algebraic effects を含むプログラムのステップ実行

古川 つきの, 浅井 健一

お茶の水女子大学

furukawa.tsukino@is.ocha.ac.jp, asai@is.ocha.ac.jp

概 要　ステッパはプログラムの実行過程を見せるツールである。これまで様々な言語に対するステッパが作られてきたが、shift/reset や algebraic effects といった継続を明示的に扱う言語機能をサポートするステッパは作られていない。継続を扱うプログラムの挙動を理解するのは困難なので、そういった言語に対応したステッパを作ることが本研究の目的である。

ステッパは簡約のたびにその時点でのプログラム全体を出力するインタプリタなので、実行している部分式のコンテキストの情報が常に必要になる。継続を扱うような複雑な機能を持つ言語を対象にしたステッパでは、コンテキストがどのような構造をしているかが自明でない。そこで、通常のインタプリタ関数をプログラム変換することで機械的にコンテキストの情報を保持させてステッパを実装する方法を示し、実際に型無しλ計算と algebraic effects から成る言語に対するステッパを実装する。

1　はじめに

プログラムの実行の様子を確認する方法に、実行が 1 段階進むごとにどのような状態になっているのかを書き連ねる方法がある。このツールは、代数的ステッパ（以後、単にステッパと書く）と呼ばれ、プログラムが代数的に書き換わる様子を 1 ステップずつ表示してくれるものである。例えば OCaml のプログラム `let a = 1 + 2 in 4 + a` を入力されると、ステッパは以下のような実行ステップを表す文字列を出力する。

```
Step 0:  (let a = (1 + 2) in (4 + a))
Step 1:  (let a = 3 in (4 + a))

Step 1:  (let a = 3 in (4 + a))
Step 2:  (4 + 3)

Step 2:  (4 + 3)
Step 3:  7
```

このように 1 段階ずつ確認していけば、具体的にどのような計算がされるのかを観察することができ、プログラムの動きを理解しやすい。またこの方法はデバッグにおいても有用で、どの段階で想定と違うことが起こっているのかが見えるので、プログラムのどの部分がその原因なのかが分かりやすくなる。そのため、お茶の水女子大学では実際に関数型言語の授業でステッパを本格的に使用している。

しかし、継続を操作するようなプログラムだった場合、ステッパをどのように作ったら良いのかは明らかではない。そのような複雑なプログラムでこそ実行の様子を詳細に追いたいところだが、現在のところそのような言語に対するステッパは作られていない。わずかに、我々が過去の研究 [3] で例外処理のための構文 try-with を含む言語についてのステッパを実装した程度である。

そこで我々は、ステッパを、継続を明示的に扱う言語機能に対応させることを目指している。ステッパは、通常のインタプリタ関数に出力機能を追加することで実装できるが、その際、式全体を再構成するためにコンテキストの情報が必要になる。以前の研究 [3] では、部分式の簡約に進むたびにコンテキストを表すデータ型を作成していたが、継続を操作するようなプログラムの場合、どのようなコンテキストにすれば良いのかは即座には明らかではない。

そこで、本論文では、インタプリタに対して CPS 変換 [7] と非関数化 [9] を施すことで機械的にコンテキストの情報を得る。この方法を使うと、継続を操作するような言語でも機械的にコンテキストの情報を得ることができ、それを使ってステッパを作ることができるようになる。本論文では、この手法を algebraic effects を含む言語に対して適用し、algebraic effects を含む言語に対するステッパを作成する。また、その過程で algebraic effects を含む言語に対する definitional interpreter を示す。本論文では詳しくは述べないが、この手法は shift/reset に対するステッパの作成にも使うことができる。

本論文の構成は以下の通りである。まず 2 節でステッパを実装する方法を紹介する。そして 3 節で algebraic effects を含む言語およびそのインタプリタを定義し、4 節でインタプリタを変換してステッパを得るまでの過程を説明する。5 節では他のいくつかの言語に対するステッパを同様の変換によって得ることについて議論する。6 節で関連研究について触れ、7 節でまとめる。

2　ステッパの実装方法とコンテキスト

ステッパは small-step による実行と同じなので、small-step のインタプリタを書けば実装できる。実際、Whitington & Ridge [10] は small-step のインタプリタを書くことで OCaml に対するステッパを実装している。しかし、small-step のインタプリタをメンテナンスするのは簡単ではない。また、ステップ実行中に関数呼び出し単位でスキップする機能をつけようと思うとインタプリタは big-step で書かれていた方が都合が良い。そこで、我々の過去の研究 [3] では、big-step のインタプリタを元にしてステッパを作成している。ここでも、そのアプローチをとる。

図 1 に OCaml による型無し λ 計算の定義と代入ベースの big-step インタプリタの実装を示す。関数 `subst : e -> (string * v) list -> e` は代入関数であり、`subst e [(x, v)]` は式 `e` の中の全ての変数 `x` を値 `v` に置換した式を返す。

このインタプリタをステッパにするには、簡約をする際に簡約前後のプログラムを出力する機能を追加すればよい。しかしステッパが出力したいのは実行中の部分式ではなく式全体であり、コンテキストを含めた式全体を出力するためには、実行中の式の構文木の他にコンテキストの情報が必要である。

コンテキストの情報を得るために、Clements ら [2] は Racket の continuation mark を使用してコンテキストフレームの情報を記録することでステッパを実装した。本研究ではそのような特殊な機能は使わずに、インタプリタ関数に明示的にコンテキスト情報のための引数を追加する。図 1 のインタプリタにその変更を施すと、図 2 のようになる。ここで、関数 `memo : e -> e -> c -> unit` は、簡約前の式、簡約後の式、コンテキスト情報の 3 つを引数にとり、コンテキスト情報を利用して簡約前後の式全体をそれぞれ出力するものである。

図 2 のように、コンテキストを表すデータ型を定義して再帰呼び出し時の構造に合わせて引数として渡すようにすれば、式全体を再構成して出力することが可能になる。ここで、コンテキストを表すデータ型は、評価文脈そのものになっていることに気がつく。評価文脈のデータ型は、big-step のインタプリタを CPS 変換し、非関数化すると機械的に得られることが知られている。これは、我々が手動で定義したコンテキストのデータは、機械的に導出できることを示唆している。

λ 計算に対するステッパであれば、手動でコンテキストの型を定義するのは簡単だが、言語が複雑になってくると必ずしもこれは自明ではない。実際、以前の研究 [3] で try-with 構文を含む言語

```
(* 値 *)
type v = Var of string      (* x *)
       | Fun of string * e  (* fun x -> e *)

(* 式 *)
type e = Val of v      (* 値 *)
       | App of e * e  (* e e *)

(* インタプリタ *)
let rec eval (exp : e) : v = match exp with
  | Val (v) -> v  (* 値ならそのまま返す *)
  | App (e1, e2) ->
    (let v2 = eval e2 in  (* 引数部分を実行 *)
     let v1 = eval e1 in  (* 関数部分を実行 *)
     let reduct = match v1 with
       | Fun (x, e) -> subst e [(x, v2)]  (* 代入 e[v2/x] *)
       | _ -> failwith "type error" in  (* 関数部分が関数でなければ型エラー *)
     eval reduct)  (* 代入後の式を実行 *)
```

図 1. 型無し λ 計算とそのインタプリタ

```
(* コンテキスト *)
type c = CId             (* [.] *)
       | CApp2 of e * c  (* [e [.]] *)
       | CApp1 of v * c  (* [[.] v] *)

(* 出力しながら再帰的に実行 *)
let rec eval (exp : e) (c : c) : v = match exp with
  | Val (v) -> v
  | App (e1, e2) ->
    (let v2 = eval e2 (CApp2 (e1, c)) in  (* コンテキストを 1 層深くする *)
     let v1 = eval e1 (CApp1 (v2, c)) in  (* コンテキストを 1 層深くする *)
     let redex = App (Val v1, Val v2) in
     let reduct = match v1 with
       | Fun (x, e) -> subst e [(x, v2)]
       | _ -> failwith "type error" in
     memo redex reduct c;                (* コンテキストを利用して式全体を出力 *)
     eval reduct c)

(* 実行を始める *)
let stepper (exp : e) = eval exp CId
```

図 2. 型無し λ 計算に対するステッパ

のステッパを実装したときには、コンテキストを try-with 構文で区切る必要があったため、コンテキストの構造が一次元的でなく、リストのリストになった。algebraic effects などが入った場合、どのようなコンテキストを使えば良いのかはまた別途、考慮する必要がある。このような場合、機械的にコンテキストの定義を導出できることにはメリットがある。次節以降ではそのような方針で algebraic effects に対するステッパを導出する。

3 Algebraic effects とインタプリタの定義

この節では、algebraic effects を導入した後、型無し λ 計算と algebraic effects からなる言語を示し、そのインタプリタを定義する。

3.1 algebraic effects

algebraic effects は、例外や状態などの副作用を表現するための一般的な枠組で、副作用を起こす部分（オペレーション呼び出し）と処理する部分（ハンドラ）からなる [8]。特徴は、副作用の意味がそれを処理するハンドラ部分で決まるところである。例えば、以下のプログラムを考える。

```
with {return x -> x;
      op(x; k) -> k (x + 1)}
handle 10 + op(3)
```

`with h handle e` は、`h` というハンドラのもとで式 `e` を実行するという意味である。`e` の部分を見ると `10 + op(3)` とあるので加算を行おうとするが、そこで `op(3)` というオペレーション呼び出しが起こる。オペレーション呼び出しというのは副作用を起こす命令で、直感的にはここで例外 `op` を引数 3 で起こすのに近い。使えるオペレーションはあらかじめ宣言するのが普通だが、本論文では使用するオペレーションは全て定義されていると仮定する。

オペレーション呼び出しが起こると、プログラムの制御はハンドラ部分に移る。ハンドラは正常終了を処理する部分 `return x -> ...` とオペレーション呼び出しを処理する部分に分かれている。正常終了する部分は `with h handle e` の `e` 部分の実行が終了した場合に実行され、`x` に実行結果が入る。上の例なら、その `x` がそのまま返されて、これがプログラム全体の結果となる。

一方、`e` の実行中にオペレーション呼び出しがあった場合は、オペレーション呼び出しの処理が行われる。まず、呼び出されたオペレーションが処理するオペレーションと同じものかがチェックされる。異なる場合は、そのオペレーションはここでは処理されず、さらに外側の `with handle` 文で処理されることになる。（最後まで処理されなかったら、未処理のオペレーションが報告されてプログラムは終了する。）一方、ここで処理すべきオペレーションと分かった場合には、矢印の右側の処理に移る。ここで、`x` の部分にはオペレーションの引数が入り、`k` の部分には「オペレーション呼び出しから、この `with handle` 文までの限定継続」が入る。`k` に限定継続が入るところが例外とは異なる部分である。上の例では、矢印の右側が `k (x + 1)` となっているので、`x` の値である 3 に 1 が加わった後、もとの計算である `10 + [.]` が再開され、全体として 14 が返ることになる。

algebraic effects の特徴は、オペレーション呼び出しの意味がハンドラで決まる部分にある。`op(3)` とした時点ではこの処理の内容は未定だが、ハンドラ部分に `k (x + 1)` と書かれているため、結果として `op` は 1 を加えるような作用だったことになる。

3.2 構文の定義

型無し λ 計算と algebraic effects からなる対象言語を図 3 の `e` と定義する。ただし継続 `fun x => e` は入力プログラムに含まれることはなく、実行の過程のみで現れる構文である。`h` のオペレーション節に出てくる `op` たちは互いに全て異ならくてはいけない。

```
v  :=                                    (値)
      x                                  変数
    | fun x -> e                         関数
    | fun x => e                         継続
e  :=                                    (式)
      v                                  値
    | e e                                関数適用
    | op e                               オペレーション呼び出し
    | with h handle e                    ハンドル
h  :=                                    (ハンドラ)
      {return x -> e;                    return 節
       op(x; k) -> e; ...; op(x; k) -> e}  オペレーション節 (0 個以上)
```

図 3. 対象言語の構文

3.3 CPS インタプリタによる意味論

この節では、algebraic effects を含む言語に対する意味論を与える。オペレーション呼び出しにより非局所的に制御が移るので、意味論は CPS インタプリタを定義することで与える。対象言語の OCaml による定義を図 4 に示す。ここで `k` は各ハンドラ内部の限定継続を表す。また、`a` は `handle` 節内の式の実行が正常終了したのかオペレーション呼び出しだったのかを示す型である。

図 4 の言語に対する call-by-value かつ right-to-left のインタプリタを図 5 に定義する。ただし、関数 `subst : e -> (string * v) list -> e` は代入のための関数であり、`subst e [(x, v); (k, cont_value)]` は `e` の中の変数 `x` と変数 `k` に同時にそれぞれ値 `v` と値 `cont_value` を代入した式を返す。関数 `search_op` はハンドラ内のオペレーションを検索する関数で、例えば {`return x -> x; op1(y, k) -> k y`} を表すデータを `h` とすると `search_op "op2" h` は `None` を返し `search_op "op1" h` は `Some (y, k, App (Var "k", Var "y"))` を返す。

このインタプリタは、`handle` 節内の実行については普通の CPS になっており、メタ継続である `k` は「直近のハンドラまでの継続」である。関数 `eval` の下から 2 行目で `with handle` 文を実行する際、再帰呼び出しの継続として `(fun x -> Return x)` を渡していて、これによって `handle` 節の実行に入るたびに渡す継続を初期化している。

`handle` 節内を実行した結果を表すのが `a` 型である。`handle` 節内の実行は、オペレーション呼び出しが行われない限りは通常の CPS インタプリタによって進むが、オペレーション呼び出しが行われた場合（`eval` の下から 4 行目）は引数 `e` を実行後、結果を継続 `k` に渡すことなく `OpCall` を返している。これが `handle` 節の結果となり、`eval` の最下行で `apply_handler` に渡される。一方、`handle` 節内の実行が正常終了した場合は、初期継続 `(fun x -> Return x)` に結果が返り、それが `apply_handler` に渡される。

ここで、オペレーション呼び出しで返される `OpCall` の第 3 引数が `k` ではなく `fun v -> k v` のように η-expand されているのに注意しよう。このようにしているのは、`k` が「直近のハンドラまでの継続」を表しているのに対し、`OpCall` の第 3 引数はより広い継続を指すことがあり両者を区別したいためである。これについては、次節で非関数化を施す際に詳しく述べる。

`apply_handler` は、そのときの継続 `k`、処理すべきハンドラ `h`、そして `handle` 節内の実行結果 `a` を受け取ってハンドラの処理をする。関数 `apply_handler` の動作は `handle` 節の実行結果とハンドラの内容によって 3 種類ある。

1. `handle` 節が値 `v` になった場合：ハンドラの return 節 `return x -> e` を参照して、`e[v/x]`

```
(* 値 *)
type v = Var of string      (* x *)
       | Fun of string * e  (* fun x -> e *)
       | Cont of (k -> k)   (* 継続 fun x => ... *)
(* ハンドラ *)
and h = {return : string * e;                          (* {return x -> e;      *)
         ops : (string * string * string * e) list}  (*  op(x; k) -> e; ...} *)
(* 式 *)
and e = Val of v          (* v *)
      | App of e * e      (* e e *)
      | Op of string * e  (* op e *)
      | With of h * e     (* with h handle e *)
(* handle 内の継続 *)
and k = v -> a
(* handle 内の実行結果 *)
and a = Return of v               (* 値になった *)
      | OpCall of string * v * k  (* オペレーションが呼び出された *)
```

図 4. 対象言語の定義

を実行

2. handle 節がオペレーション呼び出し OpCall (name, v, k') になった場合で、そのオペレーション name がハンドラ内で定義されていなかった場合：さらに外側の with handle 文に処理を移すため、handle 節内の限定継続 k' に、1 つ外側の handle までの限定継続を合成した継続 fun v -> ... を作り、それを OpCall (name, v, (fun v -> ...)) と返す。この OpCall の第 3 引数は「直近のハンドラまでの継続」ではなく、より広い継続となっている。
3. handle 節がオペレーション呼び出し OpCall (name, v, k') になった場合で、そのオペレーション name がハンドラ内で定義されていた場合：そのハンドラの定義 name (x; y) -> e を参照し、e[v/x, cont_value/y] を実行する。(cont_value については、以下の説明を参照。)

オペレーション呼び出しを処理する際に k に束縛する限定継続 cont_value は、「オペレーション呼び出し時の限定継続 k'」に「現在のハンドラ h」と「cont_value が呼び出された時の継続 k''」を合成したものである。

このようにして作られた限定継続が呼び出されるのは eval の App の Cont のケースである。cont_value は、この継続が呼び出された時点での限定継続が必要なので、それを cont_value k のように渡してから値 v2 を渡している。

これまで、algebraic effects の意味論は small-step のもの [6, 8] 以外には CPS で書かれた big-step のもの [5] が提示されてきたが、この意味論はすでに部分式に名前が与えられている（A-正規形になっている）ことを仮定している上に、毎回、捕捉する継続を計算しているなど実装には必ずしも合ったものとは言えなかった。ここで示した CPS インタプリタは単純で、ハンドラの意味を的確に捉えており、algebraic effects の定義を与えるインタプリタ (definitional intepreter) と捉えて良いのではないかと考えている。

```
(* CPS インタプリタ *)
let rec eval (exp : e) (k : k) : a = match exp with
  | Val (v) -> k v  (* 継続に値を渡す *)
  | App (e1, e2) ->
    eval e2 (fun v2 ->  (* FApp2 に変換される関数 *)
      eval e1 (fun v1 -> match v1 with  (* FApp1 に変換される関数 *)
        | Fun (x, e) ->
          let reduct = subst e [(x, v2)] in  (* e[v2/x] *)
          eval reduct k
        | Cont (cont_value) -> (cont_value k) v2
          (* 現在の継続と継続値が保持するメタ継続を合成して値を渡す *)
        | _ -> failwith "type error"))
  | Op (name, e) ->
    eval e (fun v -> OpCall (name, v, fun v -> k v))  (* FOp に変換される関数 *)
  | With (h, e) ->
    let a = eval e (fun v -> Return v) in  (* FId に変換される関数、空の継続 *)
    apply_handler k h a  (* handle 節内の実行結果をハンドラで処理 *)

(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler (k : k) (h : h) (a : a) : a = match a with
  | Return v ->                          (* handle 節内が値 v を返したとき *)
    (match h with {return = (x, e)} ->  (* h は {return x -> e, ...} として *)
      let reduct = subst e [(x, v)] in (* e[v/x] に簡約される *)
      eval reduct k)                    (* e[v/x] を実行 *)
  | OpCall (name, v, k') ->        (* オペレーション呼び出しがあったとき *)
    (match search_op name h with
      | None ->                     (* ハンドラで定義されていない場合、 *)
        OpCall (name, v, (fun v ->  (* OpCall の継続の後に現在の継続を合成 *)
          let a' = k' v in
          apply_handler k h a'))
      | Some (x, y, e) ->           (* ハンドラで定義されている場合、 *)
        let cont_value =
          Cont (fun k'' -> fun v -> (* 適用時にその後の継続を受け取って合成 *)
            let a' = k' v in
            apply_handler k'' h a') in
        let reduct = subst e [(x, v); (y, cont_value)] in
        eval reduct k)

(* 初期継続を渡して実行を始める *)
let interpreter (e : e) : a = eval e (fun v -> Return v) (* FIdに変換される関数 *)
```

図 5. 継続渡し形式で書かれたインタプリタ

```
(* handle 内の継続 *)
and k = FId                  (* [.] *)
      | FApp2 of e * k       (* [e [.]] *)
      | FApp1 of v * k       (* [[.] v] *)
      | FOp of string * k    (* [op [.]] *)
```

図 6. 非関数化後の継続の型

4 インタプリタの変換

本節では、3 節で定義したインタプリタ (図 5) に対して、正当性の保証された 2 種類のプログラム変換（非関数化と CPS 変換）をかけることで、コンテキストを明示的に保持するインタプリタを得て、そこからステッパを作成する方法を示す。

4.1 非関数化

2 節で示したインタプリタは直接形式だったので、コンテキスト情報を得るのに CPS 変換をかけてから非関数化をかけたが、3 節で示したインタプリタはオペレーション呼び出しをサポートするため最初から CPS で書かれている。したがって、ここではまず非関数化をかける。

非関数化というのは、高階関数を 1 階のデータ構造で表現する方法である。高階関数は全てその自由変数を引数に持つような 1 階のデータ構造となり、高階関数を呼び出していた部分は apply 関数の呼び出しとなる。この apply 関数は、高階関数が呼び出されていたら行ったであろう処理を行うように別途、定義されるものである。この変換は機械的に行うことができる。

具体的に図 5 のプログラムの継続 `k` 型の λ 式を非関数化するには次のようにする。結果は図 6 と図 7 のようになる。

1. 継続を表す λ 式をコンストラクタに置き換える。その際、λ 式内の自由変数はコンストラクタの引数にする。その結果、得られるデータ構造は図 6 のようになる。図 5 の中には、コメントとしてどの関数がどのコンストラクタに置き換わったのかが書かれている。
2. 関数を表すコンストラクタと引数を受け取って中身を実行するような apply 関数を定義する。これは、図 7 では `apply_in` と呼ばれている。
3. λ 式を呼び出す部分を、apply 関数にコンストラクタと引数を渡すように変更する。

非関数化したした後の継続の型を見ると、ラムダ計算の通常の評価文脈に加えてオペレーション呼び出しの引数を実行するフレーム `FOp` が加わっていることがわかる。これが、ハンドラ内の実行のコンテキスト情報である。

ここで、`OpCall` の第 3 引数は非関数化されていないことに注意しよう。この部分はハンドラ内の継続とは限らないので、ここでは非関数化せずにもとのままとしている。ここを非関数化することも可能ではあるが、そうすると最終的に得られるコンテキスト情報がきれいなリストのリストの形にはならなくなってしまう。

ハンドラ内の評価文脈を表すデータ構造は非関数化により導くことができたが、図 7 のインタプリタはオペレンーション呼び出しなどの実装で継続を非末尾の位置で使っており純粋な CPS 形式にはなっていないため、全体のコンテキストは得られていない。そのため、このコンテキストを使ってステッパを構成してもプログラム全体を再構成することはできない。プログラム全体のコンテキストを得るためには、このインタプリタに対してもう一度 CPS 変換と非関数化を施し、純粋な CPS 形式にする必要がある。

```
(* CPS インタプリタを非関数化した関数 *)
let rec eval (exp : e) (k : k) : a = match exp with
  | Val (v) -> apply_in k v  (* 継続適用関数に継続と値を渡す *)
  | App (e1, e2) -> eval e2 (FApp2 (e1, k))
  | Op (name, e) -> eval e (FOp (name, k))
  | With (h, e) -> let a = eval e FId in  (* 空の継続を渡す *)
    apply_handler k h a  (* handle 節内の実行結果をハンドラで処理 *)

(* handle 節内の継続を適用する関数 *)
and apply_in (k : k) (v : v) : a = match k with
  | FId -> Return v  (* 空の継続、そのまま値を返す *)
  | FApp2 (e1, k) -> let v2 = v in eval e1 (FApp1 (v2, k))
  | FApp1 (v2, k) -> let v1 = v in
    (match v1 with
     | Fun (x, e) ->
       let reduct = subst e [(x, v2)] in
       eval reduct k
     | Cont (cont_value) -> (cont_value k) v2
     | _ -> failwith "type error")
  | FOp (name, k) ->
    OpCall (name, v, (fun v -> apply_in k v))  (* Op 呼び出しの情報を返す *)

(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler (k : k) (h : h) (a : a) : a = match a with
  | Return v ->
    (match h with {return = (x, e)} ->
       let reduct = subst e [(x, v)] in
       eval reduct k)
  | OpCall (name, v, va) ->
    (match search_op name h with
     | None -> OpCall (name, v, (fun v ->
          let a' = va v in
          apply_handler k h a'))
     | Some (x, y, e) ->
       let cont_value =
         Cont (fun k'' -> fun v ->
             let a' = va v in
             apply_handler k'' h a') in
       let reduct = subst e [(x, v); (y, cont_value)] in
       eval reduct k)

(* 初期継続を渡して実行を始める *)
let interpreter (e : e) : a = eval e FId
```

図 **7.** CPS インタプリタを非関数化して CPS 変換したプログラム

```
(* CPS インタプリタを非関数化して CPS 変換した関数 *)
let rec eval (exp : e) (k : k) (k2 : k2) : a = match exp with
  | Val (v) -> apply_in k v k2
  | App (e1, e2) -> eval e2 (FApp2 (e1, k)) k2
  | Op (name, e) -> eval e (FOp (name, k)) k2
  | With (h, e) ->
    eval e FId (fun a -> apply_handler k h a k2)  (* GHandle に変換される *)

(* handle 節内の継続を適用する関数 *)
and apply_in (k : k) (v : v) (k2 : k2) : a = match k with
  | FId -> k2 (Return v)  (* handle 節の外の継続を適用 *)
  | FApp2 (e1, k) -> let v2 = v in eval e1 (FApp1 (v2, k)) k2
  | FApp1 (v2, k) -> let v1 = v in
    (match v1 with
     | Fun (x, e) ->
       let reduct = subst e [(x, v2)] in
       eval reduct k k2
     | Cont (cont_value) ->
       (cont_value k) v2 k2
     | _ -> failwith "type error")
  | FOp (name, k) ->
    k2 (OpCall (name, v, (fun v -> fun k2' -> apply_in k v k2')))

(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler (k : k) (h : h) (a : a) (k2 : k2) : a = match a with
  | Return v ->
    (match h with {return = (x, e)} ->
       let reduct = subst e [(x, v)] in
       eval reduct k k2)
  | OpCall (name, v, va) ->
    (match search_op name h with
     | None ->
       k2 (OpCall (name, v, (fun v -> fun k2' ->  (* 外の継続を適用 *)
           va v (fun a' -> apply_handler k h a' k2'))))  (* GHandle に変換 *)
     | Some (x, y, e) ->
       let cont_value =
         Cont (fun k'' -> fun v -> fun k2 ->
             va v (fun a' -> apply_handler k'' h a' k2)) in  (* GHandle に変換 *)
       let reduct = subst e [(x, v); (y, cont_value)] in
       eval reduct k k2)

(* 初期継続を渡して実行を始める *)
let interpreter (e : e) : a = eval e FId (fun a -> a)  (* GId に変換される *)
```

図 **8.** CPS インタプリタを非関数化して CPS 変換したプログラム

```
(* 全体のメタ継続 *)
and k2 = GId
       | GHandle of h * k * k2
```

図 9. 2 回目の非関数化後の継続の型

4.2 CPS 変換

図 7 では、末尾再帰でない再帰呼び出しの際に継続が初期化されてしまうせいでコンテキスト全体に対応する情報が継続に含まれていなかった。ここでは、全てのコンテキスト情報を明示化するため、さらに CPS 変換を施す。この変換によって現れる継続は `a -> a` 型である。この型 `a -> a` の名前を `k2` とする。変換したプログラムが図 8 である。

このプログラムは、図 7 のプログラムを機械的に CPS 変換すれば得られるもので、`OpCall` の第 3 引数も CPS 変換されている点にさえ注意すれば、特に説明を必要とする箇所はない。プログラム中には、次節で非関数化する部分にその旨、コメントが付してある。この変換により、すべての（serious な）関数呼び出しが末尾呼び出しとなり、コンテキスト情報はふたつの継続ですべて表現される。

4.3 非関数化

CPS 変換ですべてのコンテキスト情報がふたつの継続に集約された。ここでは、CPS 変換したことにより新たに現れた `a -> a` 型の関数を非関数化してデータ構造に変換する。非関数化によって型 `k2` の定義は図 9 に、インタプリタは図 10 に変換される。

この非関数化によって、引数 `k` と引数 `k2` からコンテキスト全体の情報が得られるようになった。ここで、得られたコンテキストの情報を整理しておこう。`k` はハンドラ内のコンテキストを示している。`FId` 以外はいずれの構成子も `k` を引数にとっているので、これは `FId` を空リストととらえれば評価文脈のリストと考えることができる。`k2` も同様に `h` と `k` が連なったリストと考えることができる。全体として「ハンドラに囲まれた評価文脈のリスト」のリストになっており、直感に合ったハンドラによって区切られたコンテキストが得られていることがわかる。

得られたコンテキストはごく自然なものだが、ハンドラの入る位置などは必ずしも自明ではない。プログラム変換を使うことで、algebraic effects の入った体系に沿ったコンテキストのデータ型が機械的に得られたことには一定の価値があると考えられる。

次に図 10 を見てみよう。ここでは、全ての時点でのコンテキスト情報がふたつの引数 `k` と `k2` によって表現されている。これはつまり、任意の時点で全体のコンテキスト情報が得られていることを意味している。この情報を使うと、簡約が行われるたびにプログラム全体を再構成でき、したがってステッパを作ることができる。

4.4 出力

4.3 節までの変換によって、コンテキストの情報を引数に保持するインタプリタ関数を得ることができた。この情報を用いて簡約前後のプログラムを出力するように、図 10 のインタプリタを変更するとステッパが得られる。具体的には、簡約が起こる部分でプログラム全体を再構成し表示するようにする。図 11 が表示を行う関数 `memo` を足した後の関数 `apply_in` と `apply_handler` である。他の関数は簡約している部分が無いので図 10 と変わらない。

関数 `memo : e -> e -> (k * k2) -> unit` は、簡約基とその簡約後の式と簡約時のコンテキストを受け取って、簡約前のプログラムと簡約後のプログラムをそれぞれ再構成して出力する。

`apply_in` では普通の関数呼び出しと継続呼び出しが `memo` されている。また、`apply_handler` ではハンドラが正常終了した場合とオペレーション呼び出しが起きた場合にそれぞれ `memo` 関数が挿

```
(* CPS インタプリタを非関数化して CPS 変換して非関数化した関数 *)
let rec eval (exp : e) (k : k) (k2 : k2) : a = match exp with
  | Val (v) -> apply_in k v k2
  | App (e1, e2) -> eval e2 (FApp2 (e1, k)) k2
  | Op (name, e) -> eval e (FOp (name, k)) k2
  | With (h, e) -> eval e FId (GHandle (h, k, k2))

(* handle 節内の継続を適用する関数 *)
and apply_in (k : k) (v : v) (k2 : k2) : a = match k with
  | FId -> apply_out k2 (Return v)
  | FApp2 (e1, k) -> let v2 = v in eval e1 (FApp1 (v2, k)) k2
  | FApp1 (v2, k) -> let v1 = v in (match v1 with
      | Fun (x, e) ->
        let reduct = subst e [(x, v2)] in
        eval reduct k k2
      | Cont (cont_value) ->
        (cont_value k) v2 k2
      | _ -> failwith "type error")
  | FOp (name, k) ->
    apply_out k2 (OpCall (name, v, (fun v -> fun k2' -> apply_in k v k2')))

(* 全体の継続を適用する関数 *)
and apply_out (k2 : k2) (a : a) : a = match k2 with
  | GId -> a
  | GHandle (h, k, k2) -> apply_handler k h a k2

(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler (k : k) (h : h) (a : a) (k2 : k2) : a = match a with
  | Return v -> (match h with {return = (x, e)} ->
      let reduct = subst e [(x, v)] in eval reduct k k2)
  | OpCall (name, v, va) ->
    (match search_op name h with
     | None ->
       apply_out k2 (OpCall (name, v,
                             (fun v -> fun k2' -> va v (GHandle (h, k, k2')))))
     | Some (x, y, e) ->
       let cont_value =
         Cont (fun k'' -> fun v -> fun k2 -> va v (GHandle (h, k'', k2))) in
       let reduct = subst e [(x, v); (y, cont_value)] in
       eval reduct k k2)

(* 初期継続を渡して実行を始める *)
let interpreter (e : e) : a = eval e FId GId
```

図 **10.** CPS インタプリタを非関数化して CPS 変換して非関数化したプログラム

```
(* handle 節内の継続を適用する関数 *)
and apply_in (k : k) (v : v) (k2 : k2) : a = match k with
  | FId -> apply_out k2 (Return v)
  | FApp2 (e1, k) -> let v2 = v in eval e1 (FApp1 (v2, k)) k2
  | FApp1 (v2, k) -> let v1 = v in (match v1 with
      | Fun (x, e) ->
        let redex = App (Val v1, Val v2) in  (* (fun x -> e) v2 *)
        let reduct = subst e [(x, v2)] in    (* e[v2/x] *)
        memo redex reduct (k, k2); eval reduct k k2
      | Cont (x, (k', k2'), cont_value) ->
        let redex = App (Val v1, Val v2) in  (* (fun x => k2'[k'[x]]) v2 *)
        let reduct = plug_all (Val v2) (k', k2') in  (* k2'[k'[v2]] *)
        memo redex reduct (k, k2); (cont_value k) v2 k2
      | _ -> failwith "type error")
  | FOp (name, k) ->
    apply_out k2 (OpCall (name, v, (k, GId),
                          (fun v -> fun k2' -> apply_in k v k2')))

(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler (k : k) (h : h) (a : a) (k2 : k2) : a = match a with
  | Return v ->
    (match h with {return = (x, e)} ->
       let redex = With (h, Val v) in  (* with {return x -> e} handle v *)
       let reduct = subst e [(x, v)] in  (* e[v/x] *)
       memo redex reduct (k, k2); eval reduct k k2)
  | OpCall (name, v, (k', k2'), vk2a) -> (match search_op name h with
      | None ->
        apply_out k2 (OpCall (name, v, (k', compose_k2 k2' h (k, GId)),
                              (fun v -> fun k2' -> vk2a v (GHandle (h, k, k2'))))))
      | Some (x, y, e) ->
        (* with {name(x; y) -> e} handle k2'[k'[name v]] *)
        let redex = With (h, plug_all (Op (name, Val v)) (k', k2')) in
        let cont_value =
          Cont (gen_var_name (), (k', compose_k2 k2' h (FId, GId)),
                (fun k'' -> fun v -> fun k2 -> vk2a v (GHandle (h, k'', k2)))) in
        (* e[v/x, (fun n => with {name(x; y) -> e} handle k2'[k'[n]]) /y *)
        let reduct = subst e [(x, v); (y, cont_value)] in
        memo redex reduct (k, k2);
        eval reduct k k2)
```

図 11. 変換の後、出力関数を足して得られるステッパ

```
(* コンテキスト k2_in の外側にフレーム GHandle (h, k_out, k2_out) を付加する *)
let rec compose_k2 (k2_in : k2) (h : h) ((k_out, k2_out) : k * k2) : k2 =
  match k2_in with
  | GId -> GHandle (h, k_out, k2_out)
  | GHandle (h', k', k2') -> GHandle (h', k', compose_k2 k2' h (k_out, k2_out))
```

図 **12.** 継続を外側に拡張する関数

```
(* 値 *)
type v = ...
       | Cont of string * (c * c2) * ((c * k) -> k) (* 継続 *)
(* handle 内の実行結果 *)
and a = Return of v                              (* 値になった *)
      | OpCall of string * v * (c * c2) * k  (* オペレーションが呼び出された *)
(* handle 内のメタ継続 *)
and k = v -> c2 -> a
(* handle 内のコンテキスト *)
and c = FId                (* [.] *)
      | FApp2 of e * c     (* [e [.]] *)
      | FApp1 of v * c     (* [[.] v] *)
      | FOp of string * c  (* [op [.]] *)
(* 全体のコンテキスト *)
and c2 = GId                     (* [.] *)
       | GHandle of h * c * c2  (* [[with h handle [.]]] *)
```

図 **13.** 継続の情報を保持するための言語やコンテキストの定義

入されている。また、オペレーション呼び出しが処理されず外側の `with handle` 文に制御を移す際には、図 12 に示される関数を使ってコンテキストの結合を行なっている。

4.5 非関数化されていないステッパ

前節で algebraic effects を持つ言語に対するステッパを作ることができた。しかし、前節で作ったステッパではコンテキストの情報が非関数化されていた。また、CPS 変換されているためふたつの継続を扱っており、もともとの CPS インタプリタとは形がかなり異なったものとなっている。しかし、一度、前節までで必要なコンテキストの情報がどのようなものかが判明すると、それを直接、もとの CPS インタプリタに加えてステッパを作ることができる。

もとの CPS インタプリタの型定義に必要なコンテキストの情報を加えた定義が図 13 になる。ここで、`c` と `c2` がそれぞれハンドラ内、全体のコンテキストの情報で、前節までの非関数化によって得られたものである。一方、`k` はもとからある高階の継続の型である。継続 `k` は、簡約ごとにプログラム全体を表示するので、必要なコンテキストの情報を新たに引数に取るようになっている。

このデータ定義を使って、もとの CPS インタプリタをステッパに変換したのが図 14 である。このインタプリタは、もとの CPS インタプリタにコンテキストの情報として引数 `c` と `c2` を加え、簡約ごとにプログラムを再構成し、ステップ表示するようにしたものである。一度、必要なコンテキストの情報が特定されると、algebraic effects のように非自明な言語構文が入っていても、直接、ステッパを作ることができるようになる。

```
(* CPS ステッパ *)
let rec eval (exp : e) ((c, k) : c * k) (c2 : c2) : a = match exp with
  | Val (v) -> k v c2
  | App (e1, e2) -> eval e2 (FApp2 (e1, c), (fun v2 c2 ->
    eval e1 (FApp1 (v2, c), (fun v1 c2 -> match v1 with
      | Fun (x, e) ->
        let redex = App (Val v1, Val v2) in  (* (fun x -> e) v2 *)
        let reduct = subst e [(x, v2)] in  (* e[v2/x] *)
        memo redex reduct (c, c2); eval reduct (c, k) c2
      | Cont (x, (c', c2'), cont_value) ->
        let redex = App (Val v1, Val v2) in  (* (fun x => c2[c[x]]) v2 *)
        let reduct = plug_all (Val v2) (c', c2') in  (* c2[c[v2]] *)
        memo redex reduct (c, c2); (cont_value (c, k)) v2 c2
      | _ -> failwith "type error")) c2)) c2
  | Op (name, e) -> eval e (FOp (name, c), (fun v c2 ->
    OpCall (name, v, (c, GId), (fun v c2' -> k v c2')))) c2
  | With (h, e) ->
    let a = eval e (FId, (fun v c2 -> Return v)) (GHandle (h, c, c2)) in
    apply_handler (c, k) h a c2

(* handle 節内の実行結果をハンドラで処理する関数 *)
and apply_handler ((c, k) : c * k) (h : h) (a : a) (c2 : c2) : a = match a with
  | Return v -> (match h with {return = (x, e)} ->
     let redex = With (h, Val v) in  (* with {return x -> e} handle v *)
     let reduct = subst e [(x, v)] in  (* e[v/x] *)
     memo redex reduct (c, c2); eval reduct (c, k) c2)
  | OpCall (name, v, (c', c2'), k') ->
    (match search_op name h with
      | None -> OpCall (name, v, (c', compose_c2 c2' h (c, GId)),
        (fun v' c2'' -> let a' = k' v' (GHandle (h, c, c2'')) in
          apply_handler (c, k) h a' c2''))
      | Some (x, y, e) ->
        (* with {name(x; y) -> e} handle c2'[c'[name v]] *)
        let redex = With (h, plug_all (Op (name, Val v)) (c', c2')) in
        let cont_value = Cont (gen_var_name (),
          (c', compose_c2 c2' h (FId, GId)), (fun (c'', k'') v' c2'' ->
            let a' = k' v' (GHandle (h, c'', c2'')) in
            apply_handler (c'', k'') h a' c2)) in
       (* e[v/x, (fun n => with {name(x; y) -> e} handle c2'[c'[y]])/y *)
       let reduct = subst e [(x, v); (y, cont_value)] in
       memo redex reduct (c, c2); eval reduct (c, k) c2)

let stepper (e : e) : a = eval e (FId, (fun v c2 -> Return v)) GId
```

図 14. 変換の結果得られたステッパ

5　他の言語への対応

3 節で示した algebraic effects を含む言語の CPS インタプリタをステッパにするには、非関数化、CPS 変換、非関数化が必要だったが、他のいくつかの言語についても同様にインタプリタを変換することでステッパを導出することを試みた。それぞれの言語のステッパ導出について説明する。

5.1　型無し λ 計算

型無し λ 計算の DS インタプリタは、CPS 変換して非関数化したら全てのコンテキストを引数に保持するインタプリタになり、出力関数を入れるのみでステッパを作ることができた。これは、継続を区切って一部を捨てたり束縛したりするという操作が無いためだと考えられる。

5.2　try-with

try-with は、algebraic effects が限定継続を変数に束縛するのと違って、例外が起こされたときに限定継続を捨てるという機能である。よって継続を表す値は現れないので、インタプリタを CPS で書く必要は無い。Direct style でインタプリタを書いた場合、最初に CPS 変換をすることで、CPS インタプリタと同様の変換によってステッパが導出できた。最初から CPS インタプリタを書いていれば 4 節と同様の手順になる。

5.3　shift/reset

shift/reset は algebraic effects と同様に限定継続を変数に束縛して利用することができる機能である。4 節で行ったのと全く同様に、CPS インタプリタを非関数化、CPS 変換、非関数化したらコンテキストが表れ、ステッパが得られた。

5.4　Multicore OCaml

Multicore OCaml は、OCaml の構文に algebraic effects を追加した構文を持つ。我々は 4 節で得られたステッパをもとにして、Multicore OCaml の algebraic effects を含む一部の構文を対象にしたステッパの実装を目指している。以下に Multicore OCaml の algebraic effects を含むプログラムのステップ表示の例を示す。

```
Input :  (try ((perform (E 1)) + (2 + (perform (E 4))))
          with | effect (E n) k -> (continue k n))
Step 1:  (continue (fun x => (try ((perform (E 1)) + (2 + x))
                              with | effect (E n) k -> (continue k n))) 4)
Step 2:  (try ((perform (E 1)) + (2 + 4))
          with | effect (E n) k -> (continue k n))
Step 3:  (try ((perform (E 1)) + 6)
          with | effect (E n) k -> (continue k n))
Step 4:  (continue (fun y => (try (y + 6)
                              with | effect (E n) k -> (continue k n))) 1)
Step 5:  (try (1 + 6) with | effect (E n) k -> (continue k n))
Step 6:  (try 7 with | effect (E n) k -> (continue k n))
Step 7:  7
```

Multicore OCaml の「エフェクト」は 3 節で定義した言語の algebraic effects のオペレーションとほとんど同じものだが、以下のような違いがある。

- (ハンドルの構文) `with handler {return x -> er, op1(x; k) -> e1, ...} handle e` は、Multicore OCaml では `match e with x -> er | effect (Op1 x) k -> e1 | ...` と書ける。
- (宣言) `effect E : t1 -> t2` と書くことで、`t1` 型の引数をとって `t2` 型を返すエフェクト `E` が宣言できる。宣言していないエフェクトは使用できない。
- (実行) エフェクトを呼び出す際には、 `perform (E e)` というように関数 `perform` に渡す。
- (継続の適用) 継続 `k` に式 `e` を渡して実行を再開するには `continue k v` と書く。
- (継続の適用の制限) 同じ継続は 1 度しか適用できない。

このような構文の違いはあるものの、継続が one-shot であることを除いて簡約のされかたは 3 節で定めた言語のインタプリタと同様なので、インタプリタ関数を用意できれば変換によってステッパが導出できると考えられる。

6 関連研究

ステッパはもともと Racket に対して作られた。これは Clements ら [2] が設計したもので、スタックに continuation mark と呼ばれるマークを付けることで現在の評価文脈を再構成できるようにしている。しかし、例外処理などの構文には対応していない。我々は引数にコンテキストの情報を渡すことで、OCaml に対するステッパを設計した [3]。このステッパは OCaml の例外処理にも対応している。以上のステッパはいずれも big-step のインタプリタに手を加える形で作られている。一方、Whitington と Ridge [10] は small-step のインタプリタを直接、書くことで OCaml に対するステッパを実装した。しかし、いずれのステッパも algebraic effects には対応していない。

algebraic effects に対する意味論は、これまで small-step の意味論 [6] あるいは CPS による意味論 [5] が与えられて来た。しかし、後者は入力言語が A-正規形であることを仮定しているのに加え、継続がフレームのリストで与えられており、通常の CPS インタプリタにはなっていない。本論文で与えた CPS インタプリタは、入力言語を制限しておらず、また継続も普通の一引数関数となっている。

上記以外の algebraic effects に関する研究としては、ハンドラの挙動が異なる shallow ハンドラの研究 [4] や algebraic effects を含むプログラムに関する論理関係を定義する研究 [1] などがあげられる。本論文で扱っているハンドラは従来の deep ハンドラである。また、論理関係などは考慮していない。

7 まとめと今後の課題

ステッパを実装するためには、コンテキストの情報を保持しながら部分式を再帰的に実行するインタプリタを作ればよい。以前の研究 [3] では言語ごとにコンテキストを表すデータ型を考えた上でインタプリタに実行の流れに従った新しい引数を付け足す作業が必要だったが、本研究では通常のインタプリタを CPS 変換および非関数化するという機械的な操作のみでコンテキストの型およびコンテキストの情報を保持するインタプリタ関数を導出した。

その方法で、継続を明示的に扱える algebraic effects を含む言語に対するステッパを実装し、それをもとにして algebraic effects を含む言語 multicore OCaml の一部の構文のステッパも実装した。また、他の例外処理機能である try-with や shift/reset を含む言語についても同様の変換ができることを確認した。

今後は、より多くの言語機能についてステッパを導出する方法を探求していきたい。

参考文献

[1] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: Relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.*, Vol. 2, No. POPL, December 2017.

[2] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *European symposium on programming*, pp. 320–334. Springer, 2001.

[3] Tsukino Furukawa, Youyou Cong, and Kenichi Asai. Stepping OCaml. In Proceedings Seventh International Workshop on *Trends in Functional Programming in Education,* Chalmers University, Gothenburg, Sweden, 14th June 2018, Vol. 295 of *Electronic Proceedings in Theoretical Computer Science*, pp. 17–34, 2019.

[4] Daniel Hillerström and Sam Lindley. Shallow effect handlers. In Sukyoung Ryu, editor, *Programming Languages and Systems*, pp. 415–435, Cham, 2018. Springer International Publishing.

[5] Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. Continuation passing style for effect handlers. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), pp. 18:1–18:19, 9 2017.

[6] 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. Association for Computing Machinery.

[7] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, Vol. 1, No. 2, pp. 125–159, 1975.

[8] Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, Vol. 319, pp. 19–35, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

[9] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, Vol. 11, No. 4, pp. 363–397, Dec 1998.

[10] John Whitington and Tom Ridge. Direct interpretation of functional programs for debugging. In Sam Lindley and Gabriel Scherer, editors, Proceedings *ML Family / OCaml Users and Developers workshops,* Oxford, UK, 7th September 2017, Vol. 294 of *Electronic Proceedings in Theoretical Computer Science*, pp. 41–73, 2019.