限定継続を含む仮想機械導出のためのプログラム変換

木谷有沙　　浅井健一
お茶の水女子大学　　お茶の水女子大学
arisa@pllab.is.ocha.ac.jp　　asai@is.ocha.ac.jp

概 要

本研究では、限定継続処理のオペレータ shift / reset に関して、正当性の保証された処理系を与えることを目的としている。その手法として、CPS のインタプリタに定義に基づいて shift / reset を追加した評価器に対し、変換前後の等価性が保証される変換のみを用いて仮想機械を導出、遷移規則を得る。その際、CPS 変換、非関数化、カリー化等の先行研究において妥当性が保証されている変換に加え、より仮想機械の動作に近づけるために、検証が不十分と思われる変換も用いたため、その変換前後の等価性の証明もあわせて行う。現段階では、インタプリタにスタックを導入する変換、環境をスタックに退避する変換についての検証が済んでおり、CPS のインタープリタに妥当なプログラム変換を施すことにより、スタックへの環境退避を行うインタプリタを取得できることが分かっている。

1　はじめに

継続とは、プログラム実行中のある時点における、残りの計算のことである。継続を取り出したりコピーしたりすることは、プログラムの制御フローを編集することに相当し、強力な大域ジャンプのようなものとも考えられる。しかし、この操作は継続を取り出した部分にのみしかジャンプを許しておらず、また型システム [3, 5] も存在しており、無条件 goto 文よりも安全であるといえる。この継続処理を使って実装が容易になる例として、例外処理がよく挙げられる。一般的なプログラミング言語においては、try/catch や raise 等の例外処理専用の機能が特別に用意され、ユーザはこれを利用するようになっている。しかし継続処理ができるプログラミング言語においては、前述のような例外処理用の機能が用意されていなくても、「現在の継続を捨てる」と書けば例外処理に相当する動作を実現できる。

このような継続処理を可能にするために、call/cc [17] や control/prompt [8], shift/reset [5] などの継続処理オペレータが存在する。このうち control/prompt, shift/reset は限定継続であり、call/cc では継続の取り出される範囲が現在以降の全てであるのに対し、限定継続では取り出される継続の範囲をユーザが指定できる。

しかし現在、限定継続処理をユーザが明示的に扱えるプログラミング言語は多くない。継続渡し形式 (continuation-passing style : CPS) のインタプリタを書くか、call/cc ライブラリを利用すれば間接実装は可能 [9] であるが、限定継続処理そのものの性能を評価したい場合等は、ユーザが機械語で限定継続処理を実装せねばならない。

そこで本研究では、機械語での限定継続処理実装のための、正当性の保証のあるモデルを提示し、限定継続を利用したプログラミングの推奨への第一歩としたい。

その手法として、CPS のインタプリタに定義に基づいて shift/reset を追加した評価器に対し、変換前後の評価器の等価性が保証された変換のみを用いてより低レベルな評価器を導出していく。

現段階では、インタプリタに CPS 変換、非関数化、線形リスト化、スタック導入、環境退避変換を施すことにより、SECD マシン [12] に類似した抽象機械を取得しており、shift/reset によるスタックコピー及びコピー範囲の限定をモデル化できている。

関連研究

Danvy ら [7] は、SECD マシンを高階関数を使うよう書き換え、スタックを除去し、direct style に変換することで、λ 計算のインタプリタを得ている。それに伴い、λ 計算インタプリタに CPS 変換、非関数化を施したものが SECD マシンと bisimulation 等価であると述べられている。Ager ら [2] も λ 計算インタプリタに対し CPS 変換、非関数化を施すことにより CEK マシンを得ている。この手法での shift/reset を含む抽象機械の導出そのものは Danvy ら自身によるものが既に存在する [4] が、Danvy らが SECD マシンや CEK マシン等の既存の抽象機械に近い形式の抽象機械を導出することを目的としたのに対し、本研究では環境をスタックに積むように変形する等、現実の機械により近いものの導出を目的としている。そしてその際、shift/reset の入った体系に対するスタック導入の正当性を厳密に示している。また Ager ら [1] は、抽象機械を分割して仮想機械とコンパイラを得る手法を示した。この手法を応用して、五十嵐ら [10] は backquote と unquote をサポートするマルチステージ言語 $\lambda^{\bigcirc}$ に対する仮想機械とコンパイラを導出している。本論文ではコンパイラの導出には触れないが、最終的には Ager らの手法にしたがって shift/reset に対するコンパイラを導出することを目指している。我々の研究

グループでは shift/reset をスタックのコピーとして実装するコンパイラの作成を行っており [13]、本研究の成果はそれに対する正当性を与えるものと期待される。

2 shift / reset

shift オペレータは、現在から最新の reset までの継続 k を取り出すものである。継続を取り出すという点においては call/cc と似ているが、取り出される継続の範囲が reset で限定される点と、shift した後に k が呼び出されなければその継続は破棄されるという点において異なっている。本研究では、`shift (...)` と書けば、継続 k を括弧内の関数に引数として渡すものとする。たとえば `1 + shift(fun k -> 2 * (k 3))` という計算は、「値を受け取ったら 1 を足す」という継続が k に渡され、`2 * (1 + 3)` となり、`8` が返ってくる。

reset オペレータは、前述の shift オペレータで取り出される継続の範囲を限定するものである。本研究では、`reset (...)` と書けば、括弧内の shift オペレータで取り出される継続を括弧内の処理のみに限定するものとする。たとえば `1 + reset(4 + shift(fun k -> 2 * (k 3)))` という計算は、「値を受け取ったら 4 を足す」という継続が k に渡され、`1 + (2 * (4 + 3))` となり、`15` が返ってくる。最初の 1 を足す部分は、reset の括弧内の shift 命令では取って来られない。

3 CPS のλ計算のインタプリタ

以下のような syntax を持つ言語を対象とする。

$$t \quad ::= \quad x \mid \lambda x.\, t \mid t_0\ t_1 \mid \texttt{shift}(t) \mid \texttt{reset}(t)$$

変数、関数定義、関数呼び出しの他に、shift と reset を記述できる。また、値としては関数定義の際に生成されるクロージャ $[x, t, e]$ と、shift 命令の際に取り出される継続 $[c]$ の二つがある。

$$v \quad ::= \quad [x, t, e] \mid [c]$$

これらを関数型言語 OCaml を用いて実装していく。まず、項と値の定義は以下のようになる。

```
(* 項 *)
type t = Var of string           (* 変数 *)
       | Fun of string * t       (* 関数定義 *)
       | App of t * t            (* 関数呼び出し *)
       | Shift of t              (* shift *)
       | Reset of t              (* reset *)
(* 値 *)
type v = VFun of string * t * e (* クロージャ *)
       | VCont of c              (* 継続 *)
(* 環境 *)
and e = (string * v) list
(* 継続 *)
and c = (v -> v)
```

環境は 変数名 x と値 v の組のリストとして実装した。関数 $get(x, e)$ によって、環境 e における変数 x の値 v を得られるものとする。shift/reset の意味を与える評価器 `eval1` は、以下のように定義される [6]。

```
(* id : v -> v *)
let id x = x

(* eval : t * e * c -> v *)
let rec eval (t, e, c) = match t with
    Var(x) -> c (get(x, e))
  | Fun(x, t) -> c (VFun(x, t, e))
  | App(t0, t1) -> eval (t1, e, (fun v1
                      -> eval (t0, e, (fun v0
                           -> (match v0 with
                                   (VFun(x', t', e')) -> eval(t', (x', v1) :: e', c)
                                 | (VCont(c')) -> c (c' v1) )))))
  | Shift(t) -> eval (t, e, (fun v
                  -> (match v with
                          (VFun(x', t', e')) -> eval (t', (x', VCont(c)) :: e', id)
                        | (VCont(c')) -> c' (VCont(c))) ))
  | Reset(t) -> c (eval (t, e, id))

(* eval1 : t -> v *)
let eval1 t = eval (t, [], id)
```

この評価器は一般的な CPS の λ 計算インタプリタに shift/reset を加えただけのものである。13 〜 16 行目が shift、17 行目が reset の処理に相当する。shift の処理では、その時点での継続を値 `VCont(c)` としてパッケージングしてから `Shift` 項の引数 `t` （の実行結果の関数）に渡している。reset の処理では継続を `id` にリセットしており、これにより shift 命令で取り出される継続が限定される。これが shift/reset の定義に基づく実装である。以下、この評価器に対して妥当性の保証された変換を施していくことで、shift/reset の低レベルな実装を得る。

4 CPS 変換

評価器が全て末尾呼び出しになっていれば、引数を状態と見なすと評価器からそのまま抽象機械及び状態遷移規則が得られるが、`eval1` は shift/reset 実装の際、末尾呼び出しでなくなっている箇所（12, 17 行目）がある。そこで、`eval1` に CPS 変換を施して全ての項で末尾呼び出しとし、その結果得られる評価器を `eval2` とする。この変換により、型は以下のように変わる。

```
type v = (* 前と同じ *)    (* 値 *)
and e = (* 前と同じ *)     (* 環境 *)
and c = ((v * d) -> v)     (* 継続 *)
and d = (v -> v)
```

評価器 `eval2` は以下のようになる。

```
(* cid : v * d -> v *)
(* did : v -> v *)
let cid (v, d) = d v
let did x = x

(* eval : t * e * c * d -> v *)
let rec eval (t, e, c, d) = match t with
    Var(x) -> c (get(x, e), d)
  | Fun(x, t) -> c (VFun(x, t, e), d)
  | App(t0, t1)
    -> eval (t1, e, (fun (v1, d1)
        -> eval (t0, e, (fun (v0, d0)
            -> (match v0 with
                    (VFun(x', t', e')) -> eval(t', (x', v1) :: e', c, d0)
                  | (VCont(c')) -> c' (v1, (fun v' -> c (v', d0)))))), d1)), d)
  | Shift(t)
    -> eval (t, e, (fun (v, d')
        -> (match v with
                (VFun(x', t', e')) -> eval (t', (x', VCont(c)) :: e', cid, d')
              | (VCont(c')) -> c' (VCont(c), d'))), d)
  | Reset(t) -> eval (t, e, cid, (fun v -> c (v, d)))

(* eval2 : t -> v *)
let eval2 t = eval (t, [], cid, did)
```

`eval1` はもともと CPS のインタプリタで継続 c を持っていたが、さらに CPS 変換が施された結果、継続が c と d の二つに分離されている。

この変換の正当性は、CPS 変換の正当性から直接導かれる。

定理 4.1 (CPS 変換の正当性 [16]).
任意の項 t に対し `eval1` と `eval2` は、どちらも評価に失敗するか、または構造的に等しい値を返す。（項 t を `eval1` で評価した結果を CPS 変換したものは、項 t を `eval2` で評価した結果と等しい。）

5 非関数化

`eval2` は二度の CPS 変換の結果、高階関数を多く使うようになっている。しかし、これから得ようとしている低レベルな実装では高階関数のような高級な機能はない。よって、高階関数を使わずに同等の処理を行うプログラムに変換する必要がある。そのため、`eval2` に対し非関数化 [16] という処理を施し、評価器 `eval3` とする。非関数化とは、プログラム中に出現する全ての高階関数 (OCaml の fun 文) に対し、その実行に必要なデータを保持できるオブジェクトを定義し、高階関数を生成する代わりにそのオブジェクトを生成するよう変更することである。さらに各オブジェクトに対し、その処理系 (apply 関数。本論文では run_* という名前を使う。) を用意する。

例えば eval2 では Shift 項が評価器に渡されると OCaml の fun 文を生成する（16〜20 行目）。この fun 文の body において、c は自由変数である。c は後でこの継続を処理する際に必要なデータで、eval2 ではすぐ外のスコープの変数を参照することで実現されているが、非関数化する場合は別の場所でも実行できるようにオブジェクトの生成時にこの c を格納しておく必要がある。よって Shift 項の fun 文に相当するオブジェクトは項 c をその子要素として保持するよう定義する。他の fun 文についても同様に、body で自由変数になっているものはオブジェクトに保持するようにする。よって型の定義は以下のように変更される。

```
1  type v = (* 前と同じ *)      (* 値 *)
2  and e = (* 前と同じ *)       (* 環境 *)
3  (* 継続 *)
4  and c = CApp1 of t * e * c  (* App項で生成される外側の fun 文のかわり *)
5          | CApp0 of v * c    (* App項で生成される内側の fun 文のかわり *)
6          | CShift of c       (* Shift項で生成される fun 文のかわり *)
7          | CReset            (* c の id *)
8  and d = DRun of c * d       (* Reset項で d として生成される fun のかわり *)
9          | DReset            (* d の id *)
```

続いて、各オブジェクトに対する処理系を用意する。eval2 では fun 文の中で行っていた処理を、この関数で行っている。

```
1  (* run_c : c * v * d -> v *)
2  let rec run_c (c, v, d) = match c with
3      CApp1(t', e', c') -> eval(t', e', CApp0(v, c'), d)
4    | CApp0(v', c') -> (match v with
5                            VFun(x', t', e') -> eval(t', (x', v') :: e', c', d)
6                          | VCont(c'') -> run_c (c'', v', DRun(c', d)))
7    | CShift(c') ->  (match v with
8                          VFun(x', t', e') -> eval(t', (x', VCont(c')) :: e', CReset, d)
9                        | VCont(c'') -> run_c (c'', VCont(c'), d))
10   | CReset -> run_d (d, v)
11 (* run_d : d * v -> v *)
12 and run_d (d, v) = match d with
13     DRun(c', d') -> run_c (c', v, d')
14   | DReset -> v
```

fun 文の代わりに相当するオブジェクトを返すように書き換えるだけでなく、継続の処理に進む箇所は上で用意した処理系に渡すように変更する。例えば eval2 の 8 行目では Var 項が評価器に渡されると、環境から求められた値を継続 c にそのまま渡している。非関数化すると、引数 c には継続オブジェクトが入っているので、それに相当する処理をするためには上記の処理系を呼び出さなければならない。よって、eval3 では Var 項は run_c (c, get(x, e), d) と変換される。他の項や d についても同様に変更し、次のような評価器 eval3 を得る。

```
1  (* cid : c *)
2  (* did : d *)
3  let cid = CReset
4  let did = DReset
5
6  (* eval : t * e * c * d -> v *)
7  let rec eval (t, e, c, d) = match t with
8      Var(x) -> run_c (c, get(x, e), d)
9    | Fun(x, t) -> run_c (c, VFun(x, t, e), d)
10   | App(t0, t1) -> eval (t1, e, CApp1(t0, e, c), d)
11   | Shift(t) -> eval (t, e, CShift(c), d)
12   | Reset(t) -> eval (t, e, cid, DRun(c, d))
13
14 (* eval3 : t -> v *)
15 let eval3 t = eval (t, [], cid, did)
```

この変換により、OCaml の fun 文を使って受け渡されていた継続は、継続の実行に必要なデータを保持したオブジェクトとその処理系に分離された。ここで得られた継続オブジェクトは処理の段階を表すものであり、コード列のようなものと見なせる。

この変換の正当性は、非関数化の正当性から直接導かれる。

定理 5.1 (非関数化 [16] の正当性).

任意の項 t に対し eval2 と eval3 は、どちらも評価に失敗するか、または構造的に等しい値を返す。(項 t を eval2 で評価した結果を非関数化したものは、項 t を eval3 で評価した結果と等しい。)

6 線形リスト化

前節で、eval3 で受け渡される継続オブジェクトはコード列のようなものと見なせる、と書いたが eval3 の継続オブジェクトは実際には木構造を成しており、コード『列』であるとは言えない。ただし各継続オブジェクトは高々一つしか c 型の子要素を持たないため、データ構造としては線形リストと見なすことができ、実装もリストに書き換えることが可能である。よって木構造でなくリストにして受け渡しするように変更し、その評価器を eval4 とする。この変換の結果、引数 c で受け渡すものがコード列であると言えるようにする。

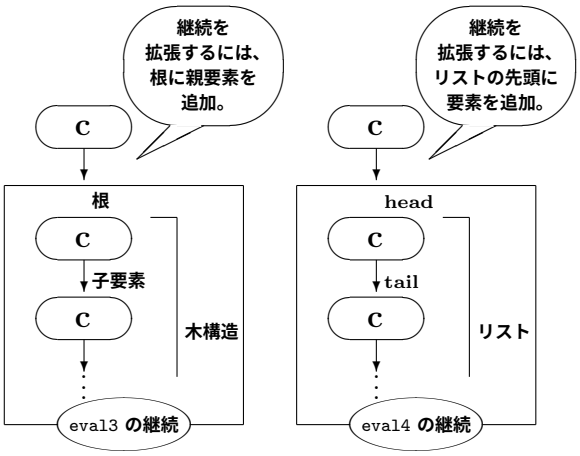

図 1: eval3 と eval4 における継続

eval3 では、c 型の継続オブジェクト のうち CApp1, CApp0, CShift がさらに c 型データを保持しており、これにより継続オブジェクトが木構造を成すようになっていた。よって、これらが c 型のデータを持たないように変更する。その代わりに評価器は c 型一つではなくリストを受け渡すようにし、それまでオブジェクト中に c 型データを保持していた箇所ではリストの先頭に新しい継続オブジェクトを付け加えることにする。すると、保持すべき c 型データをリストの tail として受け渡せる。c 型のデータを持たない CReset では、空リストを渡すように変える。d についても同様の変換を施す。この結果、c と d に関する型の定義は以下のように変わる。

```
(* 値 *)
type v = VFun of string * t * e
       | VCont of c list       (* c が c list に *)
and e = (* 前と同じ *)         (* 環境 *)
(* 継続 *)
and c = CApp1 of t * e         (* c 型のものは c 型の要素を取らないよう変更 *)
      | CApp0 of v             (* もともと c 型のデータを持たない CReset は *)
      | CShift                 (* 空リストを渡すよう変わる *)
and d = DRun of c list         (* d 型のものは d 型の要素を取らないよう変更 *)
```

評価器 eval4 は以下のようになる。

```
(* identity function *)
(* cid : c list *)
(* did : d list *)
let cid = []
let did = []

(* eval : t * e * c list * d list -> v *)
let rec eval (t, e, c, d) = match t with
    Var(x) -> run_c (c, get(x, e), d)
  | Fun(x, t) -> run_c (c, VFun(x, t, e), d)
  | App(t0, t1) -> eval (t1, e, CApp1(t0, e) :: c, d)
  | Shift(t) -> eval (t, e, CShift :: c, d)
  | Reset(t) -> eval (t, e, cid, DRun(c) :: d)

(* run_c : c list * v * d list -> v *)
```

```
16 and run_c (c, v, d) = match c with
17     CApp1(t', e') :: c' -> eval(t', e', CApp0(v) :: c', d)
18   | CApp0(v') :: c' -> (match v with
19                             VFun(x', t', e') -> eval(t', (x', v') :: e', c', d)
20                           | VCont(c'') -> run_c (c'', v', DRun(c') :: d))
21   | CShift :: c' ->  (match v with
22                           VFun(x', t', e') -> eval(t', (x', VCont(c')) :: e', cid, d)
23                         | VCont(c'') -> run_c (c'', VCont(c'), d))
24   | [] -> run_d (d, v)
25
26 (* run_d : d list * v -> v *)
27 and run_d (d, v) = match d with
28     DRun(c') :: d' -> run_c (c', v, d')
29   | [] -> v
30
31 (* eval4 : t -> v *)
32 let eval4 t = eval (t, [], cid, did)
```

前述の通り、評価器の引数が c から c list、d から d list になっている。また、11, 12, 17 行目は eval3 では継続データをオブジェクト内に格納していた箇所だが、eval4 ではリストの先頭に新しい継続オブジェクトを付け加えたものを受け渡している。13 行目では CReset の代わりに空リスト (cid) を受け渡している。またこの行では、Reset 項が評価器に渡されると現在の継続のパッケージ化したものである DRun(c) が d list の先頭に積まれている。つまり reset 命令で継続が c から切り取られ d へ移されること、d はコード列のリストになることが確認できる。通常、shift/reset 処理は「reset 命令で継続に印を付け、shift 命令でその印までの継続を取り出す」という形で実装されるのに対し、ここでの実装は、「reset 命令で継続を退避させ、shift 命令では退避していない継続を全て取り出す」という形になっている。後で触れるように、両者は相互に変換可能である。

eval3 と eval4 の違いは継続オブジェクトとその拡張時の処理だけである。 eval3 における継続オブジェクトは木構造になっているが、どの継続オブジェクトも子要素として持てる継続オブジェクトは最大一つであるため、どれだけ木が大きくなっても eval4 の継続オブジェクトのリストと一対一の対応をとることができる。eval3 で継続オブジェクトを生成する際の拡張処理も、eval4 でリストの先頭に継続オブジェクトを付け加える処理に対応し、拡張によって継続木と継続リストの間の対応関係が崩れることはない。よって、この変換の正当性が導かれる。

定理 6.1 (線形リスト化の正当性).
任意の項 t に対し eval3 と eval4 は、どちらも評価に失敗するか、または構造的に等しい値を返す。(項 t を eval3 で評価した結果を線形リスト化したものは、項 t を eval4 で評価した結果と等しくなる。)

7 スタック導入

7.1 値をスタックに積むよう変換

これまで値は継続オブジェクト内に保持されていた。例えば、 eval4 の CApp0 は引数として v を取っている。しかし、一般的な機械語ではスタックに中間結果を保存する。より機械語の挙動に近づけるために新たに評価器に引数としてスタックを持たせ、値をスタックに積んで受け渡すように変更し、その評価器を eval5 とする。ここで、スタックは値のリストとして実装する。また、継続オブジェクトの中に保存していた値がスタックに格納されるように変わるため、これまで継続として取り出していたものが継続オブジェクトとスタックに分離することとなる。よって、継続を取り出す際は c list だけでなくスタックも取り出さなければならない。そのため、以下のように型の定義を変更する。

```
1  (* 値 *)
2  type v = VFun of string * t * e
3           | VCont of (c list) * s  (* c list とスタックを格納 *)
4  (* スタック *)
5  and s = v list                    (* スタックは値のリスト *)
6  and e = (* 前と同じ *)            (* 環境 *)
7  (* 継続 *)
8  and c = CApp1 of t * e
9          | CApp0                   (* 値を保持しなくなった *)
10         | CShift
11 and d = DRun of (c list) * s      (* c list とスタックと格納 *)
```

ここまでは CApp0(v) として v を受け取っていたが、この v は継続オブジェクトで保存するのではなくスタックに積まれるように変わったので、以降は引数のない CApp0 になっている。また、継続の表現が c list から (c list) * s となっており、VCont や DRun の引数が変わっている。

評価器は、新しくスタック s を引数として受け渡すように変更する。それに伴い、これまで値を受け渡していた run_c 関数、run_d 関数でもスタックに格納されたデータを保持するために、値一つでなくスタックを受け渡すように引数を変更する。 run_c 関数、run_d 関数を呼び出す部分では、これまで渡していた値をスタックの先頭に積んで渡す。これらの変更により、評価器 eval5 は以下のようになる。

```
1  (* cid : c list *)
2  (* did : d list *)
3  let cid = []
4  let did = []
5
6  (* eval : t * s * e * c * d -> v *)
7  let rec eval (t, s, e, c, d) = match t with
8      Var(x) -> run_c (c, get(x, e) :: s, d)
9    | Fun(x, t) -> run_c (c, VFun(x, t, e) :: s, d)
10   | App(t0, t1) -> eval (t1, s, e, CApp1(t0, e) :: c, d)
11   | Shift(t) -> eval (t, s, e, CShift :: c, d)
12   | Reset(t) -> eval (t, [], e, cid, DRun(c, s) :: d)
13
14 (* run_c : c * s * d -> v *)
15 and run_c (c, s, d) = match c with
16     CApp1(t', e') :: c' -> eval(t', s, e', CApp0 :: c', d)
17   | CApp0 :: c' ->
18       (match s with
19            VFun(x', t', e') :: v' :: s' -> eval(t', s', (x', v') :: e', c', d)
20          | VCont(c'', s'') :: v' :: s' -> run_c (c'', v' :: s'', DRun(c', s') :: d))
21   | CShift :: c' ->
22       (match s with
23            VFun(x', t', e') :: s' -> eval(t', [], (x', VCont(c', s')) :: e', cid, d)
24          | VCont(c'', s'') :: s' -> run_c (c'', VCont(c', s') :: s'', d))
25   | [] -> run_d (d, s)
26
27 (* run_d : d * s -> v *)
28 and run_d (d, s) = match (d, s) with
29     (DRun(c', s') :: d', v :: s'') -> run_c (c', v :: s', d')
30   | ([], v :: s'') -> v
31
32 (* eval5 : t -> v *)
33 let eval5 t = eval (t, [], [], cid, did)
```

前述の通り、評価器及び処理系が引数としてスタック s を受け渡すようになる。8, 9 行目は eval4 では値一つを渡していた箇所だが、値をスタックに積んで渡すように変わっている。また、20, 24, 29 行目は継続を取り出して実行する箇所だが、VCont や DRun からコード列だけでなくその実行に必要なスタックも復元している。

また 29, 30 行目の、引数として受け取ったスタックの先頭を除いたもの (s'') は、評価器が正しく作られていれば空になる。

7.2 スタック導入前後の等価性

前節ではスタック導入の直観的な説明と、スタック導入した評価器の定義を与えた。しかし、このスタック導入が妥当な変換であるかどうかは明らかではない。実際、eval4 と eval5 は値や継続の型の定義が異なっており、両者の間の単純な等価関係は構築できそうにない。そのため、本節で評価器 eval4 と eval5 が等価であることを bisimulation の証明によって行う。

まずは bisimulation の概念を導入する。

定義 7.1 (bisimulation [14]).

$P \sim Q$ となる P, Q について次の二条件が成立するとき、 $\sim$ は bisimulation である。

- $P \xrightarrow{\alpha} P'$ となる P' が存在すれば、$Q \xrightarrow{\alpha'} Q'$ かつ $P' \sim Q'$ となる Q' が存在する。
- $Q \xrightarrow{\alpha'} Q'$ となる Q' が存在すれば、$P \xrightarrow{\alpha} P'$ かつ $P' \sim Q'$ となる P' が存在する。

bisimulation は、状態遷移系の同値関係である。

eval4 と eval5 は全て末尾呼び出しになっており、引数を状態と考えることができる。例えば eval4 の 9 行目では、eval 関数に Var(x) が渡されると run_c (c, get(x, e), d) が返ってくるが、これは状態 $\langle \mathtt{Var}(x), e_4, c_4, d_4 \rangle$ が状態 $\langle c_4, get(x, e_4), d_4 \rangle$ に遷移していると見なせる。

このように eval, run_c, run_d の引数、そして run_d の返す最終結果をそれぞれ $\langle t, e, c, d \rangle, \langle c, v, d \rangle, \langle d, v \rangle, \langle v \rangle$ と表記することにすると、eval4 は図 2 のような遷移規則を持つ状態遷移機械と見なすことができる。

t	$\Rightarrow$	$\langle t, [], [], [] \rangle$
$\langle \mathtt{Var}(x), e, c, d \rangle$	$\Rightarrow$	$\langle c, get(x, e), d \rangle$
$\langle \mathtt{Fun}(x, t), e, c, d \rangle$	$\Rightarrow$	$\langle c, \mathtt{VFun}(x, t, e), d \rangle$
$\langle \mathtt{App}(t0, t1), e, c, d \rangle$	$\Rightarrow$	$\langle t1, e, \mathtt{CApp1}(t0, e) :: c, d \rangle$
$\langle \mathtt{Shift}(t), e, c, d \rangle$	$\Rightarrow$	$\langle t, e, \mathtt{CShift} :: c, d \rangle$
$\langle \mathtt{Reset}(t), e, c, d \rangle$	$\Rightarrow$	$\langle t, e, [], \mathtt{DRun}(c) :: d \rangle$
$\langle \mathtt{CApp1}(t, e) :: c, v, d \rangle$	$\Rightarrow$	$\langle t, e, \mathtt{CApp0}(v) :: c, d \rangle$
$\langle \mathtt{CApp0}(v) :: c, \mathtt{VFun}(x, t, e), d \rangle$	$\Rightarrow$	$\langle t, (x, v) :: e, c, d \rangle$
$\langle \mathtt{CApp0}(v) :: c, \mathtt{VCont}(c'), d \rangle$	$\Rightarrow$	$\langle c', v, \mathtt{DRun}(c) :: d \rangle$
$\langle \mathtt{CShift} :: c, \mathtt{VFun}(x, t, e), d \rangle$	$\Rightarrow$	$\langle t, (x, \mathtt{VCont}(c)) :: e, [], d \rangle$
$\langle \mathtt{CShift} :: c, \mathtt{VCont}(c'), d \rangle$	$\Rightarrow$	$\langle c', \mathtt{VCont}(c), d \rangle$
$\langle [], v, d \rangle$	$\Rightarrow$	$\langle d, v \rangle$
$\langle \mathtt{DRun}(c') :: d, v \rangle$	$\Rightarrow$	$\langle c', v, d \rangle$
$\langle [], v \rangle$	$\Rightarrow$	$\langle v \rangle$

図 2: eval4 の状態遷移規則

eval5 についても同様に、$\langle t, s, e, c, d \rangle, \langle c, s, d \rangle, \langle d, s \rangle, \langle v \rangle$ の間の、図 3 のような遷移規則を持つ状態遷移機械とみなすことができる。

t	$\Rightarrow$	$\langle t, [], [], [], [] \rangle$
$\langle \mathtt{Var}(x), s, e, c, d \rangle$	$\Rightarrow$	$\langle c, get(x, e) :: s, d \rangle$
$\langle \mathtt{Fun}(x, t), s, e, c, d \rangle$	$\Rightarrow$	$\langle c, \mathtt{VFun}(x, t, e) :: s, d \rangle$
$\langle \mathtt{App}(t0, t1), s, e, c, d \rangle$	$\Rightarrow$	$\langle t1, s, e, \mathtt{CApp1}(t0, e) :: c, d \rangle$
$\langle \mathtt{Shift}(t), s, e, c, d \rangle$	$\Rightarrow$	$\langle t, s, e, \mathtt{CShift} :: c, d \rangle$
$\langle \mathtt{Reset}(t), s, e, c, d \rangle$	$\Rightarrow$	$\langle t, [], e, [], \mathtt{DRun}(c, s) :: d \rangle$
$\langle \mathtt{CApp1}(t, e) :: c, s, d \rangle$	$\Rightarrow$	$\langle t, s, e, \mathtt{CApp0} :: c, d \rangle$
$\langle \mathtt{CApp0} :: c, \mathtt{VFun}(x, t, e) :: v :: s, d \rangle$	$\Rightarrow$	$\langle t, s, (x, v) :: e, c, d \rangle$
$\langle \mathtt{CApp0} :: c, \mathtt{VCont}(c', s') :: v :: s, d \rangle$	$\Rightarrow$	$\langle c', v :: s', \mathtt{DRun}(c, s) :: d \rangle$
$\langle \mathtt{CShift} :: c, \mathtt{VFun}(x, t, e) :: s, d \rangle$	$\Rightarrow$	$\langle t, [], (x, \mathtt{VCont}(c, s)) :: e, [], d \rangle$
$\langle \mathtt{CShift} :: c, \mathtt{VCont}(c', s') :: s, d \rangle$	$\Rightarrow$	$\langle c', \mathtt{VCont}(c, s) :: s', d \rangle$
$\langle [], s, d \rangle$	$\Rightarrow$	$\langle d, s \rangle$
$\langle \mathtt{DRun}(c', s') :: d, v :: s \rangle$	$\Rightarrow$	$\langle c', v :: s', d \rangle$
$\langle [], v :: s \rangle$	$\Rightarrow$	$\langle v \rangle$

図 3: eval5 の状態遷移規則

このように導き出された eval4 と eval5 の状態遷移系間の関係が bisimulation であることを証明することにより eval4 と eval5 の同値関係を示す。

そのために、eval4 と eval5 の間にどういった関係があるかをまず定義したい。ここで、eval4 での値有り継続オブジェクト c_4 は、eval5 ではスタック s と値無し継続オブジェクト c_5 に分離されていることに着目する。次のような三項関係 $\_ =_s \_ \otimes \_$ を定義し、$c_4 =_s s \otimes c_5$ と書いて、 c_4 はスタック導入により s と c_5 に分離されることを表すことにする。記号 $=_s$ は eval4 と eval5 での状態の対応関係を意図している。

定義 7.2 (三項関係 $\_ =_s \_ \otimes \_$).
eval4 における状態の集合を S_4、eval5 における状態の集合を S_5 とし、
$\langle t, e_4, c_4, d_4\rangle \in S_4$, $\langle c_4, v_4, d_4\rangle \in S_4$, $\langle t, s, e_5, c_5, d_5\rangle \in S_5$, $\langle c_5, v_5 :: s, d_5\rangle \in S_5$ とする。
このとき、$\_ =_s \_ \otimes \_$ を以下のように定義する。

(i) $c_4 = \texttt{[]}$, $s = \texttt{[]}$, $c_5 = \texttt{[]}$ のとき、 $c_4 =_s s \otimes c_5$

(ii) $c_4 =_s s \otimes c_5$, $e_4 =_s e_5$ が成り立ち、$c_4' = \texttt{CApp1}_4(t, e_4) :: c_4$, $c_5' = \texttt{CApp1}_5(t, e_5) :: c_5$ であるとき、$c_4' =_s s \otimes c_5'$

(iii) $c_4 =_s s \otimes c_5$, $v_4 =_s v_5$ が成り立ち、$c_4' = \texttt{CApp0}_4(v_4) :: c_4$, $c_5' = \texttt{CApp0}_5 :: c_5$ であるとき、$c_4' =_s (v_5 :: s) \otimes c_5'$

(iv) $c_4 =_s s \otimes c_5$ が成り立ち、$c_4' = \texttt{CShift}_4 :: c_4$, $c_5' = \texttt{CShift}_5 :: c_5$ であるとき、 $c_4' =_s s \otimes c_5'$

ただし、$c_4 =_s s \otimes c_5$ のとき $\texttt{VCont}_4(c_4) =_s \texttt{VCont}_5(s, c_5)$, $\texttt{DRun}_4(c_4) =_s \texttt{DRun}_5(c_5, s)$ と定義する。他の項についても、その子要素にすべて対応関係が言えれば、関係 $=_s$ が成り立つものと定義する。

定義 7.2 の (iii) において、c_4 には $\texttt{CApp0}_4(v_4)$ として v_4 を付け加えているのに対し、c_5 には $\texttt{CApp0}_5$ だけを付け加えていることに注意する。そのかわり v_5 が s に付け加えられており、これがスタックに値を積む動作を表している。
以下、この三項関係 $\_ =_s \_ \otimes \_$ を用いて eval4 と eval5 の状態遷移系の間にある関係 $\sim_s$ を定義し、これが bisimulation であることを示す。

定理 7.1 (スタック導入前後の bisimulation).
$c_4 =_s s \otimes c_5$, $e_4 =_s e_5$, $d_4 =_s d_5$, $v_4 =_s v_5$ を満たすとき $\langle t, e_4, c_4, d_4\rangle \sim_s \langle t, s, e_5, c_5, d_5\rangle$
$c_4 =_s s \otimes c_5$, $d_4 =_s d_5$, $v_4 =_s v_5$ を満たすとき $\langle c_4, v_4, d_4\rangle \sim_s \langle c_5, v_5 :: s, d_5\rangle$
$d_4 =_s d_5$, $v_4 =_s v_5$ を満たすとき $\langle d_4, v_4\rangle \sim_s \langle d_5, v_5 :: s'\rangle$
$v_4 =_s v_5$ を満たすとき $\langle v_4\rangle \sim_s \langle v_5\rangle$
と書く事にする。
このとき、$\sim_s$ は bisimulation である。

証明. $P \in S_4$, $Q \in S_5$ について、$P \sim_s Q$ が成り立つとする。
このとき各状態遷移に対して、$P \xrightarrow{\text{eval4}} P'$ ならば $Q \xrightarrow{\text{eval5}} Q'$ かつ $P' \sim_s Q'$ となる Q が存在することと、$Q \xrightarrow{\text{eval5}} Q'$ ならば $P \xrightarrow{\text{eval4}} P'$ かつ $P' \sim_s Q'$ となる P が存在することとを示す。そのために、各状態について場合分けをし、全ての場合でこれが言えることを確かめていく。
各場合の詳細な検証は参考文献 [11] において与えている。 □

eval4 と eval5 の状態遷移系の間の関係が bisimulation であることが言えたので、スタック導入は妥当な変換である。

定理 7.2 (スタック導入の正当性).
任意の項 t に対し eval4 と eval5 は、どちらも評価に失敗するか、または構造的に等しい値を返す。(項 t を eval4 で評価した結果を v_4、eval5 で評価した結果を v_5 とすると、$v_4 =_s v_5$ が成り立つ。)

スタック導入の正当性が厳密な形で示されたことは本研究の重要な成果の一つである。Biernacka ら [4] はスタック導入について詳述していない。Danvy ら [7] はスタック除去前後の評価器が等価であることの直感的な説明は行っているが、本研究で行ったような厳密な証明は行っていない。λ式のみであればそれだけで正しさを確信できるかもしれないが、shift/reset が入ると、継続の取り出し方など、直感的説明だけでは正当性の確信が難しい部分が出てくる。実際、我々が eval5 を実装する際、証明するまで見つからなかった小さな誤りが発見された。また今後、機械語による実装を視野に入れるようになると、ここで示したような厳密な証明は必須であると考えられる。

8 環境退避変換

8.1 環境をスタックに積む

eval5 は関数呼び出しされると継続オブジェクト c にその時点での環境 e を格納している (10 行目) 。この評価器の c list はコード列に相当するが、実際の機械語においては普通『環境を保存する』『環境を復帰させる』に相当するコードはあっても、コードとして環境そのものを保存することはない。よって、継続オブジェクト内に保持されていた環境もスタックに積んで受け渡すように変更し、その評価器を eval6 とする。

この変換により、これまで環境を保持していた CApp1 は環境を保持しなくなる。その代わり、スタックに環境を積むために、パッケージ化して値として扱えるよう型の定義を以下のように変更する。

```
1  (* 値 *)
2  type v = VFun of string * t * e
3           | VCont of (c list) * s
4           | VEnv of e                    (* 環境を格納する *)
5  and s = (* 前と同じ *)                  (* スタック *)
6  and e = (* 前と同じ *)                  (* 環境 *)
7  (* 継続 *)
8  and c = CApp1 of t                     (* 環境をもたない *)
9          | CApp0
10         | CShift
11 and d = (* 前と同じ *)
```

評価器は関数呼び出しの際の処理が変わる。これまでは CApp1 に格納していた環境 e を、VEnv(e) として値にパッケージ化してスタックに積むようにする。この継続は CApp1 の処理時にスタックから復帰される。よって、eval6 の定義は以下のようになる。

```
1  (* cid : c list *)
2  (* did : d list *)
3  let cid = []
4  let did = []
5
6  (* eval : t * s * e * c list * d list -> v *)
7  let rec eval (t, s, e, c, d) = match t with
8      Var(x) -> run_c (c, get(x, e) :: s, d)
9    | Fun(x, t) -> run_c (c, VFun(x, t, e) :: s, d)
10   | App(t0, t1) -> eval (t1, VEnv(e) :: s, e, CApp1(t0) :: c, d)
11   | Shift(t) -> eval (t, s, e, CShift :: c, d)
12   | Reset(t) -> eval (t, [], e, CReset :: [], DRun(c, s) :: d)
13
14 (* run_c : c list * s * d -> v *)
15 and run_c (c, s, d) = match c with
16     CApp1(t') :: c' ->
17       (match s with
18            v :: VEnv(e') :: s -> eval(t', v :: s, e', CApp0 :: c', d))
19   | CApp0 :: c' ->
20       (match s with
21            VFun(x', t', e') :: v' :: s' -> eval(t', s', (x', v') :: e', c', d)
22          | VCont(c'', s'') :: v' :: s' -> run_c (c'', v' :: s'', DRun(c', s') :: d))
23   | CShift :: c' ->
24       (match s with
25            VFun(x', t', e') :: s'
26              -> eval(t', [], (x', VCont(c', s')) :: e', CReset :: [], d)
27          | VCont(c'', s'') :: s' -> run_c (c'', VCont(c', s') :: s'', d))
28   | CReset :: _ -> run_d (d, s)
29
30 (* run_d : d  list * s -> v *)
31 and run_d (d, s) = match (d, s) with
32     (DRun(c', s') :: d', v :: s'') -> run_c (c', v :: s', d')
33   | (DReset :: _, v :: s'') -> v
34
35 (* eval6 : t -> v *)
36 let eval6 t = eval (t, [], [], cid, did)
```

前述の通り、App 項の処理（10 行目）で環境が VEnv にパッケージ化されスタックに積まれた上で、実行は $t1$

へ進む。$t1$ の実行が終了し、$\texttt{CApp1}(t0)$ に結果が渡されると、その時点でのスタックには先頭に $t1$ の実行結果、その下に先程保存した環境が積まれている筈である（16〜18 行目）。その後の実行は、保存されていた環境を復活してから行われる。

この変換の結果、継続オブジェクトは項 (t) 以外のデータを持たなくなる。これは、項さえ与えられればコード列を生成できることを意味する。また、これにより後に使う変数の退避・復活をモデル化できていると考えられる。

8.2 環境退避変換前後の等価性

eval5 について、eval，run_c，run_d の引数及び run_d の返す最終結果をそれぞれ $\langle t, s, e, c, d\rangle, \langle c, s, d\rangle, \langle d, s\rangle, \langle v\rangle$ と表現し、eval5 はこの状態の間を遷移する状態遷移機械と見なすことができるということは 7.2 節で述べた。eval6 も全て末尾呼び出しになっているため、同様に $\langle t, s, e, c, d\rangle, \langle c, s, d\rangle, \langle d, s\rangle, \langle v\rangle$ の間を遷移する、図 4 のような遷移規則を持つ状態遷移機械と見なすことができる。

t	$\Rightarrow$	$\langle t, [], [], [], []\rangle$
$\langle \texttt{Var}(x), s, e, c, d\rangle$	$\Rightarrow$	$\langle c, get(x, e) :: s, d\rangle$
$\langle \texttt{Fun}(x, t), s, e, c, d\rangle$	$\Rightarrow$	$\langle c, \texttt{VFun}(x, t, e) :: s, d\rangle$
$\langle \texttt{App}(t0, t1), s, e, c, d\rangle$	$\Rightarrow$	$\langle t1, \texttt{VEnv}(e) :: s, e, \texttt{CApp1}(t0) :: c, d\rangle$
$\langle \texttt{Shift}(t), s, e, c, d\rangle$	$\Rightarrow$	$\langle t, s, e, \texttt{CShift} :: c, d\rangle$
$\langle \texttt{Reset}(t), s, e, c, d\rangle$	$\Rightarrow$	$\langle t, [], e, [], \texttt{DRun}(c, s) :: d\rangle$
$\langle \texttt{CApp1}(t) :: c, v :: \texttt{VEnv}(e) :: s, d\rangle$	$\Rightarrow$	$\langle t, v :: s, e, \texttt{CApp0} :: c, d\rangle$
$\langle \texttt{CApp0} :: c, \texttt{VFun}(x, t, e) :: v :: s, d\rangle$	$\Rightarrow$	$\langle t, s, (x, v) :: e, c, d\rangle$
$\langle \texttt{CApp0} :: c, \texttt{VCont}(c', s') :: v :: s, d\rangle$	$\Rightarrow$	$\langle c', v :: s', \texttt{DRun}(c, s) :: d\rangle$
$\langle \texttt{CShift} :: c, \texttt{VFun}(x, t, e) :: s, d\rangle$	$\Rightarrow$	$\langle t, [], (x, \texttt{VCont}(c, s)) :: e, [], d\rangle$
$\langle \texttt{CShift} :: c, \texttt{VCont}(c', s') :: s, d\rangle$	$\Rightarrow$	$\langle c', \texttt{VCont}(c, s) :: s', d\rangle$
$\langle [], s, d\rangle$	$\Rightarrow$	$\langle d, s\rangle$
$\langle \texttt{DRun}(c', s') :: d, v :: s\rangle$	$\Rightarrow$	$\langle c', v :: s', d\rangle$
$\langle [], v :: s\rangle$	$\Rightarrow$	$\langle v\rangle$

図 4: eval6 の状態遷移規則

この状態遷移系間の関係が bisimulation であることを証明することにより eval5 と eval6 の間の同値関係を示す。

そのために、環境退避変換によって状態遷移系がどう変わったかを明確に記述する必要がある。ここで、eval5 では c に保持されていた環境が eval6 ではスタックに積まれることと、環境保持、復元処理以外は変化がないことに着目し、eval5 のスタック s_5, コード列 c_5, eval6 のスタック s_6, コード列 c_6 の間の四項関係を定義する。環境退避変換によって c_5 に保持されていた環境がスタックに積まれるよう変わり、スタックが s_6, コード列が c_6 になることを、 $s_5 \otimes c_5 =_e s_6 \odot c_6$ と書き表すとする。記号 $=_e$ は eval5 と eval6 の間の対応関係を意図している。

定義 8.1 (四項関係 $\_ \otimes \_ =_e \_ \odot \_$).

eval5 における状態の集合を S_5、eval6 における状態の集合を S_6 とし、
$\langle t, s_5, e_5, c_5, d_5\rangle \in S_5$, $\langle c_5, v_5 :: s_5, d_5\rangle \in S_5$, $\langle t, s_6, e_6, c_6, d_6\rangle \in S_6$, $\langle c_6, v_6 :: s_6, d_6\rangle \in S_6$ とする。

このとき、$\_ \otimes \_ =_e \_ \odot \_$ を以下のように定義する。

(i) $s_5 = [], c_5 = []$, $s_6 = []$, $c_6 = []$ のとき、 $s_5 \otimes c_5 =_e s_6 \odot c_6$

(ii) $s_5 \otimes c_5 =_e s_6 \odot c_6$ が成り立ち、かつ $e_5 =_e e_6$ となるとき、$c'_5 = \texttt{CApp1}_5(t, e_5) :: c_5$, $c'_6 = \texttt{CApp1}_6(t) :: c_6$ であるとき、 $s_5 \otimes c'_5 =_e (\texttt{VEnv}(e_6) :: s_6) \odot c'_6$

(iii) $s_5 \otimes c_5 =_e s_6 \odot c_6$ が成り立ち、$c'_5 = \texttt{CApp0}_5 :: c_5$, $c'_6 = \texttt{CApp0}_6 :: c_6$ であるとき、 $(v_5 :: s_5) \otimes c'_5 =_e (v_6 :: s_6) \odot c'_6$

(iv) $s_5 \otimes c_5 =_e s_6 \odot c_6$ が成り立ち、$c'_5 = \mathtt{CShift}_5 :: c_5$, $c'_6 = \mathtt{CShift}_6 :: c_6$ であるとき、 $s_5 \otimes c'_5 =_e s_6 \odot c'_6$

さらに、$s_5 \otimes c_5 =_e s_6 \odot c_6$ のとき $\mathtt{VCont}_5(c_5, s_5) =_e \mathtt{VCont}_6(c_6, s_6)$, $\mathtt{DRun}_5(c_5, s_5) =_e \mathtt{DRun}_6(c_6, s_6)$ と定義する。他の項についても、その子要素が全て等しければ、関係 $=_e$ が成り立つものと定義する。

定義 8.1 の (ii) において、c_5 には $\mathtt{CApp1}_5(t, e_5)$ を付け加えているのに対し、c_6 には $\mathtt{CApp1}_6(t)$ だけを付け加えていることに注意する。そのかわり $\mathtt{VEnv}(e_6)$ が s_6 に付け加えられており、これがスタックに環境を積む動作を表している。

四項関係 $\_ \otimes \_ =_e \_ \odot \_$ を用いて eval5 と eval6 の状態遷移系間の関係を定義し、それが bisimulation であることを示す。

定理 8.1 (環境退避前後の bisimulation).

$s_5 \otimes c_5 =_e s_6 \odot c_6$, $e_5 =_e e_6$, $d_5 =_e d_6$, $v_5 =_e v_6$ を満たすとき $\langle t, s_5, e_5, c_5, d_5\rangle \sim_e \langle t, s_6, e_6, c_6, d_6\rangle$

$s_5 \otimes c_5 =_e s_6 \odot c_6$, $d_5 =_e d_6$, $v_5 =_e v_6$ を満たすとき $\langle c_5, v_5 :: s_5, d_5\rangle \sim_e \langle c_6, v_6 :: s_6, d_6\rangle$

$d_5 =_e d_6$, $v_5 =_e v_6$ を満たすとき $\langle d_5, v_5 :: s_5\rangle \sim_e \langle d_6, v_6 :: s_6\rangle$

$v_5 =_e v_6$ を満たすとき $\langle v_5\rangle \sim_e \langle v_6\rangle$

と書く事にする。

このとき、$\sim_e$ は bisimulation である。

証明. $P \in S_5$, $Q \in S_6$ について、$P \sim_s Q$ が成り立つとする。

このとき各状態遷移に対して、$P \xrightarrow{\mathrm{eval5}} P'$ ならば $Q \xrightarrow{\mathrm{eval6}} Q'$ かつ $P' \sim_s Q'$ となる Q が存在することと、$Q \xrightarrow{\mathrm{eval6}} Q'$ ならば $P \xrightarrow{\mathrm{eval5}} P'$ かつ $P' \sim_s Q'$ となる P が存在することとを示す。定理 7.1 の証明と同様、各状態について場合分けをし、全ての場合でこれが言えることを確かめていく。

各場合の詳細な検証は参考文献 [11] において与えている。 □

eval5 と eval6 の状態遷移系の間の関係が bisimulation であることが言えたので、環境退避変換は妥当な変換である。

定理 8.2 (環境退避変換の正当性).

任意の項 t に対し eval5 と eval6 は、どちらも評価に失敗するか、または構造的に等しい値を返す。(項 t を eval5 で評価した結果を v_5、eval6 で評価した結果を v_6 とすると、$v_5 =_e v_6$ が成り立つ。)

環境退避変換の正当性が示されたことにより、機械語の実装に見られる局所変数の退避、復活が、言語の定義を与える CPS インタプリタ（3 節）と等価であることが厳密に示された。従来、Danvy らの系統的な抽象機械導出法は抽象機械や仮想機械のレベルまでととらえられていたが、ここでの結果は、それが実際の機械語レベルまで拡張可能であることを示している。shift/reset のような複雑な操作が入った体系では、機械語レベルの正当性を示すのが重要であるが、ここでの結果はそれに道を開くものとなっている。

9 抽象機械

評価器 eval6 は全て末尾呼び出しになっており、引数をそのまま状態だと見なすと遷移規則を得られる。よってこれまでの変換の結果、図 4 のような状態遷移規則を持つ抽象機械が得られたことになる。

この抽象機械は項 t, スタック s, 環境 e, 継続（コード列） c, 継続（コード列）リスト d を状態として持ち、Landin の SECD マシン [12] に類似しているが、

- 関数呼び出し時に環境をスタックに積む
- shift/reset 機能をサポートしている

の二点において異なる。

前者は、機械語による関数呼び出しの動作をほぼ模倣できていると考えられる。実際、機械語による関数呼び出しでは、後に使う可能性のあるレジスタの中身をスタックに退避させるが、ここで得られた抽象機械はほぼ同様のことを行っている。具体的には $t1$ を実行する際、後に $t0$ を実行するときに必要な値（環境に格納されている値）をスタックに退避させている。通常の関数呼び出しでは、関数呼び出し自体の前後でレジスタの退避復活

を行うが、ここでの結果は、部分式 $t1$ を実行する際（そしてそのときのみ）に退避復活をすれば良いことを示している。

また、後者についても機械語による shift/reset の実装を模倣できていると考えられる。実際、 $s@d$ (s と d を append したもの) が機械語実装でのスタックに相当すると考えれば、@ の位置がリセットの位置になり、 shift で最も近い reset までの継続を取り出せる。また、`CShift` の実行（抽象機械の下から四つ目の規則）によって、そのときのスタックとコードを `VCont` に入れているが、前者はスタックの内容をヒープにコピーすることに相当し、後者は継続のコードを表す番地を保存することに相当する。これは、ちょうど機械語の実装で継続を表すクロージャを生成 [13] することに対応している。よって、shift 命令によってスタックがヒープにコピーされることが抽象機械でモデル化できているといえる。この抽象機械は前述の通り、shift/reset の定義に従った実装から妥当性を検証した変換のみを用いて導出したものであるため、その正当性が保証されている。つまり、shift によるスタックコピーに対して、正当性の保証されたモデルが導かれたといえる。

ただし、現段階ではスタック s は値 v のリストとして実装されており、この v が $\texttt{VCont}(c, s)$ となるかもしれず、自己参照が起こる可能性が排除できない。そのため、実際の機械上での実装において、スタックを元のアドレスと異なる場所に復元する際、自己参照のアドレスを修正しながら複写しなければならなくなり、自明でない操作が要求される。この問題に対しては、`VCont` におけるスタックへの参照を先頭からの index で行うことにした上でその透明性を証明することで解決できると思われる。

10　まとめと今後の課題

本論文では、shift/reset を定義に基づいて実装したインタプリタに CPS 変換、非関数化、線形化、スタック導入、環境退避変換を施してスタックに値を退避するインタプリタを得られることを示した。CPS 変換、非関数化は先行研究によって既にその妥当性が検証済みであり、それ以外の変換の妥当性については本論文内で検証を行った。そして、このインタプリタから項 t, スタック s, 環境 e, 継続（コード列） c, 継続（コード列）リスト d を状態として持ち、shift/reset 命令をサポートする抽象機械の状態遷移規則が得られることを示した。

この抽象機械は Landin の SECD マシンに類似しているが、shift/reset が実装されている点と、環境を退避する点において異なる。後者は実際の機械語実装を模倣したためである。shift/reset についても shift によるスタックコピーと reset によるコピー範囲の限定がモデル化されたものとなっている。

今後、ここでの結果をもとに、さらに厳密に機械語による shift/reset の実装の正当性を確立していく予定である。現段階で得られている抽象機械は shift/reset の挙動を忠実に表しているが、まだインタプリタ実行の形である。今後、この抽象機械を Ager [1] らの手法に従って分割し、コンパイラと仮想機械の抽出を試みる予定である。本論文で得られた抽象機械では継続オブジェクトが項以外のデータを保持しないため、項さえ与えられれば継続（コード列）を生成することができるはずである。よって、抽象機械を「項を受け取ってコード列を返す」部分と「コード列、スタック、環境、継続を受け取って値を返す」部分に分割する。後者が前者に依存しないように分割できれば、前者をコンパイラ、後者を仮想機械と見なせる。本研究では、この方向を追求するとともに、分割時及び分割後に施す変換について、その妥当性の検証を与えながら行う。

具体的に施す変換としては、まずはカリー化が挙げられる。カリー化により、評価器が項を評価する部分とそれ以降に分割される。ただしカリー化しただけでは高階関数を生成するようになるため、非関数化を行う。すると命令オブジェクト i の型とその処理系が新たに定義される。この命令オブジェクトが仮想機械の命令列、処理系が仮想機械である。さらに i に対し線形リスト化等の変換を施していき、より低レベルな言語と対応がとりやすいものにしていく。予備的な考察では、shift/reset が入っても問題なくコンパイラ及び仮想機械が得られる見通しである。

最終的には、アセンブラでの実装と対応がとれる仮想機械を得て、shift/reset の機械語実装に対し正当性が保証された処理系を与えることを目指す。また、あわせて低レベルな実装を系統的に導く手法の確立を目指す。

謝辞

本稿を詳しく読んで下さり、大変有益なコメントを下さった査読者の方々に感謝いたします。

参考文献

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard: “From Interpreter to Compiler and Virtual Machine: A Functional Derivation,” Technical Report RS-03-14, BRICS, Aarhus, Denmark (March 2003).

[2] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard: "A Functional Correspondence between Evaluators and Abstract Machines," Technical Report RS-03-13, BRICS, Aarhus, Denmark (March 2003).

[3] K. Asai, and Y. Kameyama: "Polymorphic Delimited Continuations," *Proceedings of the Fifth Asian Symposium on Programming Languages and Systems (APLAS'07), LNCS 4807*, pp. 239–254 (November 2007).

[4] M. Biernacka, D. Biernacki, and O. Danvy "An Operational Foundation for Deliminated Continuations in the CPS Hierarchy," *Logical Methods in Computer Science*, Vol. 1 (2:5), pp. 1-39 (November 2005).

[5] O. Danvy, and A. Filinski: "A Functional Abstraction of Typed Contexts," Technical Report 89/12, DIKU, University of Copenhagen (July 1989).

[6] O. Danvy, and A. Filinski: "Abstracting Control," *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).

[7] O. Danvy, and K. Millikin: "A Rational Deconstruction of Landin's J Operator," Technical Report RS-06-17, BRICS, Aarhus, Denmark (December 2006).

[8] M. Felleisen: "The Theory and Practice of First-Class Prompts," *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 180–190 (January 1988).

[9] A. Filinski: "Representing Monads," *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pp. 446–457 (January 1994).

[10] A. Igarashi and M. Iwaki: "Deriving Compilers and Virtual Machines for a Multi-Level Languages," *Proceedings of the Fifth Asian Symposium on Programming Languages and Systems (APLAS'07), LNCS 4807*, pp. 206–221 (November 2007).

[11] 木谷有沙、浅井健一「限定継続を含む仮想機械導出のためのプログラム変換」お茶の水女子大学理学部情報科学科テクニカルレポート OCHA-IS 08-3 (February 2009).

[12] P. J. Landin: "The mechanical evaluation of expressions," *The Computer Journal*, Vol. 6, No. 4, pp. 308–320, (1964).

[13] 増子萌、浅井健一「MinCaml コンパイラにおける shift/reset の実装」第 11 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 (March 2009).

[14] R. Milner: "Communication and Concurrency" Prentice Hall International Series in Computer Science, (1995).

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

[16] J. C. Reynolds: "Definitional Interpreters for Higher-Order Programming Languages," *Proceedings of the ACM National Conference*, Vol. 2, pp. 717–740, (August 1972), reprinted in *Higher-Order and Symbolic Computation*, Vol. 11, No. 4, pp. 363–397, Kluwer Academic Publishers (December 1998).

[17] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten: "Revised^6 report on the algorithmic language Scheme", http://www.r6rs.org/ (2007).