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

継続を操作するプログラムの挙動を理解するためには、プログラムがどのように実行されていくのかを詳細に追う必要がある。その継続を得た時点での実行の状態が分からなければどのような継続なのかが分からないからである。

そのような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。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。

そこで我々は、プログラムが代数的に書き換わる様子を 1 ステップずつ表示するツールである代数的ステッパを、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、例えば OCaml のプログラム `let a = 1 + 2 in 4 + a` を入力されると、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
```

ステッパはプログラムを評価するプログラムなのでインタプリタの一種である。古川ら (2019)[1] は、例外処理のための構文 try-with を含む言語について、通常のインタプリタ関数に出力機能を

```
(* 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
  | App (e1, e2) ->
    (let v2 = eval e2 in
     let v1 = eval e1 in
     let reduct = match v1 with
       | Fun (x, e) -> subst e [(x, v2)]  (* e[v2/x] *)
       | _ -> failwith "type error" in
     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

追加することでステッパを実装した。インタプリタは部分式を再帰的に実行するが、その先で簡約があった場合に式全体を出力するために、部分式を再帰的に実行している時にそのコンテキストの情報を得ることが必要になる。古川ら (2019)[1] は評価順序をもとにコンテキストの構造を考えてコンテキストを表すデータ型を定義していたが、本研究ではインタプリタ関数に CPS 変換 [2] および非関数化 [3] という変換を施すことでコンテキストの情報を引数に持つインタプリタ関数を導出し、それを用いてプログラム全体を再構成して出力する方法を提案する。そして実際に algebraic effects を含む言語についてインタプリタ関数を定義し、それを変換することによってステッパ関数を導出する過程について説明する。

2 ステッパの実装におけるコンテキスト

ステッパは簡約のたびに簡約前後のプログラムを出力するインタプリタである。例えばプログラム `((fun a -> a) ((fun b -> b) (fun c -> c)))` を入力した場合、以下のように出力したい。

```
((fun a -> a) ((fun b -> b) (fun c -> c)))
((fun a -> a) (fun c -> c))

((fun a -> a) (fun c -> c))
(fun c -> c)
```

ステッパはインタプリタにステップ出力機能が加わったものなので、通常のインタプリタ関数に書き足すことで実装できる。図 1 に OCaml による型無し λ 計算の定義とインタプリタの実装を示す。関数 `subst : e -> (string * v) list -> e` は代入のための関数であり、`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。

```
(* 不十分なステップ出力をするインタプリタ *)
let rec eval (exp : e) : v = match exp with
  | Val (v) -> v
  | App (e1, e2) ->
    (let v2 = eval e2 in
     let v1 = eval e1 in
     let redex = App (Val v1, Val v2) in  (* 簡約前の式 *)
     let reduct = match v1 with
       | Fun (x, e) -> subst e [(x, v2)]
       | _ -> failwith "type error" in
     memo redex reduct;                   (* 出力 *)
     eval reduct)
```

図 2. 出力関数を挿入した λ 計算のインタプリタ

簡約時に簡約前後の式を出力するために、図 1 の関数 `eval` に、式を 2 つ受け取ってそれぞれ出力する関数 `memo` の呼び出しを挿入すると図 2 のようになる。図 1 との違いはコメントの付いた 2 つの行が増えたのみである。

図 2 の関数 `eval` にプログラム `((fun a -> a) ((fun b -> b) (fun c -> c)))` を表す構文木 (`e` 型) を渡すと、以下のような文字列が出力される。

```
((fun b -> b) (fun c -> c))
(fun c -> c)

((fun a -> a) (fun c -> c))
(fun c -> c)
```

これは、期待した出力と比較すると、最初の簡約を表すステップ `((fun b -> b) (fun c -> c))` $\rightsquigarrow$ `(fun c -> c)` の外側の式、すなわちコンテキストの `((fun a -> a) [.])` が不足している。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。図 2 の関数 `eval` では、実行中の式の構文木は引数として渡されるのに対して、コンテキストの情報は得られない。

そこで、著者らは以前 [**?**] インタプリタ関数にコンテキスト情報のための引数を増やすことで式全体を再構成できるようにした。図 1 のインタプリタにその変更を施すと、図 3 のようになる。ここでは関数 `memo` は、簡約前の式、簡約後の式、コンテキスト情報の 3 つの引数をとり、コンテキスト情報を利用して簡約前後の式にそれぞれコンテキストを付加することで簡約前後の式全体を得て出力する。

図 3 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。以前の研究 [1] ではこの方法を用いて try-with 構文を含む言語のステッパを実装することに成功した。しかし、try-with のような制御構文を含む言語では、コンテキストを区切ってある区間のコンテキストを一度に捨てるなどの操作が必要になるため、コンテキストの構造が一次元的でなくなり、複雑なデータ構造の定義が必要になる可能性がある [1] 。

ところが、図 3 の `c` 型の定義を見ると、各コンストラクタはインタプリタ関数の「どの再帰呼び出しか」に対応している。コンテキストの型は評価順序によって定まるものであり、評価順序はインタプリタ関数で定義されているので、コンテキストを表すデータ型の定義はインタプリタ関数か

```
(* 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 3. 型無し λ 計算に対するステッパ

ら導出できるものであると考えられる。次節以降ではその導出方法の 1 つを提案する。

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 対象言語の構文

対象言語の OCaml による定義を図 4 に示す。値のコンストラクタ `Cont` は継続を表すコンストラクタであり、入力プログラムに含まれることはないが、ステップ実行の過程で現れる。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。

3.2 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。

```
(* 値 *)
type v = Var of string                (* x *)
       | Fun of string * e            (* fun x -> e *)
       | Cont of string * (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 *)
(* 継続 *)
and k = v -> a
(* 実行結果 *)
and a = Return of v                (* 正常終了 *)
      | OpCall of string * v * k   (* オペレーション呼び出し *)
```

図 4. 対象言語の定義

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 のプログラムを非関数化する。具体的には以下を施す。

1. k 型すなわち v -> a 型の匿名関数全てを、関数内に現れる全ての自由変数を引数に持つコンストラクタに分類し、継続の型 k をそれらのコンストラクタから構成されるヴァリアント型に変更する (図 6 のようになる)。
2. v -> a 型の匿名関数全てを、該当するコンストラクタに置き換える
3. 継続に引数を渡している部分 k v を apply_k k v に置き換え、意味が変わらないように関数 apply_k を定義する

変換後のプログラムを図 7 に示す。

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", Var "b"), FApp1 (Fun ("d", Var "d"))) である。これは式 (a (fun c -> c)) のコンテキス

```
(* 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 ->
        eval e1 (fun v1 -> match v1 with
            | Fun (x, e) ->
              let reduct = subst e [(x, v2)] in
              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))
  | With (h, e) ->
    let a = eval e (fun v -> Return v) in
    apply_handler k h a

(* ハンドラを処理する関数 *)
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, ...} *)
       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 *)
    (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)
```

図 5. 継続渡し形式で書かれたインタプリタ

```
and k = FId
      | FApp2 of e * k
      | FApp1 of v * k
      | FOp of string * k
      | FCall of k * h * k
```

図 6. 非関数化後の継続の型

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。

```
let rec eval (exp : e) (k : k) : a = match exp with
  | Val (v) -> apply_in k v
  | 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
    apply_handler k h a

and apply_in (k : k) (v : v) : a = match k with
  | FId -> Return v
  | 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)
  | FCall (k_last, h, k') ->
    let a = apply_in k' v in
    apply_handler k_last h a

and apply_handler (k_last : 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_last)
  | OpCall (name, v, k') ->
    (match search_op name h with
     | None ->
       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)

let stepper (e : e) : a = eval e FId
```

図 7. 非関数化した CPS インタプリタ

```
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)

and apply_in (k : k) (v : v) (k2 : k2) : a = match k with
  | FId -> 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) -> k2 (OpCall (name, v, k))
  | FCall (k_last, h, k') ->
    apply_in k' v (fun a -> apply_handler k_last h a k2)

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 ->
       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)

let stepper (e : e) : a = eval e FId (fun a -> a)
```

図 **8.** CPS 変換した非関数化した CPS インタプリタ

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

図 **9.** 2 回目の非関数化後の継続の型

```
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))

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))

and apply_out (k2 : k2) (a : a) : a = match k2 with
  | GId -> a
  | GHandle (k, h, k2) ->
    apply_handler k h a k2

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)

let stepper (e : e) : a = eval e FId GId
```

図 10. 非関数化して CPS 変換して非関数化した CPS インタプリタ

```
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
       let reduct = subst e [(x, v2)] in
       memo redex reduct (k, k2);
       eval reduct k k2
     | Cont (x, k') ->
       let redex = App (Val v1, Val v2) in
       let reduct = plug_in_handle (Val v2) (k' FId) in
       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))

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 redex = With (h, Val v) in
       let reduct = (subst e [(x, v)]) in
       memo redex reduct (k_last, k2);
       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 redex = 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_last -> FCall (k_last, h, k')) in
       let reduct = subst e [(x, v); (k, cont_value)] in
       memo redex reduct (k_last, k2);
       eval reduct k_last 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

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。

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

？

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 [1] 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] Tsukino Furukawa, Youyou Cong, and Kenichi Asai. Stepping OCaml. Vol. 295, pp. 17–34. Open Publishing Association, Jun 2019.

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

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