでに提唱されたインタプリタを比較する。そして本研究のインタプリタから導出できた抽象機械を 5 節で、型システムを 6 節で示す。最後に 7 節で関連研究について議論し、8 節で結論を述べる。

2 代数的エフェクトとハンドラ

代数的エフェクトとハンドラは副作用を起こす演算子とそれを処理する演算子からなる。通常の例外処理とは異なり、オペレーションを起こした時点におけるハンドラで限定された継続を副作用時に扱うことができる。本稿では *try ... with* をハンドラとし、*call* とすることでオペレーション呼び出しをする。具体的に `try e1 with (x; k) -> e2` と記述した式は、もし `e1` で `call(a)` とオペレーション呼び出しが行われると、`a` が `x` に、オペレーション呼び出しをした地点から `try` と `with` で囲まれた部分までの継続が `k` に代入され、`e2` を実行する。

代数的エフェクトとハンドラには deep と shallow の 2 種類のハンドラが存在するが、両者の違いが出る例として以下の式について考える。なお、本稿では計算の評価を値呼び、かつ左から行うものとする。

```
1 + try (try 10 + call(3) * call(4)
         with (x; k) -> k x - 2)
         - 5
    with (y; g) -> y * 2
```

この式では、外側のハンドラの中でオペレーション呼び出しが起きると、これまでの継続 (g) を捨て引数を 2 倍する作用が、内側のハンドラの中でオペレーション呼び出しが起きると、これまでの

継続 (k) を呼び出してから 2 を引く作用が施される。この内側のハンドラのようにオペレーションの処理に k を用いることで残りの継続を扱うことができている。この式を用いた 2 種類のハンドラの詳細については 2.1 節 (deep) と 2.2 節 (shallow) で述べる。

2.1 deep handler

はじめに上記の式のハンドラが deep なハンドラであった場合を説明する。まず内側のハンドラ内にある call(3) でオペレーションが呼び出され、x には 3 が、k には内側のハンドラで限定された継続 try 10 + [.] * call(4) with (x; k) -> k x - 2 が代入される。deep なハンドラで捕捉されたため限定継続に同じハンドラが入る。そして計算が内側のハンドラの制御部分 k x - 2 へ移り、以下のように計算が進む。

```
1 + try ((try 10 + 3 * call(4) with (x; k) -> k x - 2)
         - 2)
        - 5
    with (y; g) -> y * 2
```

次にまた内側のハンドラ内にある call(4) でオペレーション呼び出しが行われる。同様にして、x には 4 が、k には deep なハンドラで限定された継続 try 10 + 3 * [.] with (x; k) -> k x - 2 が代入される。計算も下記の式へと評価される。

```
1 + try (((try 10 + 3 * 4 with (x; k) -> k x - 2)
          - 2)
         - 2)
        - 5
    with (y; g) -> y * 2
```

すると内側のハンドラ内の計算 10 + 3 * 4 ではオペレーションが呼び出されずに正常終了するためハンドラ外の計算へと進み、外側にあったハンドラ内でもオペレーションが呼び出されることなく正常終了し、結果として 14 と評価される。

2.2 shallow handler

次にハンドラが shallow なハンドラであった場合について考える。deep な場合と同様にして内側のハンドラ内にある call(3) でオペレーション呼び出しが行われる。x に 3 が、k には内側のハンドラで限定された継続が代入されるが、shallow なハンドラで限定された継続はハンドラ自分自身は囲まれないため 10 + [.] * call(4) となる。そして計算は下記のようになる。

```
1 + try ((10 + 3 * call(4))
         - 2)
        - 5
    with (y; g) -> y * 2
```

ここで限定継続 k が呼び出されたときの継続 ([.] - 2) - 5 に注目する。この継続は「呼び出し文脈 (invocation context)」と呼ばれている。次に call(4) が呼び出されているが、deep の場合とは違って外側にあったハンドラ内で起こっている。そのため y に 4 が、g にハンドラで限定された継続 ((10 + 3 * [.]) - 2) - 5 が代入される。このように deep なハンドラとは違い shallow なハンドラは k の呼び出し文脈であった ([.] - 2) - 5 まで限定継続として g に代入する。そしてハンドラの制御部分 y * 2 へ移り、以下のように計算が評価される。

```
1 + 4 * 2
```

よって計算結果は 9 と評価される。

ハンドラの中で一度しかオペレーションが呼ばれないなら deep と shallow なハンドラで計算結果に違いは現れないが、このように同じハンドラの中で 2 度以上オペレーションが呼び出されていると違いが現れる。

2.3 オペレーションに名前を付ける

本稿では 4 種類の限定継続演算子に合わせたため、上記で紹介した代数的エフェクトとハンドラはオペレーションが呼び出されたら直近のハンドラで処理を担う構文となっているが、より一般的にはオペレーションに名前を与え複数のハンドラが存在する際にどこのハンドラにあるオペレーションを用いるか自由に定義できるようにする。例えば以下の式を考える。

```
1 + try (try 10 + call2(4) with {call1(x; k) -> k x - 2})
        - 5
    with {call2(y; g) -> g y + 2;
          call3(y; g) -> y * 2}
```

どのオペレーションをハンドルするかを示すために `with` 以下では引数と限定継続の変数だけでなくオペレーション名も指定する。外側のハンドラのように 2 つ以上のオペレーションを定義することも可能である。この式の計算は、まず内側のハンドラの中でオペレーション `call2` が呼び出されているが、内側のハンドラには `call2` は定義されていない。しかし外側のハンドラを見てみると `call2` の作用が定義されているため、外側のハンドラの処理に飛ぶことができる。このとき `y` に 4 が、`g` には外側のハンドラ内の限定継続が代入される。つまり deep handler の場合は

```
g = try (try 10 + [.] with {call1(x; k) -> k x - 2})
        - 5
    with {call2(y; g) -> g y + 2;
          call3(y; g) -> y * 2}
```

となり、 shallow handler の場合は

```
g = (try 10 + [.] with {call1(x; k) -> k x - 2})
    - 5
```

と限定継続が切り取られる。どちらの場合も内側のハンドラは外側のハンドラ内の限定継続の一部として残っている。そして式は以下のように計算が進み、限定継続を呼び出し 2 を足す作用を施すことができる。

```
1 + (g y + 2)
```

このようにオペレーションに名前を与えると直近ではないそれよりも外側のハンドラにあるオペレーションを呼び出すことも可能にする。

本稿ではハンドラが限定継続を切り取る際の挙動における基本的な体系を示したいため名前付きオペレーションに対応しない仕様としたが、deep handler であればオペレーションに名前を与える代わりに条件文を使って模倣できることを紹介する。手法として、ハンドラの処理を `if x = a then ... else k call(x)` と記述する。ここで `x` はオペレーション呼び出しの引数が、`k` には限定継続が代入されており、`a` は任意の値となる。`x = a` が真であればオペレーションの名前が一致したこととし、このハンドラで行いたい処理を記述する。目的のハンドラでなければ (オペレーショ

ンの名前が一致しなければ) else 部分を行う。else 部分では限定継続 k の引数として call(x) を渡すことで、これまでの限定継続を保持したまま一つ外側のハンドラに飛ぶようになっている。先程例に出した式の場合は以下のように模倣する。

```
1 + try (try 10 + call(4) with (x; k) -> if x = a1 then k x - 2 else k call(x))
        - 5
    with (y; g) -> if y = 4 then g y + 2
                   else if y = a2 then y * 2
                   else g call(y)
```

ここで a1 と a2 は 4 ではない任意の値とする。計算の進め方として、まず内側のハンドラでオペレーションが呼び出されているため

```
x = 4
k = try 10 + [.] with (x; k) -> if x = a1 then k x - 2 else k call(x)
```

としてハンドラの処理に移り、式は以下のように進む。

```
1 + try (if x = a1 then k x - 2 else k call(x))
        - 5
    with (y; g) -> if y = 4 then g y + 2
                   else if y = a2 then y * 2 else g call(y)
```

条件分岐において x = a1 は偽であるため else 部分を評価する。するとまた call(x) が起こり外側にあったハンドラの処理に移る。このとき

```
y = x = 4
g = try (k [.]) - 5 with (y; g) -> if y = 4 then g y + 2
                                   else if y = a2 then y * 2 else g call(y)
```

となり、式は以下のようになる。

```
  1 + (if y = 4 then g y + 2 else if y = a2 then y * 2 else g call(y))
```

よって y = 4 であるためその then 部分の計算 g y + 2 を行うことができる。そして呼び出された限定継続 g は、中の k も呼び出せば

```
g = try (try 10 + [.] with (x; k) -> if x = a1 then k x - 2 else k call(x))
        - 5
    with (y; g) -> if y = 4 then g y + 2
                   else if y = a2 then y * 2 else g call(y)
```

となり、名前付きオペレーションに対応した場合の限定継続と本質的に同じになっている。

shallow handler の場合でも条件文を用いて名前付きオペレーションを模倣することを試みたが、shallow handler は限定継続を元々囲んでいたハンドラがなくなるという性質があるため、k call(a) としても残るべきハンドラが消えてしまい上記の方法では上手くいかない。

$$
\begin{array}{rrcl}
\text{項} & e & ::= & x \mid \underline{\lambda} x.\, e \mid e_1 \mathbin{\underline{@}} e_2 \mid \underline{try}\, e_1\, \underline{with}\, (x; k).\, e_2 \mid \underline{call}(e) \\
\\
\text{値} & v & = & (v \to c \to t \to m \to v) \\
\text{継続} & c & = & v \to t \to m \to v \\
\text{trail} & t & = & \bullet \ + \ (v \to t \to m \to v) \\
\text{ハンドラ} & h & = & (v \to v \to c \to t \to m \to v) \\
\text{メタ継続} & m & = & (c \times t \times h)\ list
\end{array}
$$

$$
\begin{array}{rlcl}
\text{初期継続} & id_c\, v\, id_t\, [] & = & v \\
 & id_c\, v\, id_t\, ((c, t, h) :: m) & = & c\, v\, t\, m \\
 & id_c\, v\ \ c\ \ m & = & c\, v\, id_t\, m \\
\text{初期 trail} & id_t & = & Bullet
\end{array}
$$

$$
\begin{array}{rcl}
(::) : c \to t & \to & t \\
c :: id_t & = & c \\
c :: c' & = & \lambda v.\, \lambda t'.\, c\, v\, (c' :: t')
\end{array}
$$

$$
\begin{array}{rcl}
(@) : t \to t & \to & t \\
id_t \mathbin{@} t & = & t \\
c \mathbin{@} t & = & c :: t
\end{array}
$$

図 1. 構文、インタプリタで使用する型や初期継続、初期 trail、trail の合成のための関数

3　代数的エフェクトとハンドラの CPS インタプリタ

先述の通り本稿のインタプリタは 4 種類の限定継続演算子のインタプリタ [23, 12] を参考にしており、評価関数の引数に trail やメタ継続というデータも渡す形式として CPS インタプリタを定義している。trail は呼び出し文脈の追跡 [2] をするために、メタ継続 [6] はハンドラの内側と外側の継続 (計算) を区切るために使用する。trail は control や control0 で用いられていた概念であり、今回は shallow handler のインタプリタで用いられる。

項やインタプリタで使用する値の型は図 1 に定義する。この言語は λ 計算にハンドラとオペレーション呼び出しを追加したものである。値の型 v は再帰型となる。継続の型 c やハンドラ h の型は関数型として定義される。trail t の型は空であることを表す $\bullet$ 型であるか、継続 (呼び出し文脈) を関数合成したものを表す関数型 (再帰型) のいずれかになる。そしてメタ継続 m は 継続 c と trail t、ハンドラ h を要素に持つリスト型として定義する。初期継続は id_c として定義され、継続が空になったら trail、メタ継続の順番に中身の実行に移る関数となっている。初期 trail id_t は $\bullet$ 型の値を $Bullet$ として空の trail を表す。また $(::)$ は継続 c と trail t の、$(@)$ は trail t 同士の関数合成を行う。

図 2 に代数的エフェクトとハンドラの CPS インタプリタを示す。継続 c に加え、trail t とメタ継続 m も引数として受け取る。呼び出し文脈を扱うために trail を、CPS インタプリタにさらにもう一度 CPS 変換を施した 2CPS となるためメタ継続を使用する。そのため trail やメタ継続は代数的エフェクトとハンドラを実装するためのデータである。まず λ 項の解釈は、trail t とメタ継続 m を η-簡約すれば通常の CPS インタプリタと変わらない。trail やメタ継続は λ 計算の評価時にはただそのまま引き渡すだけになっている。また 2.3 節で説明した通り以降の節で条件文項が必要になる場合があるが、λ 項同様 trail とメタ継続を η-簡約すれば従来のインタプリタと変わらないため本稿では条件文の CPS インタプリタは示さない。

ハンドラ $\underline{try}\, e_1\, \underline{with}\, (x; k).\, e_2$ はまず e_1 を評価する。その際、継続と trail は空のものをセットし、メタ継続に現在の継続 c、trail t、ハンドラ h を新たに追加する。メタ継続に $\underline{try}\ \underline{with}$ の外側

$$
\begin{array}{rcl}
\mathcal{E}\,[\![x]\!]\,\rho\,c & = & \lambda t.\,\lambda m.\,c\,\rho(x)\,t\,m \\
\mathcal{E}\,[\![\underline{\lambda} x.\,e]\!]\,\rho\,c & = & \lambda t.\,\lambda m.\,c\,(\lambda v.\,\lambda c'.\,\lambda t'.\,\lambda m'.\,\mathcal{E}\,[\![e]\!]\,\rho[v/x]\,c'\,t'\,m')\,t\,m \\
\mathcal{E}\,[\![e_1\,\underline{@}\,e_2]\!]\,\rho\,c & = & \lambda t.\,\lambda m. \\
 & & \quad \mathcal{E}\,[\![e_1]\!]\,\rho\,(\lambda v_1.\,\lambda t_1.\,\lambda m_1. \\
 & & \quad \mathcal{E}\,[\![e_2]\!]\,\rho\,(\lambda v_2.\,\lambda t_2.\,\lambda m_2.\,v_1\,v_2\,c\,t_2\,m_2)\,t_1\,m_1)\ t\ m \\
 & & \\
\mathcal{E}\,[\![\underline{try}\,e_1\,\underline{with}\,(x;k).\,e_2]\!]\,\rho\,c & = & \lambda t.\,\lambda m.\,\mathcal{E}\,[\![e_1]\!]\,\rho\,id_c\,id_t\,((c,t,h)::m) \\
 & & \text{where } h = \lambda v.\,\lambda v_c.\,\lambda c'.\,\lambda t'.\,\lambda m'.\,\mathcal{E}\,[\![e_2]\!]\,\rho[v/x][v_c/k]\,c'\,t'\,m' \\
\mathcal{E}\,[\![\underline{call}(e)]\!]\,\rho\,c & = & \lambda t.\,\lambda m.\,\mathcal{E}\,[\![e]\!]\,\rho\,(\lambda v.\,\lambda t''.\,\lambda((c_0,t_0,h)::m_0).\,h\,v\,v_c\,c_0\,t_0\,m_0)\,t\,m \\
\textit{deep handler のとき} & & \text{where } v_c = \lambda v'.\,\lambda c'.\,\lambda t'.\,\lambda m'.\,c\,v'\ t''\ ((c',t',h)::m') \\
\textit{shallow handler のとき} & & \text{where } v_c = \lambda v'.\,\lambda c'.\,\lambda t'.\,\lambda m'.\,c\,v'\ (t''\,@\,c'::t')\ m'
\end{array}
$$

図 2. 代数的エフェクトとハンドラのための CPS インタプリタ

の継続 c を保持することで、e_1 の評価中にオペレーション呼び出しが現れても $\underline{try}$ $\underline{with}$ の内側の継続のみが扱えるようになっている。つまりメタ継続に要素を追加することが、ハンドラで限定する範囲の区切りとなっている。そのため $\underline{try}$ $\underline{with}$ を評価するたびにメタ継続の要素が増えていくことになる。そしてメタ継続に追加されるハンドラ h は高階関数として定義している。「値 v (引数値)、限定継続を表す値 v_c、継続 c'、trail t'、メタ継続 m' を受け取り、そのもとで e_2 を実行する」というハンドラの処理を表す関数となる。ここで e_2 を評価する際の環境 ρ に x が v、k が v_c である情報が追加され、引数値 x と切り取られた限定継続 k を使うことが可能となる。

次に $\underline{call}(e)$ は、「まず e を評価しその後ハンドラの処理に移る」という評価になる。そのため実際のインタプリタでは e の評価時の継続が「ハンドラの処理に移る」となっている。具体的に、この継続の定義は e を評価して得られた値 v、評価後の trail t''、メタ継続が渡される。このときのメタ継続は $(c_0,t_0,h)::m_0$ と必ず要素が一つ入ったリストとなる。これはオペレーション呼び出しは必ずハンドラの中で行われるという仮定に対応し、メタ継続の最初の要素を取得することで直近のハンドラにアクセスしている。そしてハンドラ h に引数値 v、オペレーション呼び出しが行われた時点における限定継続を表す関数 v_c、ハンドラの外側の継続 c_0、trail t_0、メタ継続 m_0 を渡す。ここで v_c は限定継続に渡す値 v' とこの限定継続が呼び出されたときの継続 (呼び出し文脈) c'、t'、m' を受け取り、$\underline{call}(e)$ が評価された時点での継続 c 及び trail t'' を実行する高階関数である。つまり限定継続 v_c は現在の継続 c と trail t'' のみ捕捉し、そのときのメタ継続 $(c_0,t_0,h)::m_0$ は含まないため、ハンドラの内側の継続のみが限定継続として扱えるようになっている。そしてこの v_c の関数の本体部分に deep handler と shallow handler の違いが現れている。実際に、c を実行するときの trail とメタ継続 (図のハイライトされた部分) に注目してみよう。

- deep handler の場合
  c は引数として v'、trail として t''、メタ継続として m' に $(c',\ t',\ h)$ を新たに追加したものを受け取ったもとで実行される。メタ継続 m' に要素が追加されることで継続 c と c' が区切られているため、継続 c の計算にハンドラが入っていることを表している。
- shallow handler の場合
  c は引数として v'、trail として t'' と c'、t' が合成されたもの、メタ継続として要素の追加なしに m' のみを受け取ったもとで実行される。このように呼び出し文脈を trail に保持することで、c の中でオペレーション呼び出しが現れた際に呼び出し文脈 (trail) も限定継続として捕捉されるようになり、shallow handler によって限定された継続にはハンドラが囲まれないことを表現している。

trail は shallow handler の場合のみ合成されており、deep handler の場合では trail は実質的には

$$
\begin{aligned}
\mathcal{E}\,[\![\underline{try}\,e_1\,\underline{with_D}\,(x;k).\,e_2]\!]\,\rho\,c &= \lambda t.\,\lambda m.\,\mathcal{E}\,[\![e_1]\!]\,\rho\,id_c\,id_t\,((c,t,h)::m)\\
&\quad \text{where } h = \lambda v.\,\lambda c_1.\,\lambda t_1.\,\lambda((c_0,t_0,h)::m_0).\,\mathcal{E}\,[\![e_2]\!]\,\rho[v/x][v_c/k]\,c_0\,t_0\,m_0\\
&\quad \text{where } v_c = \lambda v_2.\,\lambda c_2.\,\lambda t_2.\,\lambda m_2.\,c_1\,v_2\,t_1\,((c_2,t_2,h)::m_2)\\
\mathcal{E}\,[\![\underline{try}\,e_1\,\underline{with_S}\,(x;k).\,e_2]\!]\,\rho\,c &= \lambda t.\,\lambda m.\,\mathcal{E}\,[\![e_1]\!]\,\rho\,id_c\,id_t\,((c,t,h)::m)\\
&\quad \text{where } h = \lambda v.\,\lambda c_1.\,\lambda t_1.\,\lambda((c_0,t_0,h)::m_0).\,\mathcal{E}\,[\![e_2]\!]\,\rho[v/x][v_c/k]\,c_0\,t_0\,m_0\\
&\quad \text{where } v_c = \lambda v_2.\,\lambda c_2.\,\lambda t_2.\,\lambda m_2.\,c_1\,v_2\,(t_1 \mathbin{@} c_2::t_2)\,m_2\\
\mathcal{E}\,[\![\underline{call}(e)]\!]\,\rho\,c &= \lambda t.\,\lambda m.\\
&\qquad \mathcal{E}\,[\![e]\!]\,\rho\,(\lambda v.\,\lambda t'.\,\lambda((c_0,t_0,h)::m_0).\,h\,v\,c\,t'\,((c_0,t_0,h)::m_0))\\
&\qquad t\quad m
\end{aligned}
$$

図 3. ハンドラで deep/shallow を区別するインタプリタ

使われず常に同じ値を引き渡すだけになっている。つまり trail は shallow なハンドラの挙動を表現するためだけに使われる。

4 種類の限定継続演算子のインタプリタとは、メタ継続にハンドラ h が追加されていること、$\underline{call}(e)$ の引数 e を評価した後に限定継続を切り取る操作をすることが異なるだけで、本質的な継続の扱い方は同様の挙動で定義することができている。

3.1 ハンドラで deep/shallow を区別するインタプリタ

上記で説明したインタプリタは 4 種類の限定継続演算子の CPS インタプリタを参考に作成したため、オペレーション呼び出しを評価するタイミングで限定継続の意味を与えるようになっている。つまり囲んでいたハンドラが deep か shallow なのかは $\underline{call}(e)$ の評価時に分かるということである。一方で、ハンドラ側でも切り取られる限定継続の意味を与えるインタプリタも提唱したい。提唱するインタプリタは前節のものと独立であり、片方がより表現力が高いという訳ではない。

新たなインタプリタを図 3 に示す。図 2 のインタプリタとの違いは、e_2 の実行環境に追加する k の値 v_c をどこで実際に定義するかである。図 2 のインタプリタでは $\underline{try}\,e_1\,\underline{with}\,(x;k).\,e_2$ の評価時に現れる v_c はただ単に何らかの値であるだけで、$\underline{call}(e)$ の評価時にそれぞれの限定継続 v_c を定めていた。一方で図 3 では $\underline{try}\ \underline{with}$ に D が付いているものを deep handler の項、S が shallow handler の項として、ハンドラ h の中で v_c の意味も与えることにした。

図 3 のインタプリタを詳しく見てみる。deep と shallow どちらのハンドラであっても e_1 の評価で空の継続と trail をセットし、メタ継続に要素を追加していることはこれまでと同様であり、ハンドラ h の定義が少し変わる。まず関数の引数として値 v と $\underline{call}(e)$ が評価された時点での継続 c_1、trail t_1、メタ継続を受け取る。そのときのメタ継続は $(c_0,\ t_0,\ h)::m_0$ の形になっている (オペレーション呼び出しは必ずハンドラで囲まれた状態で起こるため)。そして e_2 はメタ継続に積まれていた一つ外側の継続 c_0、trail t_0、メタ継続 m_0 のもとで実行される。さらに e_2 の環境として持つ k に対応する値 v_c も定義する。v_c の定義は、図 2 の deep の場合と shallow の場合のそれぞれと同様の定義を与えれば良い。$\underline{call}(e)$ の評価は、e を評価後直近のハンドラ h にアクセスをし、h にそのときの継続 c と trail t、メタ継続を渡すだけで良くなっている。ここで限定継続に自分自身と同じハンドラが入るという deep handler の性質上ハンドラ h に同じ h を渡しており、再帰的になっている。この再帰が型システムを導出する際に関係してくるが詳しくは 6 節で述べる。

3.2 ハンドラの非関数化

図 2 と図 3 のインタプリタが同等の挙動をすることを確かめるため、高階関数として定義されているハンドラの処理 h を非関数化 [21, 22] する。非関数化とは関数の body 部分で自由変数となっ

$$
\begin{array}{rcl}
\mathcal{E}\,[\![\underline{try}\,e_1\,\underline{with}\,(x;\!\!\cdot\,k).\,e_2]\!]\,\rho\,c & = & \lambda t.\,\lambda m.\,\mathcal{E}\,[\![e_1]\!]\,\rho\,id_c\,id_t\,((c,t,(x,\,k,\,e_2,\,\rho))::m) \\
\mathcal{E}\,[\![\underline{call}(e)]\!]\,\rho\,c & = & \lambda t.\,\lambda m.\,\mathcal{E}\,[\![e]\!]\,\rho\,(\lambda v.\,\lambda t''.\,\lambda((c_0,t_0,(x,\,k,\,e_0,\,\rho_0))::m_0). \\
 & & \qquad\qquad \mathcal{E}\,[\![e_0]\!]\,\rho_0[v/x][v_c/k]\,c_0\,t_0\,m_0)\;t\;m \\
\textit{deep handler のとき} & & \text{where } v_c = \lambda v'.\,\lambda c'.\,\lambda t'.\,\lambda m'.\,c\,v'\,t''\,((c',t',(x,\,k,\,e_0,\,\rho_0))::m') \\
\textit{shallow handler のとき} & & \text{where } v_c = \lambda v'.\,\lambda c'.\,\lambda t'.\,\lambda m'.\,c\,v'\,(t''\mathbin{@}c'::t')\,m' \\
\hline
\mathcal{E}\,[\![\underline{try}\,e_1\,\underline{with}_D\,(x;\!\!\cdot\,k).\,e_2]\!]\,\rho\,c & = & \lambda t.\,\lambda m.\,\mathcal{E}\,[\![e_1]\!]\,\rho\,id_c\,id_t\,((c,t,(x,\,k,\,e_2,\,\rho)_D)::m) \\
\mathcal{E}\,[\![\underline{try}\,e_1\,\underline{with}_S\,(x;\!\!\cdot\,k).\,e_2]\!]\,\rho\,c & = & \lambda t.\,\lambda m.\,\mathcal{E}\,[\![e_1]\!]\,\rho\,id_c\,id_t\,((c,t,(x,\,k,\,e_2,\,\rho)_S)::m) \\
\mathcal{E}\,[\![\underline{call}(e)]\!]\,\rho\,c & = & \lambda t.\,\lambda m.\,\mathcal{E}\,[\![e]\!]\,\rho\,(\lambda v.\,\lambda t''.\,\lambda((c_0,t_0,h)::m_0). \\
 & & \qquad\qquad \mathcal{E}\,[\![e_0]\!]\,\rho_0[v/x][v_c/k]\,c_0\,t_0\,m_0)\;t\;m \\
 & & \text{if } h = (x,\,k,\,e_0,\,\rho_0)_D \\
 & & \text{where } v_c = \lambda v'.\,\lambda c'.\,\lambda t'.\,\lambda m'.\,c\,v'\,t''\,((c',t',h)::m') \\
 & & \text{if } h = (x,\,k,\,e_0,\,\rho_0)_S \\
 & & \text{where } v_c = \lambda v'.\,\lambda c'.\,\lambda t'.\,\lambda m'.\,c\,v'\,(t''\mathbin{@}c'::t')\,m'
\end{array}
$$

図 4. 図 2 と 図 3 の型 h を非関数化したインタプリタ

ているものを保持するコンストラクタを定義し、関数の代わりにそのコンストラクタが必要になったタイミングで元々関数の本体部分で行っていた挙動をするような変換である。

非関数化を行ったインタプリタを図 4 に示す。上段は図 2 のインタプリタを下段は図 3 のインタプリタを非関数化したものである。$\underline{try}\,e_1\,\underline{with}\,(x;\!\!\cdot\,k).\,e_2$ では、ハンドラ内の式 e_1 の評価でメタ継続に追加するハンドラは関数ではなく、ハンドラの処理に必要な情報を保持したコンストラクタ $(x,\,k,\,e_2,\,\rho)$ となる。下段は構文として deep handler と shallow handler を分けているためこのコンストラクタに deep か shallow の区別をつけているが、本質的には上下段の挙動は同じである。そして $\underline{call}(e)$ では e を評価した後の継続として、そのときのメタ継続 $(c_0,t_0,h)::m_0$ を展開し、ハンドラ h の情報 $(x,\,k,\,e_0,\,\rho_0)$ を使う。上段の場合は e_0 を c_0、t_0、m_0 のもとで評価するようにし、そのときの環境に追加する限定継続の関数は deep か shallow かの場合によってそれぞれ定義する。下段の場合も e_0 を c_0、t_0、m_0 のもとで評価するようにし、そのときの限定継続はハンドラのコンストラクタに付随された D か S かによって適当な限定継続の関数を定義している。ゆえに、deep handler だった場合の上段の挙動と、下段の $\underline{try}\,\underline{with}_D$ の挙動は同じとみなせ、shallow handler の場合も同様である。

したがって 3.1 節の議論の結論を以下にまとめる。

- 限定継続の具体的な定義を $\underline{try}\;\underline{with}$ と $\underline{call}$ のどちらのタイミングで行っても実装できる。
- 同じインタプリタで deep handler と shallow handler を共にサポートしたい場合は図 3 のような意味論になるが、ハンドラの定義が再帰的となる。

4 既存のインタプリタとの比較

本節では Hillerström らの既存の CPS インタプリタ [14] と本稿のインタプリタが同様の挙動をすることをインフォーマルな形で議論する。代数的エフェクトとハンドラのインタプリタの要となる限定継続を表す関数の定義について、本稿のインタプリタは高階関数で定義したのに対し、Hillerström らの既存のインタプリタはリストの構造を用いて定義している。正しくは deep handler については Hillerström らも高階関数でも定義しているが、deep handler と shallow handler の両方を実装するものはリスト構造でのみ示しているため本節ではリスト構造を用いたインタプリタで詳しい比較を行う。

Hillerström らのインタプリタでは正常終了したときの継続 (pure continuations) c とハンドラ h のペアを要素としたリストをメタ継続 m としている。本稿では継続 c が正常終了したときの継続に相当する。そのため後の説明ではどちらのインタプリタであってもこれらの記号を c、h、m に統一する。ここで Hillerström らのインタプリタのメタ継続は $(c,\ h) :: m$ と定義されているが、先頭の要素が現在の継続とハンドラを表しており、ペア同士の c を h で囲むこととしている。対して本稿のインタプリタは継続 c とメタ継続 m を別々の引数としてインタプリタに渡している。そのため現在の継続はインタプリタの引数である c で、そのときのメタ継続が $(c',\ h) :: m$ であった場合、h は c を囲むハンドラ、c' は h の一つ外側の継続であることに注意する。

Hillerström らの CPS インタプリタではハンドラ $\underline{try}\ e\ \underline{with}\ H$ (H にはオペレーションが連なっている。) は、そのときのメタ継続 m に初期継続を表す空リストと H を評価して得られたハンドラ h のペアを追加した新しいメタ継続 $([], h) :: m$ のもとで e を評価する。Hillerström らのインタプリタもハンドラが評価されるごとに継続とハンドラをメタ継続に追加していき、ハンドラで限定される継続の範囲を区切っている。オペレーション呼び出しの評価では、そのときのメタ継続 $(c,\ h) :: m$ を $rk = (c,\ h) :: []$ と残りのリスト m で分ける。これはオペレーション呼び出しが起こった時点における直近のハンドラ h 内の継続 c とその外側の継続を分けているということである。そして h に引数の値やオペレーションの名前、rk、m を渡してハンドラの処理に移る。h に渡った後 rk を用いて限定継続を表す関数が作られるが、本稿と Hillerström らのインタプリタではこの関数の定義の仕方が異なる。そのためこの関数の定義に着目し、deep handler と shallow handler に分けて二つのインタプリタの挙動を確認していく。

4.1 deep handler の場合

Hillerström らのインタプリタは名前付きオペレーションに対応しているが、本稿のインタプリタはオペレーション呼び出しが起きたら必ず直近のハンドラで処理を行う仕様になっている。しかし 2.3 節で示した通り deep handler は名前付きオペレーションを模倣することが可能であるため、直近のハンドラで処理する場合と名前付きオペレーションの場合に分けて説明する。

4.1.1 オペレーション呼び出しが起こったとき、直近のハンドラで処理する場合

まず直近のハンドラにあるオペレーションの名前が目的のオペレーションと一致した場合を考える。Hillerström らのインタプリタは $rk = (c,\ h) :: []$ を用いて限定継続を表す関数が以下のように定義される。

$$\lambda\ v\ ((c',\ h') :: m').\ \mathbf{app}\ ((c,\ h) :: (c',\ h') :: m')\ v$$

この関数の v は限定継続に渡される引数、$((c',\ h') :: m')$ は限定継続が呼び出されたときのメタ継続である。**app** は Hillerström らの CPS インタプリタ特有の計算構文であるが、メタ継続の最初の要素から順に継続を適用していくものであり、この限定継続の関数は「継続 c から実行し、この継続 c をハンドラ h が囲んでいること」を表している。本稿で定義される deep handler の限定継続の関数は

$$\lambda v.\ \lambda c'.\ \lambda t'.\ \lambda m'.\ c\ v\ t\ ((c',\ t',\ h) :: m')$$

となっているが、これも「継続 c を実行し、このときハンドラ h が囲んでいること」を意味しており、これら二つの限定継続を表す関数が同等の挙動であることが確認できる。

4.1.2 オペレーション呼び出しが起こったとき、直近のハンドラで処理する場合 (名前付きオペレーションの場合)

次に目的のオペレーションが存在するハンドラは内側から n 個目のハンドラで、目的のハンドラまでに別の $n-1$ 個のハンドラにも囲まれていると言う状況を考える。内側から i 番目のハンドラの中の継続を c_i、ハンドラでの処理を h_i とする。

Hillerström らの CPS インタプリタは名前付きのオペレーションに対応しているため、この状況を考慮したインタプリタとして定義されている。前節と同様にして rk と m を用いるが、ここで rk の中身を $rk = (c_1,\ h_1) :: []$ と置き換える。また n 個以上のハンドラで囲まれている状況であるため m も $(c_2,\ h_2) :: ... :: (c_{n-1},\ h_{n-1}) :: (c_n,\ h_n) :: m_{out}$ とする。h_1 には名前が一致するオペレーションが定義されていないため限定継続の関数は作らず、m の先頭の要素である $(c_2,\ h_2)$ を rk の先頭に追加して一つ外側のハンドラ h_2 を見ていくようにする。そのため目的のハンドラ n に達するまでにこの動作を繰り返していき $rk = (c_n,\ h_n) :: (c_{n-1},\ h_{n-1}) :: ... :: (c_2,\ h_2) :: (c_1,\ h_1) :: []$ と増幅する。そして h_n には目的のオペレーションがあるため rk を用いて限定継続を表す関数を作る。しかし c_1 から処理するような関数にしたいため、リスト rk の順番を反転させ

$$\lambda\ v\ ((c',\ h') :: m').\ \mathbf{app}\ ((c_1,\ h_1) :: (c_2,\ h_2) :: ... :: (c_{n-1},\ h_{n-1}) :: (c_n,\ h_n) :: (c',\ h') :: m')\ v$$

として限定継続の関数を定義する。

本稿のインタプリタでは 2.3 節で示したように、ハンドラの処理部分を `if x = a then ... else k call(x)` として模倣した場合で考える。最初にオペレーション呼び出しが起こったときの継続が c_1、メタ継続 m は $(c_2,\ h_1) :: (c_3,\ h_2) :: ... :: (c_n,\ h_{n-1}) :: (c_{out},\ h_n) :: m_{out}$ [2]となっている状況から説明する。まずメタ継続の最初の要素 $(c_2,\ h_1)$ にアクセスし、h_1 を実行するがそのとき限定継続の関数 v_{c_1} や一つ外側の継続 c_2、残りのメタ継続を渡す。ここで $v_{c_1} = \lambda v'.\,\lambda c'.\,\lambda m'.\,c_1\,v'\,((c', h_1) :: m')$ である。そして h_1 ではオペレーションの名前が一致ないこととなるため `else` 部分を行い一つ外側のハンドラ h_2 に飛ぶ。このとき切り取られる限定継続は $c_2' = \lambda v.\,\lambda m.\,v_{c_1}\,v\,c_2\,m$ (継続 c_2 のもとで関数 v_{c_1} に適用する。) となる。そして h_2 にはこの限定継続を実行するための関数 $v_{c_2'}$ が渡され処理していく。これを目的の n 番目のハンドラに達するまで繰り返していくため、内側から i 番目 $(2 \leq i \leq n)$ のハンドラ h_i には、限定継続 $c_n' = \lambda v.\,\lambda m.\,v_{c_{n-1}}\,v\,c_n\,m$ の関数 $v_{c_n'} = \lambda v'.\,\lambda c'.\,\lambda m'.\,c_n'\,v'\,((c', h_n) :: m')$ が渡されることになる。そして目的のハンドラ h_n ではオペレーションの名前が一致したこととして処理を行うようになり、ここで限定継続 $v_{c_n'}$ が使われたとする。すると関数を展開していくことになり、最終的に

$$\lambda v.\,\lambda c'.\,\lambda m'.\,c_1\ v\ ((c_2,\ h_1) :: (c_3,\ h_2) :: ... :: (c_n,\ h_{n-1}) :: (c',\ h_n) :: m')$$

となる。

まとめると、Hillerström らの CPS インタプリタはリストの要素を増やして最後に要素を反転するのに対し、我々のインタプリタは関数の合成を行って後に展開していくという違いであり、同等の挙動をしていることが見て取れる。

また、deep handler のみ Hillerström らもリスト構造を使わず高階関数の形でのインタプリタも定義している。こちらとの比較としては、メタ継続の定義が多少異なるものの、我々と Hillerström らのインタプリタは本質的に同等のものとなっている。

4.2 shallow handler の場合

2.3 節の通り shallow handler の場合は名前付きオペレーションが模倣できなかったため、直近のハンドラで処理する場合でのみ限定継続を表す関数を比較する。

4.1.1 節と同様に Hillerström らの CPS インタプリタは $rk = (c,\ h) :: []$ を用いて限定継続を表す関数が定義される。

$$\lambda\ v\ ((c',\ h') :: m').\ \mathbf{app}\ ((c\ @\ c', h') :: m')\ v$$

$(c\ @\ c')$ の部分は実際のものとは厳密には異なるが、「元々囲んでいたハンドラ h は捨てて、その外側の pure continuations c' と c が合わさった下で c を実行すること」を表している。本稿で定義される shallow handler の限定継続の関数は

$$\lambda v.\,\lambda c'.\,\lambda t'.\,\lambda m'.\,c\ v\ (t\ @\ c' :: t')\ m'$$

[2]deep handler では実質的には trail を使用しないため今回は省略している。

e	$\Rightarrow$	$\langle e,\, [],\, [],\, id_t,\, [] \rangle$
$\langle x,\, \rho,\, c,\, t,\, m \rangle$	$\Rightarrow$	$\langle c,\, \rho(x),\, t,\, m \rangle$
$\langle \underline{\lambda} x.\, e,\, \rho,\, c,\, t,\, m \rangle$	$\Rightarrow$	$\langle c,\, \mathit{VFun}(e,\, x,\, \rho),\, t,\, m \rangle$
$\langle e_1 \mathbin{\underline{@}} e_2,\, \rho,\, c,\, t,\, m \rangle$	$\Rightarrow$	$\langle e_1,\, \rho,\, \mathit{CApp1}(e_2,\, \rho) :: c,\, t,\, m \rangle$
$\langle \underline{\mathit{try}}\, e_1\, \underline{\mathit{with}}\, (x; k).\, e_2,\, \rho,\, c,\, t,\, m \rangle$	$\Rightarrow$	$\langle e_1,\, \rho,\, [],\, id_t,\, (c,\, t,\, (x,\, k,\, e_2,\, \rho)) :: m \rangle$
$\langle \underline{\mathit{call}}(e),\, \rho,\, c,\, t,\, m \rangle$	$\Rightarrow$	$\langle e,\, \rho,\, \mathit{CCall} :: c,\, t,\, m \rangle$
$\langle [],\, v,\, id_t,\, [] \rangle$	$\Rightarrow$	v
$\langle [],\, v,\, id_t,\, (c,\, t,\, h) :: m \rangle$	$\Rightarrow$	$\langle c,\, v,\, t,\, m \rangle$
$\langle [],\, v,\, c,\, m \rangle$	$\Rightarrow$	$\langle c,\, v,\, id_t,\, m \rangle$
$\langle \mathit{CApp1}(e,\, \rho) :: c,\, v,\, t,\, m \rangle$	$\Rightarrow$	$\langle e,\, \rho,\, \mathit{CApp2}(v) :: c,\, t,\, m \rangle$
$\langle \mathit{CApp2}(\mathit{VFun}(e,\, x,\, \rho)) :: c,\, v,\, t,\, m \rangle$	$\Rightarrow$	$\langle e,\, \rho[x \mapsto v],\, c,\, t,\, m \rangle$
$\langle \mathit{CApp2}(\mathit{VContD}(c',\, t',\, h)) :: c,\, v,\, t,\, m \rangle$	$\Rightarrow$	$\langle c',\, v,\, t',\, (c,\, t,\, h) :: m \rangle$
$\langle \mathit{CApp2}(\mathit{VContS}(c',\, t')) :: c,\, v,\, t,\, m \rangle$	$\Rightarrow$	$\langle c',\, v,\, t' @ (\lambda v_2.\, \lambda t_2.\, \lambda m_2.\, c\, v_2\, t_2\, m_2) :: t,\, m \rangle$
$\langle \mathit{CCall} :: c,\, v,\, t,\, (c_0,\, t_0,\, h) :: m_0 \rangle$	$\Rightarrow$	*deep handler* の場合
where $h = (x,\, k,\, e_0,\, \rho_0)$		$\langle e_0,\, \rho_0[x \mapsto v][k \mapsto \mathit{VContD}(c,\, t,\, h)],\, c_0,\, t_0,\, m_0 \rangle$
		shallow handler の場合
		$\langle e_0,\, \rho_0[x \mapsto v][k \mapsto \mathit{VContS}(c,\, t)],\, c_0,\, t_0,\, m_0 \rangle$

図 5. 代数的エフェクトとハンドラのための抽象機械

となっていて、継続 c と c' が合わさるのではなく c' を trail に保持し、その下で c を実行している。そしてハンドラ h は c を囲まずに捨てられている。よって shallow handler の場合でも二つのインタプリタが同等の挙動であることが確認できる。

5　代数的エフェクトとハンドラのための抽象機械

代数的エフェクトとハンドラの機械語での実装の先駆けとして、本節では図 2 のインタプリタから導出した抽象機械を示す。抽象機械の導出には 4 種類の限定継続演算子のインタプリタに施した手法 [12] を用いている。代数的エフェクトとハンドラのための抽象機械もこの手法で問題なく導出することができた。手法の詳細としては、はじめに、高階関数として表現されている継続 c と ハンドラ h、クロージャを非関数化する。そして非関数化された継続 c は初期継続と関数適用の二つの継続、オペレーション呼び出しでの継続に対応するデータ構造が得られるが、ここで初期継続を空リスト、他の三つの継続の形はリストの cons とみなすことができる。そのため継続をリスト構造に変換する。継続のリストの要素は、関数適用 $e_1 \mathbin{\underline{@}} e_2$ の関数部分 e_1 を評価した後の継続 ($\mathit{CApp1}$) と引数部分 e_2 を評価後の継続 ($\mathit{CApp2}$)、$\underline{\mathit{call}}(e)$ で e を評価後のハンドラの処理に移るための継続 (CCall) をそれぞれ表すデータとなっている。以上の変換を施すとインタプリタの関数はすべて末尾呼び出しとなり、関数の引数を状態とみなすだけで抽象機械を導出することができる。

代数的エフェクトとハンドラの抽象機械は図 5 に示す。この抽象機械は式 e を $\langle e,\, [],\, [],\, \mathit{Bullet},\, [] \rangle$ として空の環境、継続、trail (Bullet) そしてメタ継続のもとではじめる。λ 計算は従来のもの [2, 9, 23] と変わらないためここでは割愛する。ハンドラ $\underline{\mathit{try}}\, e_1\, \underline{\mathit{with}}\, (x; k).\, e_2$ は e_1 を実行するが、そのときの継続と trail は空のもので新たにセットされ、現在の継続と trail はメタ継続に追加される。また一緒に $(x,\, k,\, e_2,\, \rho)$ として例外処理に必要なハンドラ情報も追加する。次に $\underline{\mathit{call}}(e)$ のときは引数 e を実行し、その後ハンドラの処理に移るために CCall を継続に追加する。そして CCall が呼び出されたとき、メタ継続の最初の要素 $(c_0,\, t_0,\, h)$ にアクセスし、h は $(x,\, k,\, e_0,\, \rho_0)$ となっている。すると式 e_0 を環境 ρ_0、継続 c_0、trail t_0、メタ継続 m_0 の下で実行してハンドラの処理に移る。このとき環境 ρ_0 には x が v、k が限定継続である情報が追加されるが、deep handler のときは VContD として現在の継続と trail、同様のハンドラを保持するようにし、shallow handler のときは VContS として現在の継続と trail のみを持つようにする。ここで deep handler のみハンドラの情報を持つのは限定継続にハンドラが囲まれるため必要となる。さらに、$\mathit{CApp2}$ として $\mathit{VContD}(c',\, t',\, h)$ や

$$
\begin{array}{rrcl}
\text{項の型} & \tau & := & \mathsf{int} \mid \tau_1 \to \tau_2 \langle \mu_\alpha, \sigma_\alpha \rangle \, \alpha \, \langle \mu_\beta, \sigma_\beta \rangle \, \beta \\
\text{trail の型} & \mu & := & \bullet \mid \tau_1 \to \langle \mu, \sigma \rangle \, \tau_2 \\
\text{メタ継続の型} & \sigma & := & [] \mid (\tau \to \langle \mu, \sigma \rangle \, \tau) \times \mu' \times \eta :: \sigma' \\
\text{ハンドラの型} & \eta & := & \tau_v \to \tau_{vc} \to \tau \, \langle \mu_\alpha, \sigma_\alpha \rangle \, \alpha \, \langle \mu_\beta, \sigma_\beta \rangle \, \beta \\
 & & & \tau_{vc} \ := \ \tau_1 \to \tau_2 \, \langle \mu_2, \sigma_2 \rangle \, \tau_3 \, \langle \mu_3, \sigma_3 \rangle \, \tau_4
\end{array}
$$

図 6. 型システム用の型定義

$\mathit{VContS}(c', t')$ が呼び出されたときは、それぞれ c' を実行するようにすることで限定継続を呼び出すようになっている。そして、その時点での継続 c、trail t は、VContD (deep handler) のときは h と一緒にメタ継続 m に追加してハンドラを囲むことを表現する。VContS (shallow handler) のときは 関数合成で t' に追加する。関数合成するために継続 c は η expand されている。trail に入れることによって c' の中で再び CCall が呼び出されると (オペレーション呼び出しが起こると) この増幅された trail ごと限定継続となりキャプチャされることとなる。

Hillerström らの既存の抽象機械 [14] と比較すると、CPS インタプリタと同様に、直近のハンドラで処理するときは deep と shallow の場合で同等の挙動をすることが確認できており、名前付きオペレーションの対応は deep handler のみ確認している。本稿の抽象機械はインタプリタから導出されたものであるため、抽象機械同士の比較も 4 節で説明した挙動を対応するコンストラクタで置き換えれば同様にして示される。

6 代数的エフェクトとハンドラのための型システム

本稿で提唱したインタプリタは継続 c やハンドラ h など全てを高階関数として定義している。高階関数で定義すると、特定のデータ構造を要求せずとも挙動を規定しており汎用性の高いものになっている。そのため、高階関数で定義されたインタプリタから自然に型システムを導くことが可能となる。具体的な手法としては Danvy と Filinski [5] や Cong ら [4] の方針に従い、インタプリタに型を割り振ることで導出していく。ハンドラが再帰的になる図 3 のインタプリタは再帰型が必要になるため、本稿では base type で型を付けられる図 2 のインタプリタから導いた型システムを提唱する。この型付き言語を定義し、それに対するインタプリタは定理証明支援言語 Agda で実装できており、停止性や健全性を証明した。

型システムのための型定義を図 6 に示す。型判定は $\Gamma \vdash e : \tau \, \langle \mu_\alpha, \sigma_\alpha \rangle \, \alpha \, \langle \mu_\beta, \sigma_\beta \rangle \, \beta$ という形になる。answer type に trail とメタ継続の型が付随している。これで「型環境 Γ のもとで式 e は τ 型を持ち、e を「τ 型の値と μ_α 型の trail、σ_α 型のメタ継続のもとで answer type が α になる継続」と μ_β 型の trail、σ_β 型のメタ継続のもとで実行すると answer type が β 型になる」と読む。

代数的エフェクトとハンドラのための型規則を図 7 に示す。λ 計算の三つの規則は trail とメタ継続を無視すれば一般的な型規則と同様となる。($\mathit{OperationCall{-}Deep}$) は deep handler で処理されるときの、($\mathit{OperationCall{-}Shallow}$) は shallow handler で処理されるときのオペレーション呼び出しの型規則となっている。($\mathit{Handler}$) の規則で使われる id-cont-type は初期継続に伴う制約で、($\mathit{OperationCall{-}Shallow}$) に出てくる compatible は trail を関数合成する際に必要な制約となっており、それぞれ図 8 に詳細を示す。$\equiv$ の両辺の型は同じ型であることを定義する。

7 関連研究

代数的エフェクトとハンドラと、4 種類の限定継続演算子の関係性は提唱されている。Forster ら [11] は型なし言語での deep handler と shift0/reset0 間の意味論とその同値性を証明している。Piróg

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \,\langle \mu_\alpha,\, \sigma_\alpha \rangle\, \alpha \,\langle \mu_\alpha,\, \sigma_\alpha \rangle\, \alpha} \;(\mathit{Var})$$

$$\frac{\Gamma,\ x : \tau_2 \vdash e : \tau_1 \,\langle \mu_\beta,\, \sigma_\beta \rangle\, \beta \,\langle \mu_\gamma,\, \sigma_\gamma \rangle\, \gamma}{\Gamma \vdash \underline{\lambda} x.\, e : (\tau_2 \to \tau_1 \,\langle \mu_\beta,\, \sigma_\beta \rangle\, \beta \,\langle \mu_\gamma,\, \sigma_\gamma \rangle\, \gamma) \,\langle \mu_\alpha,\, \sigma_\alpha \rangle\, \alpha \,\langle \mu_\alpha,\, \sigma_\alpha \rangle\, \alpha} \;(\mathit{Fun})$$

$$\frac{\Gamma \vdash e_1 : (\tau_2 \to \tau_1 \,\langle \mu_\alpha,\, \sigma_\alpha \rangle\, \alpha \,\langle \mu_2,\, \sigma_2 \rangle\, \beta) \,\langle \mu_1,\, \sigma_1 \rangle\, \gamma \,\langle \mu_\beta,\, \sigma_\beta \rangle\, \delta \quad \Gamma \vdash e_2 : \tau_2 \,\langle \mu_2,\, \sigma_2 \rangle\, \beta \,\langle \mu_1,\, \sigma_1 \rangle\, \gamma}{\Gamma \vdash e_1 \,\underline{@}\, e_2 : \tau_1 \,\langle \mu_\alpha,\, \sigma_\alpha \rangle\, \alpha \,\langle \mu_\beta,\, \sigma_\beta \rangle\, \delta} \;(\mathit{App})$$

$$\frac{\begin{array}{l}\eta = \tau' \to \tau_{vc} \to \tau_3 \,\langle \mu_3,\, \sigma_3 \rangle\, \tau_4 \,\langle \mu_4,\, \sigma_4 \rangle\, \tau_5 \\ \mathsf{id\text{-}cont\text{-}type}(\gamma,\, \mu_{id},\, \sigma_{id},\, \gamma') \\ \Gamma \vdash e_1 : \gamma \,\langle \mu_{id},\, \sigma_{id} \rangle\, \gamma' \,\langle \bullet,\, (\tau \to \langle \mu_\alpha,\, \sigma_\alpha \rangle\, \alpha) \times \mu_\beta \times \eta :: \sigma_\beta \rangle\, \beta \\ \Gamma,\ x : \tau',\ k : \tau_{vc} \vdash e_2 : \tau_3 \,\langle \mu_3,\, \sigma_3 \rangle\, \tau_4 \,\langle \mu_4,\, \sigma_4 \rangle\, \tau_5\end{array}}{\Gamma \vdash \underline{try}\, e_1\, \underline{with}\, (x; k).\, e_2 : \tau \,\langle \mu_\alpha,\, \sigma_\alpha \rangle\, \alpha \,\langle \mu_\beta,\, \sigma_\beta \rangle\, \beta} \;(\mathit{Handler})$$

$$\frac{\begin{array}{l}\eta = \tau' \to \tau_{vc} \to \tau_0 \,\langle \mu_0,\, \sigma_0 \rangle\, \tau_0' \,\langle \mu_0',\, \sigma_0' \rangle\, \gamma \\ \tau_{vc} = \tau \to \tau_1 \,\langle \mu_1,\, \sigma_1 \rangle\, \tau_2 \,\langle \mu_2,\, \sigma_2 \rangle\, \alpha \\ \Gamma \vdash e : \tau' \,\langle \mu_\alpha,\, (\tau_0 \to \langle \mu_0,\, \sigma_0 \rangle\, \tau_0') \times \mu_0' \times \eta :: \sigma_0' \rangle\, \gamma \,\langle \mu_\beta,\, \sigma_\beta \rangle\, \beta\end{array}}{\Gamma \vdash \underline{call_d}(e) : \tau \,\langle \mu_\alpha,\, (\tau_1 \to \langle \mu_1,\, \sigma_1 \rangle\, \tau_2) \times \mu_2 \times \eta :: \sigma_2 \rangle\, \alpha \,\langle \mu_\beta,\, \sigma_\beta \rangle\, \beta} \;(\mathit{Operation\,Call{-}Deep})$$

$$\frac{\begin{array}{l}\eta = \tau' \to \tau_{vc} \to \tau_0 \,\langle \mu_0,\, \sigma_0 \rangle\, \tau_0' \,\langle \mu_0',\, \sigma_0' \rangle\, \gamma \\ \tau_{vc} = \tau \to \tau_1 \,\langle \mu_1,\, \sigma_1 \rangle\, \tau_2 \,\langle \mu_2,\, \sigma_\alpha \rangle\, \alpha \\ \mathsf{compatible}((\tau_1 \to \langle \mu_1,\, \sigma_1 \rangle\, \tau_2),\, \mu_2,\, \mu_2') \\ \mathsf{compatible}(\mu',\, \mu_2',\, \mu_\alpha) \\ \Gamma \vdash e : \tau' \,\langle \mu',\, (\tau_0 \to \langle \mu_0,\, \sigma_0 \rangle\, \tau_0') \times \mu_0' \times \eta :: \sigma_0' \rangle\, \gamma \,\langle \mu_\beta,\, \sigma_\beta \rangle\, \beta\end{array}}{\Gamma \vdash \underline{call_s}(e) : \tau \,\langle \mu_\alpha,\, \sigma_\alpha \rangle\, \alpha \,\langle \mu_\beta,\, \sigma_\beta \rangle\, \beta} \;(\mathit{Operation\,Call{-}Shallow})$$

図 7. 代数的エフェクトとハンドラのための型システム

$$\begin{array}{rcl}
\mathsf{id\text{-}cont\text{-}type}(\tau,\, \bullet,\, [],\, \tau') & = & \tau \equiv \tau' \\
\mathsf{id\text{-}cont\text{-}type}(\tau,\, \bullet,\, (\tau_1 \to \langle \mu_1,\, \sigma_1 \rangle\, \tau_1') \times \mu_2 \times \eta :: \sigma,\, \tau') & = & (\tau_1 \equiv \tau) \wedge (\mu_1 \equiv \mu_2) \\
 & & \quad \wedge (\sigma_1 \equiv \sigma_2) \wedge (\tau_1' \equiv \tau') \\
\mathsf{id\text{-}cont\text{-}type}(\tau,\, (\tau_1 \to \langle \mu_1,\, \sigma_1 \rangle\, \tau_1'),\, \sigma,\, \tau') & = & (\tau_1 \equiv \tau) \wedge (\mu_1 \equiv \bullet) \wedge (\sigma_1 \equiv \sigma) \wedge (\tau_1' \equiv \tau') \\
 & & \\
\mathsf{compatible}(\bullet,\, \mu_2,\, \mu_3) & = & \mu_2 \equiv \mu_3 \\
\mathsf{compatible}((\tau_1 \to \langle \mu_1,\, \sigma_1 \rangle\, \tau_1'),\, \bullet,\, \mu_3) & = & (\tau_1 \to \langle \mu_1,\, \sigma_1 \rangle\, \tau_1') \equiv \mu_3 \\
\mathsf{compatible}((\tau_1 \to \langle \mu_1,\, \sigma_1 \rangle\, \tau_1'),\, (\tau_2 \to \langle \mu_2,\, \sigma_2 \rangle\, \tau_2'),\, \bullet) & = & \bot \\
\mathsf{compatible}(\tau_1 \to \langle \mu_1,\, \sigma_1 \rangle\, \tau_1',\ \tau_2 \to \langle \mu_2,\, \sigma_2 \rangle\, \tau_2',\ \tau_3 \to \langle \mu_3,\, \sigma_3 \rangle\, \tau_3') & = & (\tau_1 \equiv \tau_3) \wedge (\sigma_1 \equiv \sigma_3) \wedge (\tau_1' \equiv \tau_3') \\
 & & \quad \wedge\, \mathsf{compatible}(\tau_2 \to \langle \mu_2,\, \sigma_2 \rangle\, \tau_2',\, \mu_2,\, \mu_3))
\end{array}$$

図 8. 初期継続と、trail の合成に伴う型システムの制約

ら [19] は型付き言語での deep handler と shift0/reset0 および shallow handler と control0/prompt0 が意味論的に同値であることを示している。本稿の代数的エフェクトとハンドラのインタプリタでも Piróg らの変換方法を用いて 4 種類の限定継続演算子を模倣できることを型なしで確認している。

Kammar [16] らは Haskell のモナドを用いて effect handlers を実装している。この論文では deep handler だけでなく shallow handler が紹介されている。そして deep handler と shallow handler それぞれの型システムが示されており、deep と shallow の場合とで、限定継続を表す変数の型に違いが現れている。また継続モナドや自由モナドによって実装されているため CPS インタプリタに非常に近いものと考えられるが、本研究との対応関係については未確認のため今後の課題となる。

Leijen [17] は Koka 言語の row-type を用いて effect polymorphism を実現している。effect polymorphism によって名前付きオペレーションに対応した 代数的エフェクトとハンドラのための型システムが示されたもとで、Koka 言語での実装がされている。

Biernacki ら [3] は代数的エフェクトとハンドラにさらに lift と呼ばれるコンストラクタを追加している。複数のハンドラに同じ名前のオペレーションが定義されている状況でその名前のオペレーションを呼び出すとき、リフトでオペレーションを囲むことで直近のハンドラよりも外側にあるハンドラでの処理を指定することが可能となる。この計算の型規則は effect polymorphism に基づき lift の規則も追加されて示されている。

これらの型システムと本研究の型システムとの対応関係を確認することも今後の課題となる。

8　まとめと今後の課題

本稿では trail やメタ継続を用いた新たな代数的エフェクトとハンドラのための CPS インタプリタを提唱した。インタプリタの設計として、ハンドラが deep/shallow の意味を持たせるタイミングをオペレーション呼び出しの評価時に行うものと、ハンドラの評価時に行うものの 2 種類を取り上げた。そしてインタプリタから抽象機械や型システムを機械的に導出した。また構文として名前付きオペレーションに対応していなかったが、deep handler であれば条件文を追加して模倣すれば従来のインタプリタと同様の挙動であることも示した。

今後は、shallow handler の名前付きオペレーションの模倣を可能にすることが課題となる。そして我々の型システムとこれまでに提唱された型システムとの等価性を確かめたい。また、4 種類の限定継続演算子のインタプリタに対して示した手法 [12] を本稿のインタプリタにも用いて、仮想機械を導出し代数的エフェクトとハンドラのための低レベルな直接実装に向けた基盤を提供したい。

謝辞

有益なコメントをくださった査読者の皆様に感謝申し上げます。本研究は一部 JSPS 科研費 18H03218 の助成を受けたものです。

参考文献

[1] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*, Vol. 84, No. 1, pp. 108–123, 2015.

[2] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 38, No. 1, pp. 1–25, 2015.

[3] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *Proceedings of the ACM on Programming Languages*, Vol. 2, No. POPL, pp. 1–30, 2017.

[4] Youyou Cong, Chiaki Ishio, Kaho Honda, and Kenichi Asai. A functional abstraction of typed invocation contexts. In *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*, pp. 12:1–12:18, 2021.

[5] Olivier Danvy and Andrzej Filinski. *A functional abstraction of typed contexts*. BRICS 89/12, 1989.

[6] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pp. 151–160, 1990.

[7] Olivier Danvy and Andrzej Filinski. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, Vol. 2, No. 4, pp. pp. 361–391, 1992.

[8] Paul Downen and Zena M Ariola. A systematic approach to delimited control with multiple prompts. In *European Symposium on Programming*, pp. 234–253. Springer, 2012.

[9] R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *J. Funct. Program.*, Vol. 17, No. 6, pp. 687–730, 2007.

[10] Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pp. 180–190, New York, NY, USA, 1988. ACM.

[11] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, Vol. 1, No. ICFP, pp. 13:1–13:29, August 2017.

[12] Maika Fujii and Kenichi Asai. Derivation of a virtual machine for four variants of delimited-control operators. In *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*, pp. 16:1–16:19, 2021.

[13] 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, pp. 12–23, New York, NY, USA, 1995. ACM.

[14] Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect Handlers via Generalised Continuations. *Journal of Functional Programming*, Vol. 30, No. e5, 2020.

[15] Yukiyoshi Kameyama. Axioms for control operators in the cps hierarchy. *Higher-Order and Symbolic Computation*, Vol. 20, No. 4, pp. 339–369, 2007.

[16] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pp. 145–158, New York, NY, USA, 2013. ACM.

[17] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 486–499, 2017.

[18] Marek Materzok and Dariusz Biernacki. Subtyping Delimited Continuations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pp. 81–93, New York, NY, USA, 2011. ACM.

[19] Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Typed equivalence of effect handlers and delimited control. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[20] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, Vol. 5502 of *Lecture Notes in Computer Science*, pp. 80–94. Springer, 2009.

[21] John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pp. 717–740, 1972.

[22] John C Reynolds. Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation*, Vol. 11, No. 4, pp. 363–397, 1998.

[23] Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, Vol. 20, No. 4, pp. 371–401, 2007.

[24] 石尾千晶, 浅井健一. 4 種類の限定継続演算子のための型システム. 第 24 回プログラミングおよびプログラミング言語ワークショップ論文集, 2022.