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、ステッパを実装する方法を示す。

実際に型無しλ計算と algebraic effects から成る言語について実装方法を示し、それをもとにして実装した Multicore OCaml を対象としたステッパを紹介する。

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。（後で追加。）

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

```
(* 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

```
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

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 言語とインタプリタの定義

この節では、型無し λ 計算と algebraic effects からなる言語とそのインタプリタを定義する。

3.1 algebraic effects

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、実行の過程で現れる。

意味論については 3.3 節で厳密に扱うが、例えばプログラム `(with handler {return x -> (fun a -> x), op(x, k) -> k x} handle ((op (fun b -> b)) 2)) 3` は以下のように実行される。

right-to-left て評価するとして、まず関数適用の引数部分 `3` を実行し、値 `3` を得る。次に関数部分の `with ... handle ...` を実行する。`with ... handle ...` ではまず `handle` 以降の `((op (fun b -> b)) 2)` を実行する。これは関数適用なのでまず引数部分の `2` を実行し、値 `2` を得る。次に関数部分 `(op (fun b -> b))` はオペレーション呼び出しであり、まず引数 `(fun b -> b)` を実行する。値 `(fun b -> b)` を得たら、`(op (fun b -> b))` を囲んでいる `with ... handle ...` を参照して、そこにオペレーション `op` が定義されているかどうかを見る。ここでは `op(x, k) -> k x` と定義されている。すると、`with ... handle ...` の部分が `(k x)[(fun b -> b)/x, (fun y => (with handler {return x -> (fun a -> x), op(x, k) -> k x} handle (y 2)))/k` に簡約され、次はこの式の実行に移る。この `k` に代入する値は、`with ... handle ...` 式までの限定継続である。引数部分と関数部分をそれぞれ実行し、これらは元々値になっているのでそのままの値が返る。そして `(fun b -> b)` に継続 `(fun y => (with ... handle (y 2)))` を適用すると `(with ... handle ((fun b -> b) 2))` になるが、この適用は継続の適用なので、`fun b -> b` および `2` は既に実行した後だということが分かっており、次には `(fun b -> b) 2` の関数適用をする。これは `2` になり、ここで `handle ...` 内が値となった。すると今度は `with ... handle ...` がハンドラの return 節の `fun a -> x` に簡約される。このとき `x` は `handle ...` 内の値の `2` である。よってここでプログラム全体は `(fun b -> 2) 3` になっており、この関数適用をして最終結果は `2` になる。

```
(* 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;                          (* handler {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

もし呼び出されたオペレーションがハンドラで定義されていなければさらに外のハンドラを参照して、その時の継続値は定義が見つかったハンドラまでの限定継続である。見つからなかった場合はエラーを起こす。

3.2 構文の定義

対象言語の OCaml による定義を図 4 に示す。式の型 `e` で表されるものが対象言語のプログラムである。

継続の型 `k` は、インタプリタ関数自体の継続を表す。

3.3 CPS インタプリタ

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 `handler {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 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 行目で再帰呼び出しの際に継続に `(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。

そして `handle` 節内を実行した結果を表すのが `a` 型で、 `handle` 節を実行した結果値 `v` になったことを `Return v` と表し、値になる前にオペレーション呼び出しが行われたことを `OpCall (name,`

```
(* 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 (k') ->
              (k' 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, k))  (* 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)} ->  (* handler {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'))  (* FCall に変換される関数 *)
     | 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  (* FCall に変換される関数 *)
       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

v, k) で表す。この結果とハンドラを受け取ってハンドラの処理をするのが関数 apply_handler である。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] 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: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
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, k'/y] を実行

関数 apply_handler の下から 5 行目に (fun k'' -> fun v -> ...) という関数がある。このうち fun v -> ... の部分は k' の後に k'' を行うという継続であり、継続を実際に適用する際に引数に適用するものである。k'' はこの継続を適用する時点で関数 eval が引数に保持している継続である。なぜ k'' をとる関数になっているのかというと、例えば (fun a -> a) ((fun y => with h handle (fun b -> b) y) 1) の継続の適用をするとき、継続値は「受け取った値に fun b -> b を適用してそれをハンドラ h で処理する」というメタ継続を持っているが、さらにその後には「そこに fun a -> a を適用する」という継続を合成したいのに、継続を capture する時点では fun a -> a についての情報は見えないからである。関数 eval 内の (k' k) v2 という部分で、この関数 (fun k'' -> fun v -> ...) に継続の適用の後の継続 k を渡すことで合成された継続を作り、そこに値を渡している。

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) を変換することで、コンテキストの情報を保持するインタプリタを得る方法を示す。用いるプログラム変換は非関数化と CPS 変換の 2 種類である。これらの変換はプログラムの動作を変えないので、変換の結果得られるインタプリタと図 5 のインタプリタは、同じ引数 e に対して同じ値を返す。

4.1 非関数化

まず、図 5 のプログラムの k 型の λ 式を非関数化する。非関数化は以下の手順によって行われる。

1. 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。
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。
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。

図 5 のインタプリタの k 型すなわち v -> a 型の λ 式を非関数化すると、型 k の定義は図 6 のようにヴァリアント型になり、インタプリタは図 7 に書き換わる。

非関数化したことで継続 k がコンストラクタとして表されるようになったので、継続の構造を参照することや、継続を部分的に書き換えることが可能になった。具体的な k の構造の例を示す。図 7 の関数 stepper に入力プログラム ((fun a -> a) (with handler {return x -> x, a(x; k) -> x} handle ((fun b -> b) (a (fun c -> c))) (fun d -> d))) を表す構文木を渡して実行を始めた場合、(a (fun c -> c)) を関数 eval に渡して実行を始める際の継続は FApp2 (Fun ("b",

```
and k = FId                    (* [.] *)
      | FApp2 of e * k         (* [e [.]] *)
      | FApp1 of v * k         (* [[.] v] *)
      | FOp of string * k      (* [op [.]] *)
      | FCall of k * h * k     (* [with h handle [[.]]] *)
```

図 6. 非関数化後の継続の型

Var "b"), FApp1 (Fun ("d", Var "d"))) である。これは式 (a (fun c -> c)) のコンテキスト ((fun a -> a) (with handler {return x -> x, a(x; k) -> x} handle ((fun b -> b) [.]) (fun d -> d))) のうち、handle の内側に対応している。handle から外側が継続に含まれないのは、関数 eval で with h handle e の e の実行の再帰呼び出し時に初期継続を表す FId を渡しているためである。コンテキスト全体に対応した継続を得るために、この後の変換をさらに施す。

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なかったので、全ての継続を引数に持つようにするため、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。

4.3 非関数化

CPS 変換したことにより新たに現れた a -> a 型の匿名関数を非関数化する。非関数化によって型 k2 の定義は図 9 に、ステッパ関数は図 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。?? 節で示した例について比較する。stepper に入力プログラム ((fun a -> a) (with handler {return x -> x, a(x; k) -> x} handle ((fun b -> b) (a (fun c -> c))) (fun d -> d))) を表す構文木を渡して実行を始めた場合、(a (fun c -> c)) を表す構文木を関数 eval に渡して実行を始める際の継続 k は ?? 節と同様に FApp2 (Fun ("b", Var "b"), FApp1 (Fun ("d", Var "d"))) である。そして継続 k2 は GHandle () 具体的な k の構造の例を示す。図 7 の関数 stepper に入力プログラム ((fun a -> a) (with handler {return x -> x, a(x; k) -> x} handle ((fun b -> b) (a (fun c -> c))) (fun d -> d))) を表す構文木を渡して実行を始めた場合、(a (fun c -> c)) を関数 eval に渡して実行を始める際の継続は FApp2 (Fun ("b", Var "b"), FApp1 (Fun ("d", Var "d"))) である。これは式 (a (fun c -> c)) のコンテキスト ((fun a -> a) (with handler {return x -> x, a(x; k) -> x} handle ((fun b -> b) [.]) (fun d -> d))) のうち、handle の内側に対応している。handle から外側が継続に含まれないのは、関数 eval で with h handle e の e の実行の再帰呼び出し時に初期継続を表す FId を渡しているためである。コンテキスト全体に対応した継続を得るために、この後の変換をさらに施す。

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 のインタプリタの簡約が起こる部分に副作用を足すとステッパが得られる。図 11 が副作用を足した後の関数 apply_in と apply_handler であり、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。

```
(* 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 (k') ->
       apply_in (k' k) v2
     | _ -> failwith "type error")
  | FOp (name, k) -> OpCall (name, v, k)  (* この handle 節の実行結果は OpCall *)
  | FCall (k'', h, k') ->  (* k''[with h handle k'[v]] *)
    let a = apply_in k' v in  (* handle 節までの継続を適用 *)
    apply_handler k'' h a  (* a をハンドラ h で処理して、その後 k'' を適用 *)

(* 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, k') ->
    (match search_op name h with
     | None -> OpCall (name, v, FCall (k, h, k'))
     | Some (x, y, e) ->
       let cont_value = Cont (fun k -> FCall (k, h, k')) 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 stepper (e : e) : a = eval e FId
```

図 **7.** CPS インタプリタを非関数化したプログラム

```
(* 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 に変換される関数 *)

(* 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 (k') ->
       apply_in (k' k) v2 k2
     | _ -> failwith "type error")
  | FOp (name, k) -> k2 (OpCall (name, v, k))
  | FCall (k'', h, k') ->
    apply_in k' v (fun a -> apply_handler k'' h a k2) (* GHandle に変換される関
数 *)

(* 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, k') ->
    (match search_op name h with
     | None ->
       k2 (OpCall (name, v, FCall (k, h, k'))) (* 外の継続を適用 *)
     | Some (x, y, e) ->
       let cont_value = Cont (fun k -> FCall (k, h, k')) 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 stepper (e : e) : a = eval e FId (fun a -> a)  (* GId に変換される関数 *)
```

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

```
type k2 = GId
        | GHandle of k * h * k2
```

図 9. 2 回目の非関数化後の継続の型

ステップ表示では継続値の内容も見えるようにしたいので、継続を文字列で表す必要がある。ここでは継続を関数 `fun x -> e` のように `fun x => e` と表すこととする。すると各継続が仮引数名を持つ必要があるので構文木に追加する（図 12）。新しい継続値を作る時に、プログラム中で使われていない変数名を生成する関数 `gen_var_name` を利用して仮引数を定めている。

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。

図 11 で `memo` の引数に渡している `redex` および `reduct` は、以下に示すこの言語の small-step の実行規則の簡約前後の式に対応する。

(h = `handler {return x -> e0, op1(x; k) -> e1, ..., opn(x; k) -> en}`)

$$\frac{}{\texttt{(fun x -> e) v} \Downarrow \texttt{e[v/x]}}\ (\mathsf{AppFun}) \qquad \frac{}{\texttt{(fun x => F[x]) v} \Downarrow \texttt{F[v]}}\ (\mathsf{AppCont})$$

$$\frac{}{\texttt{with h handle v} \Downarrow \texttt{e0[v/x]}}\ (\mathsf{Return}) \qquad \frac{?}{\texttt{with h handle e} \Downarrow \texttt{e0[v/x]}}\ (?)$$

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 Multicore OCaml

Multicore OCaml は、OCaml の構文に algebraic effects を追加した構文を持つ。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を実装した。3 節で定義した言語では `with handler {return x -> er, op1(x; k) -> e1, ..., opn(x; k) -> en} handle e` という構文でオペレーションを定義していたのに対し、Multicore OCaml では `try e with effect (op1 x) k -> e1 | ... | effect (opn x) k -> en` となる。この `with` 以下のパターンの中には例外のパターンも含めることができる。また、オペレーション呼び出しは `perform (opi e)` で行い、継続 `k` に値 `v` を渡して実行を再開するには `continue k v` と書く。このような構文の違いはあるものの、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。

5.2 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、非関数化して出力関数を挿入したらステッパが得られた。

```
(* 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 (k, h, 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 (k') ->
       apply_in (k' k) v2 k2
     | _ -> failwith "type error")
  | FOp (name, k) -> apply_out k2 (OpCall (name, v, k))
  | FCall (k_last, h, k') -> apply_in k' v (GHandle (k_last, h, 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 (k, h, 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_last : 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_last k2)
  | OpCall (name, v, k') -> (match search_op name h with
     | None -> apply_out k2 (OpCall (name, v, FCall (k_last, h, k')))
     | Some (x, k, e) ->
       let cont_value = Cont (fun k_last -> FCall (k_last, h, k')) in
       let reduct = subst e [(x, v); (k, cont_value)] in
       eval reduct k_last k2)

(* 初期継続を渡して実行を始める *)
let stepper (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') ->
       let redex = App (Val v1, Val v2) in               (* k' v2 *)
       let reduct = plug_in_handle (Val v2) (k' FId) in  (* k'[v2] *)
       memo redex reduct (k, k2);
       apply_in (k' k) v2 k2
     | _ -> failwith "type error")
  | FOp (name, k) -> apply_out k2 (OpCall (name, v, k))
  | FCall (k_last, h, k') ->
    apply_in k' v (GHandle (k_last, h, 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 handler{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') ->
    (match search_op name h with
     | None -> apply_out k2 (OpCall (name, v, FCall (k, h, k')))
     | Some (x, y, e) ->
       let redex = (* with handler {name(x; y) -> e} handle k'[(name v)] *)
         With (h, plug_in_handle (Op (name, Val v)) k') in
       let new_var = gen_var_name () in
       let cont_value =
         Cont (new_var, fun k -> FCall (k, h, k')) in
       let reduct = (* e[v/x, k[with h handle k']/y] *)
         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

```
type v = Var of string
       | Fun of string * e
       | Cont of e * (k -> k)
```

図 **12.** 継続を文字列で表すために変更した値の型

5.3 try-with

try-with は、algebraic effects や shift/reset 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。

5.4 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。

6 関連研究

[6] [4] [5] [1] [8]

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, p. 145158, 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. Open Publishing Association, 2019.