例外と限定継続命令をサポートする評価器からの仮想機械とコンパイラの導出

増子萌 [†,‡]　　浅井健一 [†]
[†] お茶の水女子大学　[‡](独) 日本学術振興会特別研究員 DC
moe@pllab.is.ocha.ac.jp
asai@is.ocha.ac.jp

概 要

限定継続の命令 shift/reset を用いると，ユーザによるプログラムの実行順序の制御が可能となり，さまざまな応用例が考えられる。shift/reset の使用をサポートするため，我々は先行研究において Caml Light システムを変更し，shift/reset を直接実装した。基本的に shift/reset はスタックフレームのコピーで実装出来るが，Caml Light は例外処理をサポートするために例外用ポインタを利用しているので，単純なフレームのコピーでは実装出来ず，その正当性は保証されていなかった。すでに先行研究で λ 計算を shift/reset で拡張した言語のインタプリタから仮想機械とコンパイラが導出されているため，本研究ではこれを例外で拡張し，例外も含めた実装に正しさを保証しやすい形で仮想機械とコンパイラを導出した。

1　はじめに

継続とは，計算のある時点における残りの計算を表す概念である。プログラムで継続を扱えると，ユーザによるプログラムの実行順序の制御が可能となる。その応用例には，例外処理や大域的ジャンプ，非決定的プログラミング，Web プログラミングなどがある。

継続はプログラムを継続渡し形式 (Continuation Passing Style, CPS) で書けば明示的に扱える。しかし，プログラム全体を CPS で書くのは面倒であり，継続を容易に扱うために call/cc (R^6RS) や shift/reset (DF90) などの継続命令が考えられてきた。特に shift/reset は継続を扱う範囲を限定出来ること，切り取る継続が静的に決定されること，型システム (AK07; DF89) が存在することなどから扱いやすい。

shift/reset の使用を推進するため，我々は先行研究において Caml Light の実装を変更し，shift/reset をフルサポートするシステム OchaCaml を実装した (MA11)。この実装は MinCaml での実装 (MA09) における，reset のときスタックに印を積む，shift のとき直近の印までのスタックフレームを (ヒープに) 移動する，reset の印を積むときにはインバリアントを満たすようにする，という方針に基づいたものであり，実際多くのプログラムを動かせた。

上の基本的な方針については，すでに我々の研究グループでプログラム変換の手法を利用した正当性の証明が与えられている (AK10)。だが Caml Light は例外処理をサポートするために例外用ポインタを用いているので，単純なスタックフレームのコピーだけでは正しく実装出来ず，推測で実装していた部分が存在した。そこで本研究では，プログラム変換の対象言語を例外で拡張し，例外と限定継続命令をサポートする評価器から仮想機械とコンパイラを導出する。形式的な証明ではないものの，これにより我々の実装の例外を含めた部分にも正当性の証明を与えやすくなる。

関連研究　Ager らは CPS 変換や非関数化などの変換前後のプログラムの等価性が保証された変換のみを利用し，λ 計算インタプリタから CEK や CLS，SECD，CAM などの抽象機械が得られることを示した (ABDM03)。(AK10) では λ 計算を shift/reset で拡張した言語を対象として，彼らと同様の手法を適用した上で戻り番地の退避・復活の仕組みを導入し，最終的には仮想機械とコンパイラを導出しており，機械語レベルの shift/reset の直接実装に正当性を与えていた。本研究でも Ager らの手法を利用しつつ，戻り番地の退避・復活と同様の枠組みを導入する。

Ager らはエフェクトをモナドで表す評価器に対するプログラム変換も提案している (ADM04)。その一例としてエラーモナドも挙げられているが，最終的に得られた評価器ではエラー値は伝播される形になっており，エラー発生時にハンドラまでジャンプする挙動は表現されていない。

Biernacka らは 1 ステップの簡約戦略の定義から環境を用いる抽象機械を導出する手法を提案し，Krivine の抽象機械や CEK マシン，ZINC 抽象機械を導出している (BD07)。本研究では導出対象を ZINC に限定し，より実装に近い低レベルな部分まで導出しているが，彼らの手法では抽象度を落とさずに色々な機械が導出可能になっている。

概観　本論文の構成は以下の通りである。`shift/reset` は 2 章，ZINC は 3 章でそれぞれ簡単に説明する。4 章では λ 計算を例外と `shift/reset` で拡張した言語を対象言語として定め，それに対する CPS モナディックインタプリタを示す。このインタプリタに対して 5 章で各種のプログラム変換を施してコンパイラと仮想機械を導出する。得られた結果と ZINC および実装との対応を考察し，6 章でまとめと今後の課題を述べる。

2　`shift/reset` とは

`shift/reset` は Danvy と Filinski によって提案された限定継続の命令である (DF90; DF92)。直観的には，`shift` は現在の継続を切り取る命令，`reset` は `shift` が切り取る継続の範囲を限定する命令である。ここでは，`shift (fun` x `->` M`)` という式のとき x に対応する値が現在の継続であるとして M を空の継続で実行し (つまり現在の継続を「切り取る」)，`reset (fun () ->` M`)` という式のとき M を空の継続で実行するものとする。空の継続で実行することで，その `reset` で囲まれた `shift` が切り取る継続が M 内に限定される。

例えば，`1 + reset (fun () -> 2 * shift (fun k -> 3 + k 4))` という式の場合，`shift` で切り取るのは『2 を掛ける』という継続 (`2 * □`) であり，

```
1 + reset (fun () -> 2 * shift (fun k -> 3 + k 4)) ⇒ 1 + (3 + 2 * 4) ⇒ 12
```

となる。また，`shift` で切り取った継続は通常の関数のように複数回使うことも出来て，例えば `1 + reset (fun () -> 2 * shift (fun k -> k 3 + k 4))` は `1 + (2 * 3 + 2 * 4) = 15` となる。

例外と限定継続命令が共存する際の非自明な挙動については Kiselyov ら (KSS06) や Herman (Her07) によって指摘されている。例えば `E` を例外として，

```
try
  reset (fun () -> raise E + shift (fun k -> try k 0 with E -> 1))
with E -> 2
```

というプログラム*を考える。right-to-left で実行すると `shift` が切り取る継続は `raise E + □` で，`k 0` を実行した時点で例外 `E` が発生する。`reset` の継続に結果が渡ると最終結果は `2` になるが，(KSS06) の意味論に従えば結果は `try □ with E -> 1` に渡されるべきで，最終結果は `1` になる。

3　ZINC とは

ZINC (Ler90) は Caml Light (Ler97) のコアシステムであり，ZINC 抽象機械と対応するコンパイラから構成される†。ZINC 抽象機械は，最も単純には，アキュムレータ，環境，引数スタック，リターンスタックを用いて命令列を実行する。引数スタックには関数適用の引数となる値 (および関数適用の引数の『区切り』を表す印)，リターンスタックには戻り番地を表すリターンフレーム (コードポインタと環境の組) が積まれる。例外を含む実装では，トラップフレームと呼ばれるフレームもリターンスタックに保存される。`try` 文を実行するときにはトラップフレームを保存し，例外を起こすときには直近のトラップフレームまでのフレームを破棄する。

---

*これは (Her07) で挙げられている SML のプログラムの例を，OCaml で書いたものである。

†Ager らの定義 (ADM04) によれば，これは抽象機械ではなく仮想機械である。

$$
\begin{array}{rcl}
\mathcal{T}[\![n]\!] & = & \mathcal{C}[\![n]\!] \;=\; \mathsf{Access}(n) \\
\mathcal{T}[\![M\ N_1\ \cdots\ N_k]\!] & = & \mathcal{C}[\![N_k]\!];\ \mathsf{Push};\ \cdots;\ \mathcal{C}[\![N_1]\!];\ \mathsf{Push};\ \mathcal{C}[\![M]\!];\ \mathsf{Appterm} \\
\mathcal{C}[\![M\ N_1\ \cdots\ N_k]\!] & = & \mathsf{Pushmark};\ \mathcal{C}[\![N_k]\!];\ \mathsf{Push};\ \cdots;\ \mathcal{C}[\![N_1]\!];\ \mathsf{Push};\ \mathcal{C}[\![M]\!];\ \mathsf{Apply} \\
\mathcal{T}[\![\lambda M]\!] & = & \mathsf{Grab};\ \mathcal{T}[\![M]\!] \\
\mathcal{C}[\![\lambda M]\!] & = & \mathsf{Cur}(\mathcal{T}[\![M]\!];\ \mathsf{Return})
\end{array}
$$

$$
\begin{array}{rcl}
\langle \mathsf{Access}(n); c, a, v_0 \cdots v_n \cdots, s, r\rangle & \Rightarrow & \langle c, v_n, v_0 \cdots v_n \cdots, s, r\rangle \\
\langle \mathsf{Appterm}; c_0, (c_1, e_1), e_0, v.s, r\rangle & \Rightarrow & \langle c_1, (c_1, e_1), v.e_1, s, r\rangle \\
\langle \mathsf{Apply}; c_0, (c_1, e_1), e_0, v.s, r\rangle & \Rightarrow & \langle c_1, (c_1, e_1), v.e_1, s, (c_0, e_0).r\rangle \\
\langle \mathsf{Push}; c, a, e, s, r\rangle & \Rightarrow & \langle c, a, e, a.s, r\rangle \\
\langle \mathsf{Pushmark}; c, a, e, s, r\rangle & \Rightarrow & \langle c, a, e, \epsilon.s, r\rangle \\
\langle \mathsf{Cur}(c_1); c_0, a, e, s, r\rangle & \Rightarrow & \langle c_0, (c_1, e), e, s, r\rangle \\
\langle \mathsf{Grab}; c_0, a, e_0, \epsilon.s, (c_1, e_1).r\rangle & \Rightarrow & \langle c_1, (c_0, e_0), e_1, s, r\rangle \\
\langle \mathsf{Grab}; c, a, e, v.s, r\rangle & \Rightarrow & \langle c, a, v.e, s, r\rangle \\
\langle \mathsf{Return}; c_0, a, e_0, \epsilon.s, (c_1, e_1).r\rangle & \Rightarrow & \langle c_1, a, e_1, s, r\rangle \\
\langle \mathsf{Return}; c_0, (c_1, e_1), e_0, v.s, r\rangle & \Rightarrow & \langle c_1, (c_1, e_1), v.e_1, s, r\rangle
\end{array}
$$

Fig 1. ZINC のコンパイル規則と抽象機械の定義

図 1 に，ZINC のコンパイル規則および抽象機械の定義を示す。ここでは多引数の関数適用を許す，de Bruijn 記法の λ 計算を対象言語としている。$\mathcal{T}[\![-]\!]$ は末尾位置にある式のコンパイル規則，$\mathcal{C}[\![-]\!]$ は一般の式のコンパイル規則を表す。実行は命令列，アキュムレータ，環境，引数スタック，リターンスタックの五つ組の遷移として記述しており，命令列が空になった時点のアキュムレータの値が全体の結果となる。アキュムレータの (c, e) という組はクロージャ，. は要素の結合，ϵ は引数スタックの『区切り』を表す。Grab と Return は引数スタックの先頭要素が ϵ かどうかで挙動を変えており，これによりクロージャ生成が抑えられ，カリー化された関数も効率的に実行している。

4 対象言語

本論文では，型なし λ 計算を例外および `shift/reset` で拡張した言語を対象とする。

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

簡単のために，例外はひとつしか存在しないものとする。その例外を E とすれば，`raise_exn` は Caml Light や OCaml における `raise` E，`try`(t_0, t_1) は `try` t_0 `with` E `->` t_1 に対応する。

この言語の実行結果はエラー値か正常値 (関数抽象で生成されるクロージャ，または，`shift` で生成される継続) になるため，今回はエラーモナドを使って実装する。OCaml で記述した定義となる評価器 `eval1` をコード 1 に示す。これはエラーモナドを用いて例外をサポートしたインタプリタに CPS 変換を施し，`shift/reset` もサポートするようにしたプログラムに基づく。`g1` は関数抽象の本体部分，`h1` は入れ子の関数適用を実行する関数である。どちらも後の変換のために導入してあるが，今のところ `f1` と全く同じ定義になっている。関数適用の実行は Caml Light に合わせて call-by-value，right-to-left にしている。

`f1` (および `g1`，`h1`) はいずれも環境を使って項を実行する。通常のインタプリタの環境は連想リストなどで表されるが，ここでは変数名のリスト `xs` と値のリスト `e` の二つで表している。これは，前者は静的，後者は動的なデータなので，最終的に評価器をコンパイラと仮想機械に分割することを見据えた準備である。`Var x` のケースでは `offset x xs` により変数 `x` が変数名のリスト `xs` の何番目にあるかを調べ，`e` から `x` に対応する値を得ている。

```
1  type t =                                          (* term *)
2    | Var of string | Abs of string * t | App of t * t
3    | Exn | Try of t * t | Sft of string * t | Rst of t
4
5  type v = VFun of (v -> c -> er) | VCnt of c      (* value *)
6  and  c = er -> er                                 (* continuation *)
7  and er = S of v | E   (* monadic value *)        and e = v list    (* environment *)
8
9  let return v c = c (S v)                          (* monadic operators *)
10 let bind a c f = match a with S v -> f v c | E -> c E
11 let id a = a                                      (* initial cont *)
12
13 let rec f1 t xs e c = match t with
14   | Var x -> return (List.nth e (offset x xs)) c
15   | Abs (x, t) -> return (VFun (fun v -> g1 t (x :: xs) (v :: e))) c
16   | App (t0, t1) ->
17       f1 t1 xs e (fun a1 -> bind a1 c (fun v1 c ->
18       h1 t0 xs e (fun a0 -> bind a0 c (fun v0 c -> match v0 with
19         | VFun f -> f v1 c | VCnt k -> c (return v1 k)))))
20   | Exn -> c E
21   | Try (t0, t1) -> f1 t0 xs e (function S v -> return v c | E -> f1 t1 xs e c)
22   | Sft (x, t) -> f1 t (x :: xs) (VCnt c :: e) id              (* pack c as a value *)
23   | Rst t -> c (f1 t xs e id)          (* evaluate t under initial cont, then apply c *)
24 and g1 t xs e c = f1 t xs e c                     (* execute expressions under VFun *)
25 and h1 t xs e c = match t with                    (* execute nested applications *)
26   | App (t0, t1) ->
27       f1 t1 xs e (fun a1 -> bind a1 c (fun v1 c ->
28       h1 t0 xs e (fun a0 -> bind a0 c (fun v0 c -> match v0 with
29         | VFun f -> f v1 c | VCnt k -> c (return v1 k)))))
30   | _ -> f1 t xs e c
31
32 let eval1 t = f1 t [] [] id
```

Code 1. eval1: 定義となる CPS モナディックインタプリタ

return と bind はモナディックオペレータであり，ここでは例外を扱うために用いている。正常値 v は return を介して er 型の値 S v として継続に渡される。例外を発生させるとき (Exn のケース，20 行目) は，エラー値 E が継続に渡される。bind は第一引数が正常値 S v の場合，引数の関数 f に v (と c) を渡して計算を進める。一方，第一引数がエラー値 E の場合は f を捨てて継続 c に E を渡してしまう。そのため例外が発生したときには継続たちに E が伝播する形になる。Try (t0, t1) のケースでは，t0 の実行結果が正常値の場合それを結果として返し，エラー値の場合はその発生した例外を捉えて t1 を実行している (21 行目)。

shift/reset の定義については通常の CPS インタプリタと同様である。Sft (x, t) のケースでは，現在の継続を VCnt でパッケージ化して x に対応する値として環境に登録した上で，本体 t を初期継続 id の元で実行する。値継続 VCnt k に引数を渡すときは，まず k に受け取った引数を渡し，更にその結果を現在の継続に渡す。Rst t のケースでは，初期継続の元で t を実行し，その結果を現在の継続に渡す。これにより，shift で切り取られる継続の範囲を限定している。

エラー値を扱うために return や bind が現れてはいるが，基本的には一般的な CPS インタプリタと同じ形になっている。

5 プログラム変換

本章では，上に示した評価器に各種プログラム変換を施し，最終的に ZINC と実装に対応したコンパイラおよび仮想機械を導出する。3 章で触れた，Grab と Return で実現されている効率的な実装の挙動も導出しているが，本論文では例外と限定継続命令の挙動に注目しているので途中のステップは一部割愛している。ここでは ZINC や実装に合わせ，一階のプログラムで末尾呼び出しのみから成る関数の導出を目指し，それを仮想機械と呼ぶことにする。以下の各コードでは，網掛けのある主要な変換を適用している部分に注目されたい。

```
type v =              (* value *)
  | VFun of (v -> e list -> s -> c -> er) | VCnt of c * e list * s | VApp of v list
and  f =              (* frame *)
  | CApp0 of t * xs | CApp1 | CApp2 | CTry  of t * xs * e * e list * s
and  c = f list       (* continuation *)      and  e = v list       (* environment *)
and  s = v list       (* argument stack *)    and er = S of v | E   (* monadic value *)

let rec return v es s c = run_c2 c (S v) es s
and bind a es s c f = match a with S v -> f v es s c | E -> run_c2 c E es s

and run_c2 c a es s = match c, es, s with       (* dispatch function *)
  | [], [], [] -> a
  | CApp0 (t0, xs) :: c, e :: es, VApp vs :: s ->
      bind a es s c (fun v1 es s c -> h2 t0 xs e es (VApp (v1 :: vs) :: s) c)
  | CApp1 :: c, es, VApp [] :: s -> run_c2 c a es s
  | CApp1 :: c, es, VApp (v1 :: vs) :: s | CApp2 :: c, es, VApp (v1 :: vs) :: s ->
      bind a es s c (fun v0 es s c -> match v0 with
        | VFun f -> f v1 es (VApp vs :: s) c
        | VCnt (k, es', s') ->
            run_c2 (CApp1 :: c) (return v1 es' s' k) es (VApp vs :: s))
  | CTry (t1, xs, e, es, s) :: c, es', s' -> (match a with
      | S v -> return v es' s' c | E -> f2 t1 xs e es s c)

and f2 t xs e es s c = match t with
  | Var x -> return (List.nth e (offset x xs)) es s c
  | Abs (x, t) -> return (VFun (fun v -> g2 t (x :: xs) (v :: e))) es s c
  | App (t0, t1) -> f2 t1 xs e (e :: es) (VApp [] :: s) (CApp0 (t0, xs) :: c)
  | Exn -> run_c2 c E es s
  | Try (t0, t1) -> f2 t0 xs e es s (CTry (t1, xs, e, es, s) :: c)
  | Sft (x, t) -> f2 t (x :: xs) (VCnt (c, es, s) :: e) [] [] []
  | Rst t -> run_c2 c (f2 t xs e [] [] []) es s
and g2 t xs e es s c = match t, s with
  | Abs (x, t), VApp [] :: s ->
      run_c2 c (S (VFun (fun v -> g2 t (x :: xs) (v :: e)))) es s
  | Abs (x, t), VApp (v1 :: vs) :: s -> g2 t (x :: xs) (v1 :: e) es (VApp vs :: s) c
  | App (t0, t1), VApp vs :: s ->
      f2 t1 xs e (e :: es) (VApp vs :: s) (CApp0 (t0, xs) :: c)
  | _ -> f2 t xs e es s (CApp1 :: c)
and h2 t xs e es s c = match t with
  | App (t0, t1) -> f2 t1 xs e (e :: es) s (CApp0 (t0, xs) :: c)
  | _ -> f2 t xs e es s (CApp2 :: c)

let eval2 t = f2 t [] [] [] [] []
```

Code 2. eval2: スタック導入後の評価器

5.1 スタック導入

まずはじめに，継続の非関数化および線形リスト化，スタック導入を行う。

非関数化は高階関数を一階の関数で置き換える手法である。継続の非関数化は，評価器に現れる各継続に対応するコンストラクタを定義し，データ構造として表された継続を呼び出す (値を渡す) ための関数を定義することで実現される。例えば先の評価器では，初期継続，App のケースの二つの継続，および Try のケースの継続をコンストラクタとして定義する。各コンストラクタには，対応する継続に含まれる自由変数を引数として保持させる。これにより継続内部に現れる自由変数が明示化される。以下は非関数化のみを施した場合の c の定義の抜粋である (CApp0 は eval1 の App のケースの (fun a1 -> ...)，CApp1[‡] は (fun a0 -> ...) の継続に対応する)。

```
and c = C0 | CApp0 of t * xs * e * c | CApp1 of v list * c
      | CTry of t * xs * e * c
```

上の定義は，C0 以外のコンストラクタには c がひとつずつ含まれているので，リストと同じ構造といえる。この観察に基づくと，継続をフレームの連なりによるリスト構造で表せる (線形リスト化)。さらに継続のフレームに含まれる (動的な) 環境 e と引数のリスト (CApp1 vs の vs) をそ

[‡]本来 CApp1 の引数は v * c にすべきだが，関数適用の引数をまとめて扱うため，v ではなく v list にしている。対象言語では多引数の関数適用を陽に許していないが，これによりネストした関数適用の引数などもまとめて扱える形になる。ZINC の対象言語との違いを吸収し，Grab と Return に対応する挙動を導出するために必要となっている。

れぞれ別のリストに保存・復活する形にすると，前者は環境のリスト，後者は引数のスタックになり，ZINC の引数スタックに相当するスタックが導入される。

以上の変換を順に施して得られる評価器 eval2 をコード 2 に示す。run_c2 は継続リストに値を渡す関数である。引数として増えた es は環境 e のリスト，s は引数スタックである。新しい値 VApp は先の CApp1 の引数のひとつ，v list を引数スタックで扱うために導入されており，関数適用の引数を保持する。また継続フレームに導入された CApp2 はほぼ CApp1 と同じものである。異なるのは，CApp2 が継続リストの先頭にある場合，引数スタックの先頭に VApp (v1 :: vs) とひとつ以上の引数が保存されていると保証される点である。g2 の Abs (x, t) の二つのケースは，もともと c = CApp1 :: c の場合に対応する。

```
    g2 (Abs (x, t)) xs e es (VApp vs :: s) (CApp1 :: c)
 =  return (VFun f) es (VApp vs :: s) (CApp1 :: c)  (f = fun v -> g2 t (x :: xs) (v :: e))
 =  run_c2 (CApp1 :: c) (S (VFun f)) es (VApp vs :: s)
    { run_c2 c (S (VFun f)) es s     (vs = [])
 =  { f v1 es (VApp vs :: s) c       (vs = v1 :: vs)
    {   = g2 t (x :: xs) (v1 :: e) es (VApp vs :: s) c
```

と，return や run_c2，bind を関数展開すれば等価であると確認出来る。この CApp1 は途中の変換で g2 へ渡される継続の先頭に暗黙に入っていると仮定されるためなくなっている (その代わり，g2 の _ の場合は継続リストの先頭に付与する。CApp2 と h2 についても同様の変換が施されている)。

Grab や Return の挙動を導出するための変換を加えてはいるが，本節の変換については (AK10) におけるスタック導入までの変換とほぼ同様であり，eval1 と eval2 の等価性は保証される。また，例外について特別な扱いはしていない。

5.2 例外伝播の最適化

次に，例外伝播を最適化しよう。継続がエラー値と正常値のどちらを受け取るかに着目し，具体的には，run_c2 の第二引数 a が S v の場合と E の場合に分けて関数を定義する。再帰呼び出しをしているところでは，必要ならば値にパターンマッチしながら適切な関数を呼び出す。

まず a = S v の場合，bind (S v) es s c f = f v es s c だから，継続リストの先頭フレームが CApp0，CApp1，CApp2 のケースのはじめの bind は展開出来る。また先頭フレームが CTry のケースでは，フレームの引数である項およびそれを評価するための情報を捨てて，残りの継続リストに v を渡して実行を進めることになる。一方 a = E の場合，bind E es s c f = run_c2 c E es s だから，継続リストの先頭フレームが CApp0，CApp1，CApp2 のケースでは，はじめの bind の第五引数の関数は無視され，残りの継続リストに E が渡される。これは先頭フレームが CTry になるまで続き，そのときにはフレームの引数の項が実行される (継続リストに CTry が含まれない場合は E が結果として返る)。つまり，run_c2 の第二引数として E を渡す場合，CTry のフレーム以外の情報は無視して実行しても意味は変わらない。

この最適化を導入した評価器 eval3 をコード 3 に示す。run_c3 は引数が S v の場合，exn は引数が E の場合にあたる。bind については第一引数に関する場合分けにより定義を展開したため不要になっている。return については run_c3 のエイリアスになったのでそれで置き換えた。切り取った継続の呼び出しと reset の実行のときにはマッチ文を使いながら適切な関数を呼び出している。exn により，E が継続内を伝播せずに最終結果になる (つまりエラーが返る)，あるいは CTry の実行までジャンプする (つまりエラーが飛んでハンドラで捕獲される) 挙動が実現されている。

これは Ager らが (ADM04) で整数二分木の乗算を例にとって示している最適化 “Propagating vs. stopping” (ノードに 0 が見つかったとき，伝播するか，すぐに 0 を返すか) に相当する変換と見なせる。彼らは継続を二つに分解して最適化を実現しており，本節の関数の複製と本質的な違いはない (次節では我々も継続を二つに分割する。例外ハンドラのリストが自然に導かれることに注目されたい)。

```
let rec run_c3 c v es s = match c, es, s with    (* run_c2 for a = S v *)
  | [], [], [] -> S v                            (* binds and returns are disappeared *)
  | CApp0 (t0, xs) :: c, e :: es, VApp vs :: s -> h3 t0 xs e es (VApp (v :: vs) :: s) c
  | CApp1 :: c, es, VApp [] :: s -> run_c3 c v es s
  | CApp1 :: c, es, VApp (v1 :: vs) :: s | CApp2 :: c, es, VApp (v1 :: vs) :: s ->
      (match v with
        | VFun f -> f v1 es (VApp vs :: s) c
        | VCnt (k, es', s') -> (match run_c3 k v1 es' s' with
                                  | S v -> run_c3 (CApp1 :: c) v es (VApp vs :: s)
                                  | E -> (* exn (CApp1 :: c) = *) exn c))
  | CTry _ :: c, es, s -> run_c3 c v es s
and exn c = match c with                         (* run_c2 for a = E, optimized *)
  | CTry (t, xs, e, es, s) :: c -> f3 t xs e es s c
  | [] -> E | _ :: c -> exn c

and f3 t xs e es s c = match t with
  | Var x -> run_c3 c (List.nth e (offset x xs)) es s
  | Abs (x, t) -> run_c3 c (VFun (fun v -> g3 t (x :: xs) (v :: e))) es s
  | App (t0, t1) -> f3 t1 xs e (e :: es) (VApp [] :: s) (CApp0 (t0, xs) :: c)
  | Exn -> exn c
  | Try (t0, t1) -> f3 t0 xs e es s (CTry (t1, xs, e, es, s) :: c)
  | Sft (x, t) -> f3 t (x :: xs) (VCnt (c, es, s) :: e) [] [] []
  | Rst t -> (match f3 t xs e [] [] [] with S v -> run_c3 c v es s | E -> exn c)
and g3 t xs e es s c = match t, s with
  | Abs (x, t), VApp [] :: s -> run_c3 c (VFun (fun v -> g3 t (x :: xs) (v :: e))) es s
  | Abs (x, t), VApp (v1 :: vs) :: s -> g3 t (x :: xs) (v1 :: e) es (VApp vs :: s) c
  | App (t0, t1), VApp vs :: s ->
      f3 t1 xs e (e :: es) (VApp vs :: s) (CApp0 (t0, xs) :: c)
  | _ -> f3 t xs e es s (CApp1 :: c)
and h3 t xs e es s c = match t with
  | App (t0, t1) -> f3 t1 xs e (e :: es) s (CApp0 (t0, xs) :: c)
  | _ -> f3 t xs e es s (CApp2 :: c)

let eval3 t = f3 t [] [] [] [] []
```

Code 3. eval3: 例外伝播を最適化した評価器

5.3 環境リストの除去と末尾呼び出し最適化

さて，ここまでの評価器では，関数適用など複数の項を実行する必要がある場合，項をひとつ実行する前に環境を環境リストに保存し，残りの項を実行するときに保存しておいた環境を復活して使用していた。しかし実際に環境を変更するのは VFun の場合だけである。その実行後に使用する環境，つまり継続で使う環境を適切なものにすれば環境の保存は必要なくなる。

継続を分割し，その一部を関数に戻した上で環境の保存を暗黙のものにして環境リストを除去した評価器 eval4 をコード 4 に示す。app の VFun (i, e') のケースの i に渡す継続部分で，引数の環境を無視して外側の環境を使用している (46 行目)。これにより，変更前の環境の元での実行を実現している。継続は例外とそれ以外で分けており，前者は failure continuation に，後者は success continuation に相当し，後者のみ関数に戻している。残された継続リストは例外ハンドラのフレーム (トラップフレーム) のスタックになる (リストでは定義が循環してしまうため新たにデータ構造を定義している)。また VFun は命令と環境の組で表す形に変更した。

この評価器では末尾呼び出し最適化も導入している。具体的には g4 に渡される継続に着目する。前述のように，app の VFun (i, e') のケースの i に渡される継続は受けとる環境を無視する。実際に g4 に継続を渡すのはこのケースだけなので，g4 を実行するときの継続は必ず受け取る環境を無視すると保証出来る。そのため，g4 の Abs の二つ目のケース (26 行目) では引数の c をそのまま使って再帰呼び出ししている。末尾呼び出し最適化に相当する挙動は i4 と app' の導入により実現されている。これらはそれぞれ h4 と app とほとんど同じ関数で，違いは app' の VFun のケース (52 行目) に現れている。app の VFun のケースと異なり c が受け取る環境を無視すると分かっているので，継続を (fun v' _ s' -> c v' e s') と更新することなく，そのまま利用している。

継続リストの除去は caller/callee save の違いに相当し，正当性もこれに依る。末尾呼び出し最適化については上のように継続の使われ方に着目すれば評価器の意味は変わらないと言えるだろう。

```
type v = VFun of i * e | VCnt of c * f * e * s | VApp of v list
and  c = v -> e -> s -> f -> er                  (* (success) cont, refuctionalized *)
and  f = Nil | Cons of (t * xs * e * s * c) * f  (* failure cont *)
and  e = v list          and s = v list          and er = S of v | E
and  i = e -> s -> c -> f -> er                  (* instruction *)

let idc = fun v _ [] _ -> S v

let rec exn f = match f with
  | Nil -> E
  | Cons ((t, xs, e, s, c), f) -> f4 t xs e s c f
and f4 t xs e s c f = match t with
  | Var x -> c (List.nth e (offset x xs)) e s f
  | Abs (x, t) -> c (VFun (g4 t (x :: xs), e)) e s f
  | App (t0, t1) ->
      f4 t1 xs e (VApp [] :: s) (fun v1 e (VApp vs :: s) ->
      h4 t0 xs e (VApp (v1 :: vs) :: s) c) f
  | Exn -> exn f
  | Try (t0, t1) ->
      f4 t0 xs e s (fun v e s (Cons (_, f)) -> c v e s f)
                   (Cons ((t1, xs, e, s, c), f))
  | Sft (x, t) -> f4 t (x :: xs) ((VCnt (c, f, e, s)) :: e) [] idc Nil
  | Rst t -> (match f4 t xs e [] idc Nil with S v -> c v e s f | E -> exn f)
and g4 t xs e s c = match t, s with
  | Abs (x, t), VApp [] :: s -> c (VFun (g4 t (x :: xs), e)) e s
  | Abs (x, t), VApp (v1 :: vs) :: s -> g4 t (x :: xs) (v1 :: e) (VApp vs :: s) c
  | App (t0, t1), VApp vs :: s ->
      f4 t1 xs e (VApp vs :: s) (fun v1 e (VApp vs :: s) ->
      i4 t0 xs e (VApp (v1 :: vs) :: s) c)       (* call i4, not h4 *)
  | _ -> f4 t xs e s (app' c)                    (* call app', not app *)
and h4 t xs e s c = match t with
  | App (t0, t1) ->
      f4 t1 xs e s (fun v1 e (VApp vs :: s) ->
      h4 t0 xs e (VApp (v1 :: vs) :: s) c)
  | _ -> f4 t xs e s (fun v e (VApp (v1 :: vs) :: s) ->
         app c v e (VApp (v1 :: vs) :: s))
and i4 t xs e s c = match t with                 (* duplication of h4 *)
  | App (t0, t1) ->
      f4 t1 xs e s (fun v1 e (VApp vs :: s) ->
      i4 t0 xs e (VApp (v1 :: vs) :: s) c)
  | _ -> f4 t xs e s (fun v e (VApp (v1 :: vs) :: s) ->
         app' c v e (VApp (v1 :: vs) :: s))      (* call app', not app *)
and app c v e s f = match s with
  | VApp [] :: s -> c v e s f
  | VApp (v1 :: vs) :: s -> (match v with
      | VFun (i, e') -> i (v1 :: e') (VApp vs :: s) (fun v' _ s' -> c v' e s') f
      | VCnt (k, f', e', s') ->
          (match k v1 e' s' f' with S v -> app c v e (VApp vs :: s) f | E -> exn f))
and app' c v e s f = match s with                (* duplication of app *)
  | VApp [] :: s -> c v e s f
  | VApp (v1 :: vs) :: s -> (match v with
      | VFun (i, e') -> i (v1 :: e') (VApp vs :: s) c f (* cont is reused as-is, opt *)
      | VCnt (k, f', e', s') ->
          (match k v1 e' s' f' with S v -> app' c v e (VApp vs :: s) f | E -> exn f))

let eval4 t = f4 t [] [] [] idc Nil
```

Code 4. eval4: 環境リストを除去し，末尾呼び出し最適化を施した評価器

5.4 コンビネータの抽出とリターンスタックの導入

これまでの変換でほぼ仮想機械のような評価器が得られているが，eval4 には reset の実行と切り取った継続の呼び出しの部分に非末尾の呼び出しが存在する。最終的に末尾呼び出しでのみ構成されている評価器を導出したいので，プログラム全体に CPS 変換を施し，2CPS にする。これにより，実行が継続 (またはメタ継続) で連なる形になり，コンビネータの合成として評価器を記述することが可能になる。CPS 変換を施してコンビネータを抽出し，リターンスタックを導入した評価器 eval5 をコード 5 に示す。各コンビネータは動的な値を扱うもので抽象機械の命令に相当し，静的な値にマッチしてコンビネータで実行を記述している部分 (f5 など) はコンパイラに相当する。

eval4 までの評価器の引数スタックは VApp vs_1 :: VApp vs_2 :: ... という形になっていたが，ここでは ZINC に合わせて印 VMark を導入してリストの切れ目を表すようにした。上のスタックは vs_1 @ [VMark] @ vs_2 @ [VMark] @ ... という形になる。継続が値 v を受け取ることから f5 t xs も v を受け取る形になっているが，これを実際に使うのは push および shift で構成する c だけであり，その他の場合はダミー値を渡しても結果は変わらない。

また eval4 では切り取った継続に複数引数が渡されていた場合に対応するため，app (および app') は再帰的に定義されていた。しかし仮想機械上には再帰的な命令は存在しないので，この再帰は解消したい。そこで eval5 では，shift で現在の継続をパッケージ化する際に残りの仕事とそれを実行するのに必要な値を含めた形にすることで，apply を再帰のないコンビネータにしている。これにより，app' で再帰呼び出ししていた部分 (eval5 の appterm に相当) では app を実行することになってしまうが，継続の更新が無駄になるだけであり，意味上は問題ない。

リターンスタックは，継続と環境の組のリストである (型 f と同様，リストでは定義が循環してしまうため新たにデータ構造を定義している)。先の評価器の継続で (fun v' _ s' -> c v' e s') となっていたところ (eval5 の 33 行目，apply の VFun の場合であり，eval4 では 46 行目に対応) でリターンスタックに c と e を保存している。継続が実際に使われていたところでスタックから復活すれば，正しい継続と環境を使って実行出来る。VFun の本体部分は g5 で生成されるので，具体的には g5 の各場合分けの末尾で呼ばれるコンビネータ grab と return，appterm (i5 を経て到達する可能性がある) のケースでリターンスタックの要素を復活すれば良く，実際に 24，27，31，39 行目で復活されている。要素を復活するべき場所は多くなっているが，本質的には (AK10) で導入されている戻り番地の退避・復活と同様の変換である。評価器の各部分を追うと，使われる継続と環境が eval4 と変わっていないと確認出来る。

5.5 仮想機械とコンパイラへの分割

最後に評価器を仮想機械とコンパイラに分割し，ZINC や実装の挙動に合わせる変換を行う。

先の評価器で，v は ZINC のアキュムレータ，e は環境，s は引数スタック，r および f はリターンスタックにあたる (r はリターンフレームだけ，f はトラップフレームだけ保存するスタックと見なせる)。そこで r と f をひとつにまとめる。また，すでにコンビネータの集合が評価器，コンビネータを用いて評価器を記述している部分がコンパイラとも見なせるが，より明示的に分割するため，継続とメタ継続に非関数化および線形リスト化を施す。これらの変換によって得られる評価器 eval6 をコード 6 に示す。もともと r と f は干渉しないので，リターンスタックは単なる合併で導出される。reset と切り取った継続の呼び出しの実行の際に加えられるダンプに含まれる自由変数の型は共に c，e，s，r なので，DRst と DCnt は c * e * s * r を引数として持つべきだが，ここでは c と e は FRet (c, e) :: r という形でリターンスタックに保存して渡しており，引数は s * r になっている。こうすると IReturn と IAppterm の VCnt のケースは，

```
| VCnt (c', e', s', r') ->
    let FRet (c, e) :: r = r in
    run_code c' v1 e' s' r' (DCnt (s, FRet (c, e) :: r) :: d)
```

```
type v =
  | VFun of i * e
  | VCnt of (v -> e -> s -> r -> f -> c -> e -> s -> r -> f -> d -> er) * f * e * s * r
  | VMark
and  c = v -> e -> s -> r -> f -> d -> er
and  f = FNil | FCons of (i * e * s * r * c) * f
and  e = v list          and s = v list         and er = S of v | E
and  i = v -> e -> s -> r -> c -> f -> d -> er  and  d = er -> er
and  r = RNil | RCons of c * e * r              (* return stack *)

let idc = fun v _ [] _ _ d -> d (S v)

let (>>) i0 i1 = fun v e s r c -> i0 v e s r (fun v e s r -> i1 v e s r c)
let access n = fun _ e s r c -> c (List.nth e n) e s r
let push_closure i = fun _ e s r c -> c (VFun (i, e)) e s r
let push_mark = fun v e s r c -> c v e (VMark :: s) r
let exn = fun v _ _ _ _ f d -> match f with
  | FNil -> d E | FCons ((i, e, s, r, c), f) -> i v e s r c f d
let trywith i0 i1 = fun v e s r c f ->
  i0 v e s r (fun v e s r (FCons (_, f)) -> c v e s r f)
             (FCons ((i1, e, s, r, c), f))
let push = fun v e s r c -> c v e (v :: s) r
let grab i = fun v e s r c -> match s with
  | VMark :: s -> let RCons (c, e', r) = r in c (VFun (i, e)) e' s r     (* pup r *)
  | v1 :: s -> i v (v1 :: e) s r c
let return = fun v e s r c -> match s with
  | VMark :: s -> let RCons (c, e, r) = r in c v e s r                   (* pop r *)
  | v1 :: s -> (match v with
      | VFun (f, e') -> f v (v1 :: e') s r c
      | VCnt (k, f', e', s', r') ->
          let RCons (c, e, r) = r in k v1 e' s' r' f' c e s r)           (* pop r *)
let apply = fun v e (v1 :: s) r c -> match v with
  | VFun (f, e') -> f v (v1 :: e') s (RCons (c, e, r))                   (* push r *)
                      (fun v _ _ _ _ _ -> S v)                           (* dummy cont *)
  | VCnt (k, f', e', s', r') -> k v1 e' s' r' f' c e s r
let appterm = fun v e (v1 :: s) r c -> match v with
  | VFun (f, e') -> f v (v1 :: e') s r c
  | VCnt (k, f', e', s', r') ->
      let RCons (c, e, r) = r in k v1 e' s' r' f' c e s r                (* pop r *)
let shift i = fun v e s r c f ->
  let c = fun v' e' s' r' f' c' e s r f d ->
    c v' e' s' r' f' (function
      | S v -> (match s with
                  | VMark :: s -> c' v e s r f d
                  | _ :: _ -> apply v e s r c' f d)
      | E -> (match f with FNil -> d E | FCons ((i, e, s, r, c), f) -> i v' e s r c f d))
  in let c = VCnt (c, f, e, s, r)
  in i v (c :: e) [] RNil idc FNil
let reset i = fun v e s r c f d ->
  i v e [] RNil idc FNil (function
    | S v -> c v e s r f d
    | E -> (match f with FNil -> d E | FCons ((i, e, s, r, c), f) -> i v e s r c f d))

let rec f5 t xs = match t with
  | Var x -> access (offset x xs)
  | Abs (x, t) -> push_closure (g5 t (x :: xs))
  | App (t0, t1) -> push_mark >> f5 t1 xs >> push >> h5 t0 xs
  | Exn -> exn
  | Try (t0, t1) -> trywith (f5 t0 xs) (f5 t1 xs)
  | Sft (x, t) -> shift (f5 t (x :: xs))
  | Rst t -> reset (f5 t xs)
and g5 t xs = match t with
  | Abs (x, t) -> grab (g5 t (x :: xs))
  | App (t0, t1) -> f5 t1 xs >> push >> i5 t0 xs
  | _ -> f5 t xs >> return
and h5 t xs = match t with
  | App (t0, t1) -> f5 t1 xs >> push >> h5 t0 xs
  | _ -> f5 t xs >> apply
and i5 t xs = match t with
  | App (t0, t1) -> f5 t1 xs >> push >> i5 t0 xs
  | _ -> f5 t xs >> appterm

let eval5 t = f5 t [] VMark [] [] RNil idc FNil (fun a -> a)
```

Code 5. eval5: CPS 変換後にコンビネータを抽出し，リターンスタックを導入した評価器

```
type v = VFun of i list * e | VCnt of c * e * s * r | VMark
and  c = i list      and e = v list      and s = v list     and er = S of v | E
and  i =                                 (* instructions (continuation frame) *)
  | IAccess of int   | ICur of i list    | IGrab of i list
  | IPushmark        | IPush  | IReturn  | IApply   | IAppterm
  | IExn             | IPushtrap of i list * i list | IPoptrap
  | IShift of i list | IReset of i list
and  m = DRst of s * r | DCnt of s * r          and d = m list       (* linealized dump *)
and  f = FRet of c * e | FTry of c * v * e * s  and r = f list       (* return stack *)

let rec run_dump d a = match d with      (* dispatch function of dump *)
  | [] -> a
  | DRst (s, FRet (c, e) :: r) :: d ->
      (match a with S v -> run_code c v e s r d | E -> search_until r d)
  | DCnt (s, FRet (c, e) :: r) :: d ->
      (match a with
         | S v -> (match s with
                   | VMark :: s -> run_code c v e s r d
                   | _ :: _ -> run_code (IApply :: c) v e s r d)
         | E -> search_until r d)
and search_until r d = match r with      (* skip frames *)
  | FTry (c, v, e, s) :: r -> run_code c v e s r d
  | [] -> run_dump d E | FRet _ :: r -> search_until r d
and run_code c v e s r d = match c with  (* run continuation (instruction sequence) *)
  | [] -> run_dump d (S v)
  | IAccess n :: c -> run_code c (List.nth e n) e s r d
  | ICur i :: c -> run_code c (VFun (i, e)) e s r d
  | IGrab i :: c -> (match s with
      | VMark :: s -> let FRet (c, e') :: r = r in run_code c (VFun (i, e)) e' s r d
      | v1 :: s -> run_code (i @ c) v (v1 :: e) s r d)
  | IPushmark :: c -> run_code c v e (VMark :: s) r d
  | IPush :: c -> run_code c v e (v :: s) r d
  | IReturn :: c -> (match s with
      | VMark :: s -> let FRet (c, e) :: r = r in run_code c v e s r d
      | v1 :: s ->
          (match v with
             | VFun (f, e') -> run_code f v (v1 :: e') s r d
             | VCnt (c', e', s', r') -> run_code c' v1 e' s' r' (DCnt (s, r) :: d)))
  | IApply :: c ->
      let v1 :: s = s in
      (match v with
         | VFun (f, e') -> run_code f v (v1 :: e') s (FRet (c, e) :: r) d
         | VCnt (c', e', s', r') ->
             run_code c' v1 e' s' r' (DCnt (s, FRet (c, e) :: r) :: d))
  | IAppterm :: c ->
      let v1 :: s = s in
      (match v with
         | VFun (f, e') -> run_code f v (v1 :: e') s r d
         | VCnt (c', e', s', r') -> run_code c' v1 e' s' r' (DCnt (s, r) :: d))
  | IExn :: _ -> search_until r d
  | IPushtrap (i1, i0) :: c -> run_code (i0 @ c) v e s (FTry (i1 @ c, v, e, s) :: r) d
  | IPoptrap :: c -> let FTry _ :: r = r in run_code c v e s r d
  | IShift i :: c -> let c = VCnt (c, e, s, r) in run_code i v (c :: e) [] [] d
  | IReset i :: c -> run_code i v e [] [] (DRst (s, FRet (c, e) :: r) :: d)

let rec f6 t xs = match t with
  | Var x -> [IAccess (offset x xs)]
  | Abs (x, t) -> [ICur (g6 t (x :: xs))]
  | App (t0, t1) -> [IPushmark] @ f6 t1 xs @ [IPush] @ h6 t0 xs
  | Exn -> [IExn]
  | Try (t0, t1) -> [IPushtrap (f6 t1 xs, f6 t0 xs @ [IPoptrap])]
  | Sft (x, t) -> [IShift (f6 t (x :: xs))]
  | Rst t -> [IReset (f6 t xs)]
and g6 t xs = match t with
  | Abs (x, t) -> [IGrab (g6 t (x :: xs))]
  | App (t0, t1) -> f6 t1 xs @ [IPush] @ i6 t0 xs
  | _ -> f6 t xs @ [IReturn]
and h6 t xs = match t with
  | App (t0, t1) -> f6 t1 xs @ [IPush] @ h6 t0 xs
  | _ -> f6 t xs @ [IApply]
and i6 t xs = match t with
  | App (t0, t1) -> f6 t1 xs @ [IPush] @ i6 t0 xs
  | _ -> f6 t xs @ [IAppterm]

let eval6 t = run_code (f6 t []) VMark [] [] [] []
```

Code 6. eval6: 仮想機械とコンパイラに分割した評価器

となり，リターンスタックから復活した組をすぐに保存することになるので，

```
| VCnt (c', e', s', r') -> run_code c' v1 e' s' r' (DCnt (s, r) :: d)
```

と r にマッチせずに実行しても結果は同じになる。これにより 関数や継続の呼び出しをしているすべての場所で，リターンスタックへリターンフレーム (FRet) を保存するか，しないかが VFun と VCnt のケースで合致する。

eval6 ではダンプフレームが二つに分かれているが，実装上は reset の残りの仕事と切り取った継続の残りの仕事は区別せずに実行しているので，ひとつにまとめたい。run_dump の定義をみると，ダンプが DRst (s, FRet (c, e) :: r) :: d と DCnt (VMark :: s, FRet (c, e) :: r) :: d の場合の実行結果は一致している。よって DRst が保持する引数スタック，すなわち IReset i :: c を実行するときの引数スタックの先頭に VMark が入れば挙動を変えずにふたつのダンプフレームをまとめられる。この観察に基づいてダンプフレームをまとめ，新たな命令 ICnt の導入により VFun と VCnt もまとめた評価器 eval7 をコード 7 に示す。f7 の Rst t のケースが [IReset (f6 t xs)] から [IPushmark; IReset (f7 t xs)] となっていることに注目されたい。また h7 と i7 に Rst t のケースを導入している。これは eval6 の run_dump の定義で，ダンプが DCnt (s, r) :: d で s の先頭が VMark でない場合 (20 行目) を見ると，IApply :: c が実行されるためである。((reset (fun () -> ...)) t_1 ...) と，reset 式全体が関数適用の関数部分になっている場合を考えると，VMark がスタックの先頭になければ IApply 命令が実行されるため，h7 と i7 の Rst のケースでは単に [IReset (f7 t xs)] とし，IPushmark も IApply (および IAppterm) も入れないコードを生成している。run_dump でダンプが (s, r) :: d のケースは eval6 の DCnt のケースに相当する。IApply ではなく IReturn を使っているのは OchaCaml での実装に合わせた変更だが，

```
  run_code (IApply :: c) v e s r d
= let v1 :: s = s in
  (match v with
     VFun (f, e') -> run_code f v (v1 :: e') s (FRet (c, e) :: r) d)

  run_code [IReturn] v [] s (FRet (c, e) :: r) d
= match s with
    | VMark :: s -> run_code c v e s r d
    | v1 :: s -> (match v with
        VFun (f, e') -> run_code f v (v1 :: e') s (FRet (c, e) :: r) d)
```

とそれぞれの場合を展開すれば等価であると確認出来る。

VFun と VCnt の合併については，先の変換で c と e を r に積んで渡す形にした結果，VFun と VCnt のリターンフレーム (FRet) を保存する位置が一致したため，ほぼ VCnt を ICnt で置き換えるだけで実現出来ている。ICnt の継続を実行するときに使うべき引数は VFun を実行する際に環境の先頭に保存されているので，ICnt ではそれを e から復活している。

継続とメタ継続の非関数化・線形リスト化の正当性は 5.1 節と同様，先行研究 (AK10) に従う。その他の変換については，上のように変換後の定義を展開すると意味が変わらないことは確認される。

5.6 考察

最後に，得られた仮想機械とコンパイラを ZINC および OchaCaml の実装との対応から考察する。

eval7 の仮想機械とコンパイラを図 1 と同様の形で書き下すと図 2 のようになる。図 1 と同じ命令については仮想機械の定義を省略しており，λ 計算の部分のコンパイル規則，およびそのコンパイル結果に出現する命令に関する規則については図 1 に合致する[§]。Pushtrap のケースでリターンスタックに保存される三つ組 (c, e, s) はトラップフレームを表す。

[§]導出は普通の λ 計算のインタプリタからスタートしたが，offset を利用して環境の何番目の変数かを調べている部分を変数を表す数に置き換えれば de Bruijn 記法に対応出来る。

```
type v = VFun of i list * e | VMark       (* VCnt is disappeared *)
and  c = i list  (* code *)               and  e = v list       (* environment *)
and  s = v list  (* (argument) stack *)   and er = S of v | E   (* monadic value *)
and  i =         (* instruction *)
  | IAccess of int   | ICur of i list    | IGrab
  | IPushmark        | IPush  | IReturn  | IApply   | IAppterm
  | IExn             | IPushtrap of i list * i list | IPoptrap
  | IShift of i list | IReset of i list  | ICnt of c * e * s * r
and  d = (s * r) list                     (* frame of dump is undistingished *)
and  f = FRet of c * e | FTry of c * v * e * s   and r = f list   (* return stack *)

let rec run_dump d a = match d with
  | [] -> a
  | (s, r) :: d -> (match a with
      | S v -> run_code [IReturn] v [] s r d | E -> search_until r d)
and search_until r d = match r with
  | [] -> run_dump d E | FRet _ :: r -> search_until r d
  | FTry (c, v, e, s) :: r -> run_code c v e s r d
and run_code c v e s r d = match c with
  | [] -> run_dump d (S v)
  | IAccess n :: c -> run_code c (List.nth e n) e s r d
  | ICur i :: c -> run_code c (VFun (i, e)) e s r d
  | IGrab :: c -> (match s with           (* IGrab needs not hold its argument *)
      | VMark :: s ->                     (* since g7 locates in tail position *)
          let FRet (c', e') :: r = r in run_code c' (VFun (c, e)) e' s r d
      | v1 :: s -> run_code c v (v1 :: e) s r d)
  | IPushmark :: c -> run_code c v e (VMark :: s) r d
  | IPush :: c -> run_code c v e (v :: s) r d
  | IReturn :: c -> (match s with
      | VMark :: s ->
          let FRet (c, e) :: r = r in run_code c v e s r d
      | v1 :: s -> (match v with VFun (f, e') -> run_code f v (v1 :: e') s r d))
  | IApply :: c ->
      let v1 :: s = s in
      (match v with VFun (f, e') -> run_code f v (v1 :: e') s (FRet (c, e) :: r) d)
  | IAppterm :: c ->
      let v1 :: s = s in
      (match v with VFun (f, e') -> run_code f v (v1 :: e') s r d)
  | IExn :: _ -> search_until r d
  | IPushtrap (i1, i0) :: c -> run_code (i0 @ c) v e s (FTry (i1 @ c, v, e, s) :: r) d
  | IPoptrap :: c -> let FTry _ :: r = r in run_code c v e s r d
  | IShift i :: c ->
      let c = VFun ([ICnt (c, e, s, r)], e) in run_code i v (c :: e) [] [] d
  | IReset i :: c -> run_code i v e [] [] ((s, FRet (c, e) :: r) :: d)
  | ICnt (c', e', s', r') :: c ->
      let v :: _ = e in                   (* acutual argument is saved on the top of e *)
      run_code c' v e' s' r' ((s, r) :: d)

let rec f7 t xs = match t with
  | Var x -> [IAccess (offset x xs)]
  | Abs (x, t) -> [ICur (g7 t (x :: xs))]
  | App (t0, t1) -> [IPushmark] @ f7 t1 xs @ [IPush] @ h7 t0 xs
  | Exn -> [IExn]
  | Try (t0, t1) -> [IPushtrap (f7 t1 xs, f7 t0 xs @ [IPoptrap])]
  | Sft (x, t) -> [IShift (f7 t (x :: xs))]
  | Rst t -> [IPushmark; IReset (f7 t xs)]              (* IPushmark is added *)
and g7 t xs = match t with
  | Abs (x, t) -> [IGrab] @ g7 t (x :: xs)
  | App (t0, t1) -> f7 t1 xs @ [IPush] @ i7 t0 xs
  | _ -> f7 t xs @ [IReturn]
and h7 t xs = match t with
  | App (t0, t1) -> f7 t1 xs @ [IPush] @ h7 t0 xs
  | Rst t -> [IReset (f7 t xs)]                         (* clause for reset is added *)
  | _ -> f7 t xs @ [IApply]
and i7 t xs = match t with
  | App (t0, t1) -> f7 t1 xs @ [IPush] @ i7 t0 xs
  | Rst t -> [IReset (f7 t xs)]                         (* clause for reset is added *)
  | _ -> f7 t xs @ [IAppterm]

let eval7 t = run_code (f7 t []) VMark [] [] [] []
```

Code 7. eval7: 最終的に得られた仮想機械とコンパイラ

$$
\begin{array}{rcl}
\mathcal{T}[\![x]\!]\rho & = & \mathcal{C}[\![x]\!]\rho \;=\; \mathsf{Access}(\mathsf{offset}\ \rho\ x) \\
\mathcal{T}[\![(\mathtt{reset}(M))\ N_1\ \cdots\ N_k]\!]\rho & = & \mathcal{C}[\![N_k]\!]\rho;\ \mathsf{Push};\ \cdots;\ \mathcal{C}[\![N_1]\!]\rho;\ \mathsf{Push};\ \mathsf{Reset}(\mathcal{C}[\![M]\!]\rho) \\
\mathcal{C}[\![(\mathtt{reset}(M))\ N_1\ \cdots\ N_k]\!]\rho & = & \mathsf{Pushmark};\ \mathcal{C}[\![N_k]\!]\rho;\ \mathsf{Push};\ \cdots;\ \mathcal{C}[\![N_1]\!]\rho;\ \mathsf{Push};\ \mathsf{Reset}(\mathcal{C}[\![M]\!]\rho) \\
\mathcal{T}[\![M\ N_1\ \cdots\ N_k]\!]\rho & = & \mathcal{C}[\![N_k]\!]\rho;\ \mathsf{Push};\ \cdots;\ \mathcal{C}[\![N_1]\!]\rho;\ \mathsf{Push};\ \mathcal{C}[\![M]\!]\rho;\ \mathsf{Appterm} \\
\mathcal{C}[\![M\ N_1\ \cdots\ N_k]\!]\rho & = & \mathsf{Pushmark};\ \mathcal{C}[\![N_k]\!]\rho;\ \mathsf{Push};\ \cdots;\ \mathcal{C}[\![N_1]\!]\rho;\ \mathsf{Push};\ \mathcal{C}[\![M]\!]\rho;\ \mathsf{Apply} \\
\mathcal{T}[\![\lambda x.M]\!]\rho & = & \mathsf{Grab};\ \mathcal{T}[\![M]\!](x :: \rho) \\
\mathcal{C}[\![\lambda x.M]\!]\rho & = & \mathsf{Cur}(\mathcal{T}[\![M]\!](x :: \rho);\ \mathsf{Return}) \\
\mathcal{T}[\![\mathtt{raise\_exn}]\!]\rho & = & \mathcal{C}[\![\mathtt{raise\_exn}]\!]\rho \;=\; \mathsf{Exn} \\
\mathcal{T}[\![\mathtt{try}(M_0, M_1)]\!]\rho & = & \mathcal{C}[\![\mathtt{try}(M_0, M_1)]\!]\rho \;=\; \mathsf{Pushtrap}(\mathcal{C}[\![M_1]\!]\rho, \mathcal{C}[\![M_0]\!]\rho;\ \mathsf{Poptrap}) \\
\mathcal{T}[\![\mathtt{shift}x.M]\!]\rho & = & \mathcal{C}[\![\mathtt{shift}x.M]\!]\rho \;=\; \mathsf{Shift}(\mathcal{C}[\![M]\!](x :: \rho)) \\
\mathcal{T}[\![\mathtt{reset}(M)]\!]\rho & = & \mathcal{C}[\![\mathtt{reset}(M)]\!]\rho \;=\; \mathsf{Pushmark};\ \mathsf{Reset}(\mathcal{C}[\![M]\!]\rho)
\end{array}
$$

$$
\begin{array}{rcl}
\langle \mathsf{Access}(n); c, a, v_0 \cdots v_n \cdots, s, r\rangle & \Rightarrow & \langle c, v_n, v_0 \cdots v_n \cdots, s, r\rangle \\
& \vdots & \\
\langle \mathsf{Exn}; c, a, e, s, (c_0, e_0, s_0).r\rangle & \Rightarrow & \langle c_0, a, e_0, s_0, r\rangle \\
\langle \mathsf{Exn}; c, a, e, s, (c_0, e_0).r\rangle \mid \langle \mathsf{Exn}; c, a, e, s, \gamma_r.r\rangle & \Rightarrow & \langle \mathsf{Exn}; c, a, e, s, r\rangle \\
\langle \mathsf{Exn}; c, a, e, s, r\rangle & \Rightarrow & \mathit{Error} \quad ((c_0, e_0, s_0)\ \text{が}\ r\ \text{に含まれない場合}) \\
\langle \mathsf{Pushtrap}(c_1, c_0); c, a, e, s, r\rangle & \Rightarrow & \langle c_0; c, a, e, s, (c_1; c, e, s).r\rangle \\
\langle \mathsf{Poptrap}; c, a, e, s, (c_0, e_0, s_0).r\rangle & \Rightarrow & \langle c, a, e, s, r\rangle \\
\langle \mathsf{Shift}(c_0); c, a, e, s_0.\gamma_a.s, r_0.\gamma_r.r\rangle & \Rightarrow & \langle c_0, a, (\mathsf{Cnt}(c, e, s_0, r_0), e).e, \gamma_a.s, \gamma_r.r\rangle \\
\langle \mathsf{Cnt}(c_0, e_0, s_0, r_0); c, a, v.e, s, r\rangle & \Rightarrow & \langle c_0, a, e_0, s_0.\gamma_a.s, r_0.\gamma_r.r\rangle \\
\langle \mathsf{Reset}(c_0); c, a, e, s, r\rangle & \Rightarrow & \langle c_0, a, e, \gamma_a.s, \gamma_r.(c, e).r\rangle
\end{array}
$$

Fig 2. 得られたコンパイル規則と仮想機械の定義

実装にダンプは存在しないが，図 2 のように引数スタックとリターンスタックに区切り γ_a，γ_r を導入すればスタックのまま表現可能である (Shift の場合の s_0 と r_0 にはそれぞれ γ_a，γ_r は含まれないものとする)。実際，実装上は両方のスタックに『印』に相当するポインタを導入し，`reset` のときにスタックを区切っている。Shift の場合の Cnt の構成はフレームの切り取り，Cnt の実行は切り取った継続を呼び出す際のフレームコピーや戻り番地の復活に対応する。

ZINC のトラップフレームには戻り番地と環境，引数スタックへのポインタが含まれ，図 2 の三つ組 (c, e, s) の各要素に対応する。`eval7` の `FTry` は `v` もデータとして保持しているが，これを受け取るコード `i1` は `f7` で作られたものであり，その先頭の命令は必ず `v` を捨てるので，実際には引数として受け取る必要はなく，ダミー値を渡して実行しても結果は変わらない (実装では，エラーハンドラに来ても必ず捕獲されるわけではなく，エラー値にマッチして挙動が変わるため，アキュムレータにはエラー値自体が保存されている)。実際のトラップフレームにはひとつ前のトラップフレームへのポインタも含まれるが，ここではフレームに直接マッチしているのでポインタは現れない。その代わり Exn の場合の二行目のケース (リターンフレーム以外の不要なフレームや印のスキップ) が必要になっている。

例外発生時には (リターンフレームや γ_r が飛ばされて) トラップフレームを探索すること，継続を切り取る際には γ_r までのリターンフレームを切り取り，その内部にはトラップフレームも含まれ得ることから，限定継続命令と例外が干渉し合うことは見て取れるだろう。実装上は，`shift` が呼び出されたときや切り取った継続が呼び出されるときのトラップフレームのポインタの付け替え，および例外発生時の `reset` のポインタの付け替えが必要となるが，これは OchaCaml の実装と合致している。

6 まとめと今後の課題

本論文では，例外処理と限定継続命令をサポートする評価器に各種変換を施し，仮想機械とコンパイラを導出した。これによりプログラム変換の適用範囲も広げられた。トラップフレームのような実装上の技術も自然な形で導入され，最終的に得られた仮想機械およびコンパイラは，実装との対応がとれている。この結果は推測を含む実装に (単純な例外のみのサポートではあるが) 正当性の証明を与えやすい形を導いたと言える。またフルの OchaCaml の実装の正当性の足がかりになった。`shift/reset` のための変換は 2CPS 変換と最後の `VFun` と `VCnt` の統合だけであり，例外と `shift/reset`，およびそれらの変換を除けば ZINC 自体の低レベルな正当性の証明にも繋がる。

ZINC では単純な実装の改良としてキャッシュを利用する仮想機械も提示されており (Ler90)，Caml Light (OchaCaml) での実装もそれを利用しているが，キャッシュも含めた正当性の証明は今後の課題である。また，一般の例外への拡張も行いたい。

謝辞 バグの指摘を含め，多くの有益なコメントを下さった査読者の皆様に深謝します。

参考文献

[ABDM03] Ager, M. S., D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the fifth International Symposium on Principles and Practice of Declarative Programming*, pages 8–19, August 2003.

[ADM04] Ager, M. S., O. Danvy, , and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Technical Report RS-04-28, BRICS, December 2004.

[AK07] Asai, K. and Y. Kameyama. Polymorphic delimited continuations. In *5th Asian Symposium on Programming Languages and Systems, Lecture Notes in Computer Science 4807*, pages 239–254, November 2007.

[AK10] Asai, K. and A. Kitani. Functional derivation of virtual machine for delimited continuations. In *Proceedings of the 12th International Symposium on Principles and Practice of Declarative Programming*, pages 87–97, July 2010.

[BD07] Biernacka, M. and O. Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1), December 2007.

[DF89] Danvy, O. and A. Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, July 1989.

[DF90] Danvy, O. and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160, June 1990.

[DF92] Danvy, O. and A. Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[Her07] Herman, D. Functional pearl: the great escape or, how to jump the border without getting caught. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 157–164, October 2007.

[KSS06] Kiselyov, O., C.-c. Shan, and A. Sabry. Delimited dynamic binding. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 26–37, 2006.

[Ler90] Leroy, X. The zinc experiment: An economical implementation of the ML language. Technical report, INRIA, February 1990.

[Ler97] Leroy, X. *The Caml Light system release 0.74*, December 1997.

[MA09] Masuko, M. and K. Asai. Direct implementation of shift and reset in the MinCaml compiler. In *Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 49–60, August 2009.

[MA11] Masuko, M. and K. Asai. Caml Light + shift/reset = Caml Shift. In *Theory and Practice of Delimited Continuations*, pages 33–46, May 2011. `http://pllab.is.ocha.ac.jp/~asai/OchaCaml/`.

[R^{6}RS] Sperber, M., R. K. Dybvig, M. Flatt, and A. van Straaten (Editors). Revised^6 report on the algorithmic language Scheme, `http://www.r6rs.org/`, 2007.