control/prompt の仮想機械導出

藤井 舞花, 浅井 健一

お茶の水女子大学
{g1720537,asai}@is.ocha.ac.jp

概 要 限定継続演算子 control/prompt を含む λ 計算の CPS インタプリタの定義は過去の研究で確立されている。本研究はこのインタプリタに対して CPS 変換や非関数化などのプログラム変換を施し、同等の動作をするコンパイラと仮想機械の導出を行なった。先行研究では shift/reset のインタプリタに対して仮想機械の導出がされている。control/prompt の実装のために新たに導入された trail があっても同様の変換方法で仮想機械を導出することができた。最終的には trail もスタックと同様のデータ構造になるが、途中で trail を合成する必要から単なる cons リストではなく append を伴う表現が必要であった。得られた仮想機械は control/prompt の実装におけるデータスタックの挙動やヒープへコピーするべき箇所を明確にモデル化している。

1 はじめに

継続とは、プログラムの実行中のある時点における残りの計算のことである。例えば `1 + 2 * 3` という計算があり「`2` と `3` をかける」の時点での残りの計算は「その結果に `1` を足す」である。この「`1` を足す」が継続である。

ここで、計算の範囲を指定しその範囲の継続を切り取ったり、複製をしたりすることができるのが限定継続演算子である。継続の取り出しやコピーはプログラムの制御フローを操作することに相当し、継続を処理するオペレータがあれば、大域ジャンプができるようになる。つまりユーザー自身が自由に定義できる例外処理のような挙動が可能となる。限定継続演算子には shift/reset [6, 7]、control/prompt [9]、shift0/reset0 [17]、control0/prompt0 [12] の 4 種類がある。また、限定継続演算子の他にも継続を明示的に扱えるものとして algebraic effect handlers [3, 13] が盛んに研究されている。

しかし、限定継続演算子を直接実装するプログラミング言語は少ない。CPS 変換による意味論の研究やその公理系に関する研究はなされているが、仮想機械を示している研究はほとんどない。これは、大域ジャンプの一種である例外処理が効率的に機械語でも実装されているのとは対照的である。

本研究では、すべての限定継続演算子に対する仮想機械を導出することを目指しているが、本論文ではその第一歩として control/prompt を取り上げる。本論文で使う手法は Danvy ら [1, 5] が提唱するもので、control/prompt の定義を与えるインタプリタに対して、各種の正当性の保証されたプログラム変換を施し、最終的にコンパイラと仮想機械の導出を行うというものである。

我々のグループでは、この手法を shift/reset をサポートするインタプリタに対して適用し、スタックを使った（抽象機械と）仮想機械を得ている [2]。本論文は、それを control/prompt に対して行うものである。ベースとなる control/prompt に対するインタプリタは、すでに Shan [21] によって与えられている。このインタプリタは、shift/reset に対するインタプリタとは異なり、継続の列を表す trail [4] というデータを受け渡す。この trail がインタプリタ内部で合成されるため、得られる仮想機械のスタックの構造が単なる cons で作られるリストではなく append 操作が必要なリストとなる。

最終的に得られる仮想機械は、control/prompt の実装において、いつ、どの部分のスタックをヒープへコピーするかを明確にモデル化したものになっており、スタックを直接操作する低レベルな実装の基礎になると期待される。

本論文の貢献は以下のようにまとめられる。

- control/prompt に対する仮想機械を示した。これは、スタックを直接、操作する仮想機械で、限定継続演算子が使われるときに、スタックのどの部分をコピーすべきかを明確に示したものとなっている。
- Danvy の手法が control/prompt に対するインタプリタに対しても有効であることを示した。

関連研究　限定継続演算子 shift/reset については、Scheme48 に対する直接実装 [11] と我々のグループが Caml Light に対して shift/reset を直接実装した OchaCaml [16] が存在する。しかし、これらはいずれも仮想機械は示しておらず、CPS インタプリタとの等価性についても議論していない。本研究は、正当性の保証されたプログラム変換を用いることで、CPS インタプリタと等価な仮想機械を示している。

正当性の証明されたプログラム変換をインタプリタに対して施す手法は Danvy ら [1, 5] が提案し多くのインタプリタに対して適用している。また、Igarashi と Iwaki [14] はこの手法を使ってマルチレベルの言語に対する仮想機械を得ている。我々のグループでは、この手法を shift/reset をサポートするインタプリタに対して適用し、対応する（抽象機械と）仮想機械を得ている [2]。筆者の知る限りでは、これ以外に限定継続演算子に対する仮想機械を示している研究は、存在しない。本研究は control/prompt に対してこの手法を適用するとともに、その差異を報告するものである。

仮想機械ではないが、抽象機械であれば Dyvbig ら [8] が限定継続演算子を対象とするものを示している。しかし、彼らの扱う限定継続演算子は我々の扱う 4 種類の限定継続演算子とは異なっている。そのため、CPS インタプリタも異なる形をしており、とくに trail は使われておらず、代わりに「継続の列」を表すメタ継続を使用している。

control/prompt を含む 4 種類の限定継続演算子に対するインタプリタは Shan [21] が定義を与えている。このインタプリタは trail を継続の関数合成の形で表現している。一方、trail を継続のリストとして表現するインタプリタも使われている [15, 4]。本論文は Shan による定義に基づいてプログラム変換を施すが、途中でそれをリストで表現するように変換しており、関数で表現した trail とリストで表現した trail をプログラム変換でつなぐものとなっている。

論文の構成　次節で control と prompt の簡単な紹介を行なった後、3 節でベースとなるインタプリタを紹介する。4 節で、このインタプリタに対して各種のプログラム変換を施し、仮想機械と同等のインタプリタを得る。最後に 5 節で、得られたインタプリタから仮想機械を導き、6 節でまとめる。

紙面の関係から全てのインタプリタのコードを載せることはできなかった。本論文だけで理解できるよう可能な範囲で説明を尽くしたが、詳しくは以下に置いた付随するコードを参照されたい。

```
http://pllab.is.ocha.ac.jp/~asai/jpapers/ppl/21/
```

2　control/prompt

control とは現在の継続を切り取り、prompt は control で切り取られる継続の範囲を限定する演算子である。本研究では `(control k -> ...)` と書くと変数 `k` に現在の継続が渡される。`...` 内で `k` を用いて継続を呼び出したり `k` を使わず継続を捨てることができる。また、`prompt (...)` と

書くと括弧内にある control で切り取られる継続を括弧内の処理のみに限定する。ここで prompt がいくつもある場合 control は直近の prompt までの継続を切り取る。

例えば、`1 + prompt (2 * (control k -> k (k 3)))` という計算を考えよう。control オペレータは prompt 内の継続である `2 * [.]` を切り取り `k` に渡す。そしてこの `k` を用いることで `2 * [.]` を呼び出すことができる。例のように `k` を何回も呼び出しすることが可能であり、計算の最終結果は `13` と評価される。反対に `k` を使わず継続（ここでは `2 * [.]`）を捨てることもできる。

shift/reset と control/prompt の違いは、切り取った継続の周りに reset (prompt) が存在するかしないかである。両者の違いが出る例として以下の例がある。

```
1 + reset ((shift k -> 2 * k 3) + (shift h -> 4))
1 + prompt ((control k -> 2 * k 3) + (control h -> 4))
```

まず `k` に渡される継続が、shift の式では `k = reset ([.] + (shift h -> 4))`、control の式では `k = [.] + (control h -> 4)` となり、`control` に切り取られた継続は `prompt` で囲まれない。この `k` が呼び出された際、計算は以下のように評価される。

```
1 + reset (2 * reset (3 + (shift h -> 4)))
1 + prompt (2 * (3 + (control h -> 4)))
```

そして次の `h` に渡される継続が、shift の式では `h = reset (3 + [.])` であるのに対し、control の式では `h = 2 * (3 + [.])` となる。`control` は `k` を呼び出したときの継続（invocation context）`2 * [.]` まで切り取られるようになる。結果として前者は `9`、後者は `5` と評価される。

invocation context を追跡するために、control/prompt の実装では trail が導入されている。この trail というデータがどのようにプログラム変換されるかが、これまでの shift/reset に対するプログラム変換との違いになる。

3 CPS による λ 計算のインタプリタ

本研究では、型なし λ 計算を control/prompt で拡張した構文を持つ言語を対象とする。これを関数型言語 OCaml を用いて実装していく。ベースとするインタプリタは Shan [21] によるインタプリタである。これは、control/prompt の実装に必要な trail をリストなどのデータ構造に具体化せず、関数として外延的に表現したインタプリタである。項と値の定義は以下のようになる。

```
type e = Var of string                  (* 変数 *)
       | Fun of string * e              (* λ抽象 *)
       | App of e * e                   (* 関数適用 *)
       | Control of string * e          (* control *)
       | Prompt of e                    (* prompt *)

type v = VFun of (v -> c -> t -> v)     (* 関数値 *)
       | VCont of c * t                 (* 継続値 *)
and  c = v -> t -> v                    (* 継続の型 *)
and  t = TNil | Trail of c              (* trail *)
```

trail は `t` という型で表現している。`TNil` は空の trail、`Trail (c)` は invocation contexts（継続）を全て関数合成したものが入った trail を表す。

この言語に対する評価器 `f1` は図 1 のように実装される。限定継続演算子をサポートするため、継続渡し形式 (CPS) で書かれている。加えて、control/prompt をサポートするため、さらに引数

```
(* idc : v -> t -> v *)
let idc v t = match t with
    TNil -> v
  | Trail (c) -> c v TNil

(* cons : c -> t -> t *)
let rec cons c t = match t with
    TNil -> Trail (c)
  | Trail (c') -> Trail (fun v t' -> c v (cons c' t'))

(* apnd : t -> t -> t *)
let apnd t0 t1 = match t0 with
    TNil -> t1
  | Trail (c) -> cons c t1

(* f1 : e -> string list -> v list -> c -> t -> v *)
let rec f1 e xs vs c t = match e with
    Var (x) -> c (List.nth vs (Env.offset x xs)) t
  | Fun (x, e) -> c (VFun (fun v c' t' -> f1 e (x :: xs) (v :: vs) c' t')) t
  | App (e0, e1) ->
      f1 e0 xs vs (fun v0 t0 ->
      f1 e1 xs vs (fun v1 t1 ->
      (match v0 with
           VFun (f) -> f v1 c t1
         | VCont (c', t') -> c' v1 (apnd t' (cons c t1)))) t0) t
  | Control (x, e) -> f1 e (x :: xs) (VCont (c, t) :: vs) idc TNil
  | Prompt (e) -> c (f1 e xs vs idc TNil) t
```

図 1. control/prompt のための definitional interpreter

として trail t を受け取っている。しかし、trail が実際に使われるのは control と prompt の場合だけであり、それ以外の構文では単に trail は変更することなく渡されているだけである。（継続値を呼び出す部分については後述。）関数適用の際の引数の評価は左右どちらが先でも良いが、本論文は左から先に評価するものとする。

環境は、変数名のリスト xs と値のリスト vs の二つを別々に持つようにしている。xs の型は string list、vs の型は v list である。本研究の目標はインタプリタからコンパイラと仮想機械を抽出することであるため、後の変換で項のみに依存する部分（static データ）と実行時に処理する部分（dynamic データ）とを分けたい。そのため、ここでは環境を二つに分割して実装している。環境の参照に用いられる関数 Env.offset は、参照したい変数に対応する値が値のリストの何番目に格納されているかを調べる関数である。

f1 の control と prompt のケースを見てみよう。Prompt (e) を評価するには、body 部分 e を空の継続 idc と空の trail TNil のもとで評価し、その結果を直接形式で現在の継続 c に（現在の trail である t とともに）渡す。継続と trail を初期化することで e 内の control によって切り取られる継続の範囲を限定している。ここで、idc は空の継続で、図 1 に定義されている。idc は、受け取った trail が空であれば引数をそのまま返す恒等関数だが、後に述べるように捕捉された継続を呼び出すと trail に後に実行すべき継続が保持される。そのような場合は、trail に入っている継続を呼び出して実行する。

Control (x, e) を評価するにも、prompt の場合と同様に body 部分 e を空の継続 idc と空の trail TNil のもとで評価する。その際、そのときの継続と trail が VCont として x に束縛される。これは control x -> ... という計算において現在の継続を変数 x に渡す挙動に相当する。つま

```
type v = VFun of e * string * string list * v list
       | VCont of c * t
and  c = C0
       | CApp0 of e * string list * v list * c
       | CApp1 of v * c
       | CAppend of c * c
and  t = TNil | Trail of c

(* cons : c -> t -> t *)
let rec cons c t = match t with
    TNil -> Trail (c)
  | Trail (c') -> Trail ( CAppend (c, c') )
```

図 2. 非関数化した後の値の型と cons

り、e の中では x を呼び出すことで、捕捉された継続（と trail）を実行することができる。ここで、Shan [21] の定義では高階関数によって継続の切り取りを表しているが、本研究ではこの高階関数を予め非関数化 [19, 20] した状態から始めている。これは先行研究に合わせた形にしたためであり、どちらの場合から始めても影響はない。

捕捉された継続が実行されるのは App の関数部分が VCont だった場合である。このときは VCont に保存されていた継続を呼び出すが、その際、保存されていた trail に「そのときの継続と trail」を合成する。ここが control/prompt と shift/reset が異なる部分である。shift/reset では、捕捉された継続が呼び出される際は「そのときの継続と trail」にはアクセスできない形（具体的には c (c' v1 t') t1）となる。しかし、control/prompt では trail t' に「そのときの継続 c と trail t1」が合成されるため、c' の実行中に c と t1 にアクセスできることになる。trail の合成は図 1 では、cons と apnd を使用している。前者は trail に現在の継続を追加する関数、後者は二つの trail をくっつける関数である。trail をリストで表現した場合にそれぞれリストの cons と append に相当している。

Shan [21] のインタプリタでは apnd のような表現はなく、(cons c' t') v1 (cons c t1) という形で合成している。本研究では apnd を定義して c' v1 (apnd t' (cons c t1)) としているが、両者は t' における場合分けをすると同等であることが導ける。Biernacki ら [4] や Kameyama と Yonezawa [15] は後者のように append を用いて実装している。

4 プログラム変換

前節のインタプリタに対してプログラム変換を行い、control/prompt を含む仮想機械を導出をする。4.1 節から 4.5 節までの目標は、インタプリタからスタックのデータ構造を抽出することである。変換における変更を示すため、インタプリタで使用している型を載せており、背景色でハイライトされている部分が変更された箇所となっている。また、4.6 節から 4.11 節までで、それをコンパイラと仮想機械に分離する。導出の全体の流れは shift/reset に対するプログラム変換 [2] とほぼ同様である。インタプリタの詳細は 1 節最後に示した URL のコードを参照されたい。

4.1 非関数化

スタックを導入するため、まず継続など関数値を 1 階のデータに変換する非関数化 [19, 20] を行う。非関数化とはプログラム中に出現する関数値（OCaml の fun 式、λ式）に対して関数の body

```
type v = VFun of e * string * string list * v list
       | VCont of c * t
and  f = CApp0 of e * string list * v list
       | CApp1 of v
and  c = CNil | CCons of f * c | CAppend of c * c
and  t = TNil | Trail of c
```

図 3. 継続をリスト化した後の値の型

```
type v = VFun of e * string * string list * v list
       | VCont of c * s * t
       | VEnv of v list
and  f = CApp0 of e * string list
       | CApp1
and  c = CNil | CCons of f * c | CAppend of c * c
and  s = SNil | SCons of v * s | SAppend of s * s
and  t = TNil | Trail of c * s
```

図 4. スタック導入後の値の型

部分で自由変数となっているものを保持するコンストラクタを定義し、関数値の代わりにそのコンストラクタを使用することである。

非関数化すると、値の型は図 2 のようになる。これまで関数値として表現されていた VFun と c の型が 1 階のデータになっている。C0 は idc に対応し、CApp0 と CApp1 は App のケースの二つの継続に対応する。CAppend は control/prompt に特有のデータで、これは図 1 の cons で作られる Trail の引数の継続に対応する。cons では、継続 c を Trail (c') と合成しているが、この二つの自由変数が CAppend の引数となっている。変換後の cons は図 2 に示されている。

4.2 継続をリスト化

図 2 の c に着目すると、これは C0 を空リスト、CApp0 と CApp1 が cons、CAppend が append であるようなリスト構造とみなすことができる。そのように変換したのが図 3 である。

図 2 で CApp0 と CApp1 に保持されていた継続 c 以外のデータを格納するために、f を用意した。継続 c はフレーム f を保持するリストのような構造になった。この変換は、図 2 で CApp0 of f * c、CApp1 of f * c としていたのを CCons of f * c とまとめただけの変換であるため、前節のインタプリタと同じ挙動を示す。shift/reset の場合は、継続 c の型は空リストと cons からなるリストとなったが、ここではさらに append が必要になっている。

4.3 スタック導入

図 3 のフレームの型を見るとコンストラクタ CApp0 と CApp1 には変換時のデータ（static データ）と実行時のデータ（dynamic データ）の両方が含まれている。項 e と変数名のリスト string list は変換時のデータ、値 v と値のリスト v list は実行時のデータであり、実行時のみ処理が可能である。本研究の目的はインタプリタをコンパイラと仮想機械に変換することなので、この二種類のデータを分ける変換を行う。

そこで実行時のデータである値を格納するスタックを導入する。図 4 には変換した結果の型が示されている。図 3 では c は f のリストだったが、図 4 では、f には変換時のデータのみを残し、実行時のデータは新たに導入された s の中に置く。s は c と同相の形をしているが、その SCons の

```
type v = VFun of e * string * string list * v list
       | VCont of c * s * t
       | VEnv of v list
and  c = C0
       | CApp0 of e * string list * c
       | CApp1 of c
       | CAppend of c * c
and  s = SNil | SCons of v * s | SAppend of s * s
and  t = TNil | Trail of c * s
```

図 5. 継続を非リスト化した後の値の型

第 1 引数のところだけが v となっている。ここに f の中の実行時のデータを置く。実行時のデータには v list が含まれるので、これをスタックに積めるようにするため VEnv が v に追加されている。スタックに保持する値は継続に入っているフレームに対応する。そのため、継続とスタックは常に同じ形のデータ構造となっている。

この変換は局所的にデータ構造を変更しただけであるため、変換前後で同等の振る舞いをする。

4.4 継続を非リスト化

4.1 節の非関数化と 4.2 節の継続のリスト化は、スタックを導入するための変換である。4.3 節でスタックが導入できたので元の状態に戻す変換をする。まず継続を非リスト化した形式に戻す。

図 5 に継続を非リスト化した結果の型が示されている。ここで、非線形化するのは static な f のみでスタック s は非リスト化していない。f の情報のみを c に戻している。

この変換も、4.2 節と逆の局所的なデータ構造の変更を行なっているだけであるため、本節のインタプリタもこれまでと同等の挙動を示す。

4.5 関数化

次に関数化された形に戻す。非関数化されていたコンストラクタを関数値に変換する。

図 6 に関数化した結果のインタプリタが示されている。プログラム変換をする前の 3 節のインタプリタと比較すると、評価関数にスタック s の引数が追加され、継続や VFun の中身の関数にもスタック s を渡すように引数が変更されている。また、切り取られた継続を表現する VCont にスタックも保持するようになり、trail にも継続だけでなくスタックも保持するようになった。

これまでの構成から、このインタプリタでは常に「継続 c を非関数化してリスト化したもの」とスタック s は同じ構造になっている。特に、idc は初期継続なので対応するスタックは必ず SNil となる。従って、idc の第 2 引数は SNil 以外にはなり得ない。また、cons の Trail のケースでは、必ず二つの継続を合成したものになっているので、Trail 第 1 引数の関数の第 2 引数は必ず SAppend の形になる。

また f6 は値の環境の保存と復元もモデル化している。関数適用では、関数部分 e0 が評価される際に環境 vs はデータスタックに保存される。これは、最初のインタプリタでは継続の自由変数に vs が入っていたことに相当する。そして e1 の評価時に環境 vs が必要なため e0 の評価後のスタックから復元されている。

関数化の正当性は非関数化の正当性から得られるため、変換前後で同等の値を返すことは保証される。

```
type v = VFun of (v -> s -> c -> t -> v)
       | VCont of c * s * t
       | VEnv of v list
and  c = v -> s -> t -> v
and  s = SNil | SCons of v * s | SAppend of s * s
and  t = TNil | Trail of c * s

(* idc : v -> s -> t -> v *)
let idc v SNil t = match t with
    TNil -> v
  | Trail(c,s) -> c v s TNil

(* cons : c -> s -> t -> t *)
let rec cons c s t = match t with
    TNil -> Trail(c,s)
  | Trail(c',s') ->
    Trail((fun v (SAppend(s,s')) t -> c v s (cons c' s' t)),SAppend(s,s'))

(* apnd : t -> t -> t *)
let apnd t0 t1 = match t0 with
    TNil -> t1
  | Trail(c,s) -> cons c s t1

(* f6 : e -> string list -> v list -> s -> c -> t -> v *)
let rec f6 e xs vs s c t = match e with
    Var(x) -> c (List.nth vs (Env.offset x xs)) s t
  | Fun(x,e) -> c (VFun(fun v s' c' t' -> f6 e (x::xs) (v::vs) s' c' t')) s t
  | App(e0,e1) ->
      f6 e0 xs vs (SCons(VEnv(vs),s)) (fun v0 (SCons(VEnv(vs),s)) t0 ->
      f6 e1 xs vs (SCons(v0,s)) (fun v1 (SCons(v0,s)) t1 ->
      (match v0 with
           VFun(f) -> f v1 s c t1
         | VCont(c',s',t') -> c' v1 s' (apnd t' (cons c s t1)))) t0) t
  | Control(x,e) -> f6 e (x::xs) (VCont(c,s,t)::vs) SNil idc TNil
  | Prompt(e) -> c (f6 e xs vs SNil idc TNil) s t
```

図 6. 関数化した後のインタプリタ

4.6 値とスタックの結合

図 6 の VFun と c それぞれの引数である値 v とスタック s、そしてインタプリタ f6 の引数である値の環境 vs とスタック s を結合する変換を行う。仮想機械では値の受け渡しはスタック経由で行いたいので、これら二つをまとめて、値はスタック上で受け渡すようにする。

変換後は図 7 のようになる。VFun や継続 c には値は渡されず、代わりにその値が積まれたスタックを渡すようになった。そして、f6 では、値の環境 vs とスタック s を渡していたのに対して、インタプリタ f7 では、この二つを SCons(VEnv(vs),s) としてまとめて渡すようにする。

図 6 と図 7 のインタプリタを比べると分かるが、この変換は単に、値とスタックが同時に出てくる場所では値はスタックの中で受け渡すようにしているだけである。そのため変換前後でのインタプリタは同等の挙動をする。

4.7 命令に分解

インタプリタを、「static データのみを受け取って処理する部分」と「dynamic データを受け取ったら処理する部分」に分割する。前者がコンパイラ、後者が仮想機械となる。

```
type v = VFun of ( s -> c -> t -> v ) | VCont of c * s * t
       | VEnv of v list
and  c = s -> t -> v
and  s = SNil | SCons of v * s | SAppend of s * s
and  t = TNil | Trail of c * s

(* idc : s -> t -> v *)
let idc (SCons(v,SNil)) t = match t with
    TNil -> v
  | Trail(c,s) -> c (SCons(v,s)) TNil

(* cons : c -> s -> t -> t *)
let rec cons c s t = match t with
    TNil -> Trail(c,s)
  | Trail(c',s') ->
    Trail((fun (SCons(v,SAppend(s,s'))) t -> c (SCons(v,s)) (cons c' s' t)),
          SAppend(s,s'))

(* apnd : t -> t -> t *)
let apnd t0 t1 = match t0 with
    TNil -> t1
  | Trail(c,s) -> cons c s t1

(* f7 : e -> string list -> s -> c -> t -> v *)
let rec f7 e xs (SCons(VEnv(vs),s)) c t = match e with
    Var(x) -> c (SCons(List.nth vs (Env.offset x xs),s)) t
  | Fun(x,e) -> c (Sons(VFun(fun (SCons(v,s')) c' t' ->
                                f7 e (x::xs) (SCons(VEnv(v::vs),s')) c' t'),s)) t
  | App(e0,e1) ->
      f7 e0 xs (SCons(VEnv(vs),SCons(VEnv(vs),s)))
               (fun (SCons(v0,SCons(VEnv(vs),s))) t0 ->
      f7 e1 xs (SCons(VEnv(vs),SCons(v0,s))) (fun (SCons(v1,SCons(v0,s))) t1 ->
      (match v0 with
           VFun(f) -> f (SCons(v1,s)) c t1
         | VCont(c',s',t') -> c' (SCons(v1,s')) (apnd t' (cons c s t1)))) t0) t
  | Control(x,e) -> f7 e (x::xs) (SCons(VEnv(VCont(c,s,t)::vs),SNil)) idc TNil
  | Prompt(e) -> c (SCons(f7 e xs (SCons(VEnv(vs),SNil)) idc TNil,s)) t
```

図 7. 値とスタックの結合をした後のインタプリタ

この分解は、上に示したインタプリタ f7 の型を static データである最初の二つの引数と、dynamic データであるその後の引数に分けることで行う。ここで dynamic データを受け取ったら行う仕事を表す型「命令 i」を導入する。

```
type i = s -> c -> t -> v
```

図 8 に変換した結果を示す。今回はインタプリタを分解する変換なので初期継続 idc や関数 cons, apnd には変化がないためここでは示していない。評価器 f8 は項 e と変数名の環境 xs の static データのみを処理するようにする。ここで変数名の環境も受け取るのは、変数名は項の情報のみに依存しているため static データとなるからである。評価器が返すものの型は i となる。インタプリタをコンパイラと仮想機械に分割する手順は、shift/reset の仮想機械を抽出したときとほぼ同様であるため、複雑な命令生成する部分や control/prompt 特有の動作をする部分について説明する。

まず、App のケースにおいて、図 7 では e0 の再帰の前で環境の保存、e1 の再帰の前で環境の復元が行われている。これらスタックの動作をそれぞれ命令 push_env と pop_env として定義する。

```
type v = VFun of (s -> c -> t -> v)
       | VCont of c * s * t
       | VEnv of v list
       | VK of c
and  c = s -> t -> v
and  s = SNil | SCons of v * s | SAppend of s * s
and  t = TNil | Trail of c * s
type i = s -> c -> t -> v

(* (>>) : i -> i -> i *)
let (>>) i0 i1 = fun s c t -> i0 s (fun s' t' -> i1 s' c t') t

(* access : int -> i *)
let access n = fun (SCons(VEnv(vs),s)) c t -> c (SCons(List.nth vs n,s)) t

(* push_closure : i -> i *)
let push_closure i = fun (SCons(VEnv(vs),s)) c t ->
  c (SCons(VFun(fun (SCons(v,s')) c' t' -> i (SCons(VEnv(v::vs),s')) c' t'),s)) t

(* return : i *)
let return = fun (SCons(v,SCons(VK(c),s))) _ t -> c (SCons(v,s)) t

(* push_env : i *)
let push_env = fun (SCons(VEnv(vs),s)) c t -> c (SCons(VEnv(vs),SCons(VEnv(vs),s))) t

(* pop_env : i *)
let pop_env = fun (SCons(v,SCons(VEnv(vs),s))) c t -> c (SCons(VEnv(vs),SCons(v,s))) t

(* call : i *)
let call = fun (SCons(v1,SCons(v0,s))) c t -> match v0 with
    VFun(f) -> f (SCons(v1,SCons(VK(c),s))) idc t
  | VCont(c',s',t') -> c' (SCons(v1,s')) (apnd t' (cons c s t))

(* control : i -> i *)
let control i = fun (SCons(VEnv(vs),s)) c t ->
  i (SCons(VEnv(VCont(c,s,t)::vs),SNil)) idc TNil

(* prompt : i -> i *)
let prompt i = fun (SCons(VEnv(vs),s)) c t ->
  c (SCons(i (SCons(VEnv(vs),SNil)) idc TNil,s)) t

(* f8 : e -> string list -> i *)
let rec f8 e xs = match e with
    Var(x) -> access (Env.offset x xs)
  | Fun(x,e) -> push_closure (f8 e (x::xs) >> return)
  | App(e0,e1) -> push_env >> f8 e0 xs >> pop_env >> f8 e1 xs >> call
  | Control(x,e) -> control (f8 e (x::xs))
  | Prompt(e) -> prompt (f8 e xs)
```

図 8. 命令に分解した後のインタプリタ

```
type v = VFun of (s -> c -> t -> m -> v) | VCont of c * s * t
       | VEnv of v list | VK of c
and  c = s -> t -> m -> v
and  s = SNil | SCons of v * s | SAppend of s * s
and  t = TNil | Trail of c * s
and m = v -> v
type i = s -> c -> t -> m -> v
```

図 9. CPS 変換後の値と命令の型

さらに、再帰後に関数適用する部分を命令 `call` とする。ここで命令を合成する関数 `(>>)` を定義することで「環境を push して `e0` を実行し、環境を pop して復元してから `e1` を実行し、最後に関数呼び出しをする」という命令群のように理解できる。これは、shift/reset のときと同様である。

`Control` と `Prompt` ではそれぞれその後の仕事を表す命令を `control`, `prompt` とし、引数として body 部分の命令を受け取る。すると、命令 `control` は空の継続、trail のもとで body 部分を実行し、その際に捕捉された継続をスタックに積んでいる。命令 `prompt` も body 部分を空の継続、trail のもとで実行し、その結果を直接形式でそのときの継続 `c` に渡す。

命令 `call` は、スタックから関数部分 `v0` と引数部分 `v1` を取り出し、関数呼び出しを行う。`VFun` のケースに出てくる `VK` は返り番地を保存するためのもので、図 8 で値 `v` に加えられたものである。これも shift/reset のときと同様である。一方、`VCont` の場合は、control/prompt 用に「そのときの継続と trail」を現在の trail と結合している。

ここで示した命令群は、この段階ではまだ普通の関数である。従って、これらの命令を単に関数展開すれば 4.6 節のインタプリタに戻ることになる。

4.8 CPS 変換

図 8 はほとんど仮想機械の命令列とみなすことができるが、まだ未完成の部分がある。それは、図 8 から抜粋した次の `prompt` である。

```
let prompt i = fun (SCons(VEnv(vs),s)) c t ->
  c (SCons(i (SCons(VEnv(vs),SNil)) idc TNil,s)) t
```

命令 `prompt` では `i` の命令列で処理した値を現在のスタック `s` に保存している。さらにそのスタックを現在の継続 `c` に渡し処理を行っている。これは末尾呼び出しになっていないため状態遷移図として仮想機械を表現するには不十分である。

この関数の入れ子を末尾再帰の形にするため CPS 変換を行う。ここで導入される継続はメタ継続とする。CPS 変換された結果の型は図 9 に示される。メタ継続の型 `m` が新たに定義されている。`VFun` や 継続 `c` 、命令 `i` にはメタ継続も渡されるようになった。

図 8 の命令はもともとほとんどが末尾再帰になっているため、渡されたメタ継続をそのまま引き継ぐだけで処理部分は特に変更はない。変更された命令 `prompt` は以下のようになる。

```
let prompt i = fun (SCons(VEnv(vs),s)) c t m ->
  i (SCons(VEnv(vs),SNil)) idc TNil (fun v -> c (SCons(v,s)) t m)
```

関数の入れ子が解消され、仮想機械に近い形となっている。CPS 変換は正当性が保証されている。そしてこの先の変換で実際に（現段階では関数である）命令をデータにして仮想機械を導いていく。

```
type v = VFun of i * v list | VCont of c * s * t
       | VEnv of v list | VK of c
and i = IAccess of int | IPush_closure of i | IReturn
       | IPush_env | IPop_env | ICall
       | IControl of i | IPrompt of i
       | ISeq of i * i
and  c = CNil | CCons of i * c | CAppend of c * c
and  s = SNil | SCons of v * s | SAppend of s * s
and  t = TNil | Trail of c * s
and  m = (c * s * t) list

(* cons : c -> s -> t -> t *)
let rec cons c s t = match t with
    TNil -> Trail(c,s)
  | Trail(c',s') -> Trail(CAppend(c,c'),SAppend(s,s'))
(* apnd : t -> t -> t *)
let apnd t0 t1 = match t0 with
    TNil -> t1
  | Trail(c,s) -> cons c s t1

(* run_c10 : c -> s -> t -> m -> v *)
let rec run_c10 c s t m = match (c,s) with
    (CNil,SCons(v,SNil)) -> (match t with
        TNil -> (match m with
            [] -> v
          | (c,s,t)::m -> run_c10 c (SCons(v,s)) t m)
      | Trail(c,s) -> run_c10 c (SCons(v,s)) TNil m))
  | (CCons(i,c),s) -> run_i10 i s c t m
  | (CAppend(c,c'),SCons(v,SAppend(s,s'))) -> run_c10 c (SCons(v,s)) (cons c' s' t) m)
(* run_i10 : i -> s -> c -> t -> m -> v *)
and run_i10 i s c t m = match (i,s) with
    (IAccess(n),SCons(VEnv(vs),s)) -> run_c10 c (SCons(List.nth vs n,s)) t m
  | (IPush_closure(i),SCons(VEnv(vs),s)) -> run_c10 c (SCons(VFun(i,vs),s)) t m
  | (IReturn,SCons(v,SCons(VK(c),s))) -> run_c10 c (SCons(v,s)) t m
  | (IPush_env,SCons(VEnv(vs),s)) -> run_c10 c (SCons(VEnv(vs),SCons(VEnv(vs),s))) t m
  | (IPop_env,SCons(v0,SCons(VEnv(vs),s))) -> run_c10 c (SCons(VEnv(vs),SCons(v0,s))) t m
  | (ICall,SCons(v1,SCons(v0,s))) -> (match v0 with
        VFun(i,vs) -> run_i10 i (SCons(VEnv(v1::vs),SCons(VK(c),s))) CNil t m
      | VCont(c',s',t') -> run_c10 c' (SCons(v1,s')) (apnd t' (cons c s t)) m)
  | (IControl(i),SCons(VEnv(vs),s)) ->
      run_i10 i (SCons(VEnv(VCont(c,s,t)::vs),SNil)) CNil TNil m
  | (IPrompt(i),SCons(VEnv(vs),s)) ->
      run_i10 i (SCons(VEnv(vs),SNil)) CNil TNil ((c,s,t)::m)
  | (ISeq(i0,i1),s) -> run_i10 i0 s (CCons(i1,c)) t m

(* (>>) : i -> i -> i *)
let (>>) i0 i1 = ISeq(i0,i1)

(* f10 : e -> string list -> i *)
let rec f10 e xs = match e with
    Var(x) -> IAccess(Env.offset x xs)
  | Fun(x,e) -> IPush_closure(f10 e (x::xs) >> IReturn)
  | App(e0,e1) -> IPush_env >> f10 e0 xs >> IPop_env >> f10 e1 xs >> ICall
  | Control(x,e) -> IControl(f10 e (x::xs))
  | Prompt(e) -> IPrompt(f10 e xs)
```

図 10. 非関数化後の値の型

4.9 非関数化

前節までで導入した命令をより機械語に近づけたものにするため、関数値になっているものを非関数化する。具体的には VFun の引数に加えて、継続、メタ継続、そして命令を非関数化する。以上の変換を施した結果の型は図 10 に示される。

継続を非関数化すると、これまで通り CNil, CCons, CAppend の 3 つの構成子となる。

一方、メタ継続が出てくるのは（初期メタ継続以外には）prompt に出てくる fun v -> c (SCons (v, s)) t m である。これの自由変数は継続 c, スタック s, trail t, そしてメタ継続 m である。これを非関数化すると、これらを引数に持つデータとなる。さらに、このデータは c, s, t を要素にもつリストと同相になる。継続の場合と違い append は必要ないので、ここでは OCaml のリストをそのまま用いて (c * s * t) list で表現した。ここで空リストは初期メタ継続に対応する。

なお、図 10 ではメタ継続の型 m を定義はしているが、図 10 の他の場所では使われていない。m はインタプリタ関数 f10 等で使われている。m は単に (c * s * t) list と同じことなので定義しなくても構わない。

最後に、命令 i を非関数化すると、各関数の中で作られる関数値に従って対応する構成子が作られる。これが、仮想機械の命令となる。それに加えて、二つの命令を合成して新たな命令を作る (>>) を非関数化すると、二つの命令を引数に持つようなデータが作成される。これには ISeq という名前をつけている。この ISeq は二つの命令列を持つ構成子であるため、命令が木構造になることになる。

4.10 命令列を cons リスト化

前節までは命令列を結合するのに ISeq を使っていた。しかし、図 10 の run_i10 で ISeq (i0, i1) が呼び出された際、i0 を実行し i1 は継続に cons されている。そこでこの作業を無くすため、はじめのコンパイラから命令を順に cons リストの構造となるようにする。

この変換を行ったコンパイラと仮想機械は図 11 に示す。コンパイラから生成される命令は全て cons リストの構造となるよう関数 at を定義している。この変換にすることでコンパイラから出力される命令は全て cons リストの構造になるようにした。結果、命令 i から ISeq はなくなることになる。

この変換の正当性を示す。まず、型 i を ISeq がない i の cons リストの構造に変換する関数 flat を定義する（図 12）さらに、その他のデータ型に現れる命令を cons リスト化する関数を同様に定義する（flatC, flatV, flatS, flatT, flatM）。そして、次の同等性を証明する。

- flat (f10 e xs) = f11 e xs
- flatV (run_i10 i s c t m) =
  run_c11 (at (flat i) (flatC c)) (flatS s) (flatT t) (flatM m)

前者は項 e の帰納法によって証明される。f10 のコンパイラから生成された命令を cons リスト化すると、f11 のコンパイラから得られる命令列と同じであることを示している。後者は、図 10 の仮想機械が進むステップ数に関する帰納法によって証明される。

ここで ISeq の場合の証明を図 12 に示す。ISeq(i0,i1) が run_i10 に渡されると、i0 の命令を実行し i1 は継続に cons されている。この実行を帰納法の仮定によって run_c11 に適用する。そして、関数 at は結合律を満たすことが証明できるため引数を入れ換えることができ、flat 関数の定義より仮想機械の同等性が成立している。また、ISeq に入っていた命令の順序が保たれていることも確認できる。ISeq は cons リスト化されるため、図 11 の新しい仮想機械に対応する実行ステップは存在しないが、全ての命令は有限であるので ISeq を永遠に実行することはない。よって（停止性も含めて）前節と本節の仮想機械は同等の挙動をする。

```
type v = VFun of c * v list | VCont of c * s * t
       | VEnv of v list | VK of c
and  i = IAccess of int | IPush_closure of c | IReturn
       | IPush_env | IPop_env | ICall
       | IControl of c | IPrompt of c
and  c = CNil | CCons of i * c | CAppend of c * c
and  s = SNil | SCons of v * s | SAppend of s * s
and  t = TNil | Trail of c * s
and  m = (c * s * t) list

(* run_c11 : c -> s -> t -> m -> v *)
let rec run_c11 c s t m = match (c,s) with
    (CNil,SCons(v,SNil)) -> (match t with
        TNil -> (match m with
            [] -> v
          | (c,s,t) :: m -> run_c11 c (SCons(v,s)) t m)
      | Trail (c,s) -> run_c11 c (SCons(v,s)) TNil m)
  | (CCons(IAccess(n),c),SCons(VEnv(vs),s)) -> run_c11 c (SCons(List.nth vs n,s)) t m
  | (CCons(IPush_closure(c'),c),SCons(VEnv(vs),s)) ->
      run_c11 c (SCons(VFun(c',vs),s)) t m
  | (CCons(IReturn,_),SCons(v,SCons(VK(c),s))) -> run_c11 c (SCons(v,s)) t m
  | (CCons(IPush_env,c),SCons(VEnv(vs),s)) ->
      run_c11 c (SCons(VEnv(vs),SCons(VEnv(vs),s))) t m
  | (CCons(IPop_env,c),SCons(v,SCons(VEnv(vs),s))) ->
      run_c11 c (SCons(VEnv(vs),SCons(v,s))) t m
  | (CCons(ICall,c),SCons(v1,SCons(v0,s))) -> (match v0 with
        VFun(c',vs) -> run_c11 c' (SCons(VEnv(v1::vs),SCons(VK(c),s))) t m
      | VCont(c',s',t') -> run_c11 c' (SCons(v1,s')) (apnd t' (cons c s t)) m)
  | (CCons(IControl(c'),c),SCons(VEnv(vs),s)) ->
      run_c11 c' (SCons(VEnv(VCont(c,s,t)::vs),SNil)) TNil m
  | (CCons(IPrompt(c'),c),SCons(VEnv(vs),s)) ->
      run_c11 c' (SCons(VEnv(vs),SNil)) TNil ((c,s,t)::m)
  | (CAppend(c,c'),SCons(v,SAppend(s,s'))) -> run_c11 c (SCons(v,s)) (cons c' s' t) m

(* at : c -> c -> c *)
let rec at c0 c1 = match c0 with
    CNil -> c1
  | CCons(i,c) -> CCons(i,at c c1)

(* f11 : e -> string list -> c *)
let rec f11 e xs = match e with
  | Var(x) -> CCons(IAccess(Env.offset x xs),CNil)
  | Fun(x,e) -> CCons(IPush_closure(at (f11 e (x::xs)) (CCons(IReturn,CNil))),CNil)
  | App(e0,e1) ->
    at (CCons(IPush_env,CNil))
       (at (f11 e0 xs)
           (at (CCons(IPop_env,CNil))
               (at (f11 e1 xs) (CCons(ICall,CNil)))))
  | Control(x,e) -> CCons(IControl(f11 e (x::xs)),CNil)
  | Prompt(e) -> CCons(IPrompt(f11 e xs),CNil)
```

図 11. 命令列を cons リスト化した後のインタプリタ

```
let rec flat i = match i with
    IAccess(n) -> CCons(IAccess(n),CNil)
  | ...
  | ISeq (i0,i1) -> at (flat i0) (flat i1)

let rec flatC c = match c with
    CNil -> CNil
  | CCons(i,c) -> at (flat i) (flatC c)
  | CAppend(c0,c1) -> CAppend(flatC c0,flatC c1)

  (* ISeq (i0, i1) *)
  flatV (run_i10 (ISeq(i0,i1)) s c t m)
= flatV (run_i10 i0 s (CCons(i1,c)) t m)
= run_c11 (at (flat i0) (flatC (CCons(i1,c)))) (flatS s) (flatT t) (flatM m)
= run_c11 (at (flat i0) (at (flat i1) (flatC c))) (flatS s) (flatT t) (flatM m)
= run_c11 (at (at (flat i0) (flat i)) (flatC c)) (flatS s) (flatT t) (flatM m)
= run_c11 (at (flat (ISeq(i0,i))) (flatC c)) (flatS s) (flatT t) (flatM m)
```

図 12. 命令列を cons リスト化の正当性の証明：ISeq のケース

4.11 継続と trail をリスト化

前節の変換より、コンパイラから出力される命令列は cons リストで表されていることから、OCaml の実際のリストに変換することが可能である。具体的に、CNil が空リスト、CCons(i,c) は i::c とする。このような OCaml の実際のリストとなる構造を flat であると呼ぶようにする。スタックは継続に対応しているため、同様に flat とすることができる。

ここで、CAppend について、このデータ構造は trail の合成の際に使用される。そのため継続が CAppend のデータ構造で増幅する場合は、全て trail の中のみとなる。そして、trail の中身を実行する際、CAppend が run_c11 に渡された場合、flat な継続が現れるまで CAppend の展開が行われている。そして残りはまた trail の中で増幅される。つまり CAppend は次に実行するべき flat な継続（とスタックのペア）をつなげた構造であるとみなせる。よって、TNil を空リスト、Trail を (c, s) list とし、実行すべき flat な継続を順に積んでいくようにする。

上記の変換をした結果を図 13 に示す。まず型の変更として、継続 c、スタック s、trail t が実際の OCaml のリストとなっている。そして、メタ継続は継続 c、スタック s、trail t の 3 つ組のリストであったが、c * s は t のリストととらえることができるので、(c * s) * t は t の一つ以上のリストになる。すると (c * s * t) list は t list ととらえられることになる。VCont の型も同様にして t（(c * s) * t）を格納するようになった。変換によって得られたコンパイラと仮想機械は trail が実際のリストになったため、関数 cons や apnd は不要になった。

この変換の正当性を示す。型それぞれを flat 化する関数を定義する（図 14）。そして、次の同等性を証明する。

- flatC (f11 e xs) = f12 e xs
- flatV (run_c11 c s t m) = run_c12 (flatC c) (flatS s) (flatT t) (flatM m)

前者は項 e に関する帰納法によって証明される。f11 から出力された命令列を flat 化すると OCaml の実際のリストとなり、f12 から出力された命令列と等しくなる。後者は図 11 の仮想機械が進むステップ数に関する帰納法によって証明される。ここで、run_c11 に CAppend が渡されたときに、CAppend の中身を展開する必要がある分のステップ数が run_c12 ではなくなる。しかし、これは trail が flat 化されたことによるものであり、flatCS によって trail 内の実行するべき継続の順番は保持されているため問題ない。以上より、run_c11 と run_c12 は同等の挙動を示す。

```
type v = VFun of c * v list | VCont of t
       | VEnv of v list | VK of c
and  i = IAccess of int | IPush_closure of c | IReturn
       | IPush_env | IPop_env | ICall
       | IControl of c | IPrompt of c
and  c = i list
and  s = v list
and  t = (c * s) list
and  m = t list

(* run_c12 : c -> s -> t -> m -> v *)
let rec run_c12 c s t m = match (c,s) with
    ([],v::[]) -> (match t with
        [] -> (match m with
            [] -> v
          | ((c,s)::t)::m -> run_c12 c (v::s) t m)
      | (c,s)::t -> run_c12 c (v::s) t m)
  | (IAccess(n)::c,VEnv(vs)::s) -> run_c12 c ((List.nth vs n)::s) t m
  | (IPush_closure(c')::c,VEnv(vs)::s) -> run_c12 c (VFun(c',vs)::s) t m
  | (IReturn::_,v::VK(c)::s) -> run_c12 c (v::s) t m
  | (IPush_env::c,VEnv(vs)::s) -> run_c12 c (VEnv(vs)::VEnv(vs)::s) t m
  | (IPop_env::c,v::VEnv(vs)::s) -> run_c12 c (VEnv(vs)::v::s) t m
  | (ICall::c,v1::v0::s) -> (match v0 with
        VFun(c',vs) -> run_c12 c' (VEnv(v1::vs)::VK(c)::s) t m
      | VCont((c',s')::t') -> run_c12 c' (v1::s') (t' @ ((c,s) :: t)) m)
  | (IControl(c')::c,VEnv(vs)::s) -> run_c12 c' [VEnv(VCont((c,s)::t)::vs)] [] m
  | (IPrompt(c')::c,VEnv(vs)::s) -> run_c12 c' [VEnv(vs)] [] (((c,s)::t)::m)

(* f12 : e -> string list -> c *)
let rec f12 e xs = match e with
    Var(x) -> [IAccess(Env.offset x xs)]
  | Fun(x,e) -> [IPush_closure((f12 e (x::xs))@[IReturn])]
  | App(e0,e1) -> [IPush_env]@(f12 e0 xs)@[IPop_env]@(f12 e1 xs)@[ICall]
  | Control(x,e) -> [IControl(f12 e (x::xs))]
  | Prompt(e) -> [IPrompt(f12 e xs)]
```

図 13. 継続と trail をリスト化した後のインタプリタ

これで、仮想機械に相当するインタプリタを得ることができた。このプログラム変換は、もともと trail を関数によって表現する Shan のインタプリタからスタートしたが、それが途中で継続のリストで表現され、最終的にはスタックの列と表現されることがわかる。

5 control/prompt の仮想機械導出

図 13 の `run_c12` はすべて末尾呼び出しである。よって、これは仮想機械に変換できる [1]。仮想機械の状態遷移規則は図 15 のように定義される。状態は (c, s, t, m) の形で、それぞれの型は図 13 に定義されている。この仮想機械は、これまでの shift/reset の仮想機械と同様に、戻り番地や値の環境の保存と復元などの標準の呼び出し規則だけでなく、継続の切り取りや呼び出しでのデータスタックコピーの動作をモデル化している。

ここで trail は継続とスタックの組のリスト（型は `(c * s) list`）である。`c` を命令の先頭番地として `VK (c)` という値で表したとすると、trail はスタックのリストと考えることができる。

```
let rec flatI i = match i with
    IAccess(n) -> IAccess(n)
  | ...
  | IControl(c) -> IControl(flatC c)
  | IPrompt(c) -> IPrompt(flatC c)
and flatC c = match c with
    CNil -> []
  | CCons(i,c) -> (flatI i) :: (flatC c)

let rec flatV v = match v with
    M11.VFun (c, vs) -> M12.VFun (flatC c, List.map flatV vs)
  | M11.VCont (c, s, t) -> M12.VCont ((flatC c, flatS s) :: (flatT t))
  | ...
and flatS s = match s with
    M11.SNil -> []
  | M11.SCons (v, s) -> (flatV v) :: (flatS s)
and flatCS c s = match (c, s) with
    (CNil,SNil) -> ([], []) :: []
  | (CCons(_,_),SCons(_, _)) -> (flatC c, flatS s) :: []
  | (CAppend(c,c'),SAppend(s,s')) -> (flatCS c s) @ (flatCS c' s')
and flatT t = match t with
    TNil -> []
  | Trail(c,s) -> flatCS c s

let rec flatM m = match m with
    [] -> []
  | (c, s, t) :: m -> ((flatC c, flatS s) :: (flatT t)) :: (flatM m)
```

図 14. 継続と trail をリスト化の証明をする際に使う関数

c	$\Rightarrow$	$\langle c,\ [VEnv([])],\ [],\ []\rangle$
$\langle [],\ v :: [],\ [],\ []\rangle$	$\Rightarrow$	$\langle v\rangle$
$\langle [],\ v :: [],\ [],\ ((c,\ s) :: t) :: m\rangle$	$\Rightarrow$	$\langle c,\ v :: s,\ t,\ m\rangle$
$\langle [],\ v :: [],\ (c,\ s) :: t,\ m\rangle$	$\Rightarrow$	$\langle c,\ v :: s,\ t,\ m\rangle$
$\langle IAccess(n) :: c,\ VEnv(vs) :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c,\ (List.nth\ vs\ n) :: s,\ t,\ m\rangle$
$\langle IPush\_closure(c') :: c,\ VEnv(vs) :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c,\ VFun(c',\ vs) :: s,\ t,\ m\rangle$
$\langle IReturn :: \_,\ v :: VK(c) :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c,\ v :: s,\ t,\ m\rangle$
$\langle IPush\_env :: c,\ VEnv(vs) :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c,\ VEnv(vs) :: VEnv(vs) :: s,\ t,\ m\rangle$
$\langle IPop\_env :: c,\ v :: VEnv(vs) :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c,\ VEnv(vs) :: v :: s,\ t,\ m\rangle$
$\langle ICall :: c,\ v :: VFun(c',\ vs) :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c',\ VEnv(v :: vs) :: VK(c) :: s,\ t,\ m\rangle$
$\langle ICall :: c,\ v :: VCont((c',\ s') :: t') :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c',\ v :: s',\ t'\ @\ (c,\ s) :: t,\ m\rangle$
$\langle IControl(c') :: c,\ VEnv(vs) :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c',\ VEnv(VCont((c,\ s) :: t) :: vs) :: [],\ [],\ m\rangle$
$\langle IPrompt(c') :: c,\ VEnv(vs) :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c',\ VEnv(vs) :: [],\ [],\ ((c,\ s) :: t) :: m\rangle$

図 15. control/prompt の仮想機械

さらに、メタ継続は trail のリスト（型は `t list`）であるが、結局継続とスタックの組の列になっている。これは、状態 (c, s, t, m) の s, t, m 部分は、各々の間に区切り記号を用意しておけば、全部、つなげて一本のスタックと見ることができることを示している。そのように思って仮想機械を見直して見ると、限定継続命令はスタックの一部分を切り貼りしているととらえられる。

`IControl` 命令では、現在の継続・スタック・trail は値の環境にコピーされる。これが継続の切り取りに相当する。切り取られた継続が呼び出されたとき、つまり `ICall` 命令でスタックに `VCont` が積まれていたとき、保存されていた継続（命令列）・スタックが実行される。そのときに継続やスタックや trail は保存されていた trail と繋げられる。そして最後の s, t, m は順番が変わっていないことに注意しよう。これは、s, t はコピーする必要はなく、単に区切り記号を変更すれば良いということである。

`IPrompt` 命令では、現在の継続・スタック・trail はメタ継続に追加される。ここでも最後の s, t, m は順番が変わっていないので、コピーの必要はないことを示している。

control/prompt は trail が追加されただけであって命令 `IControl`, `IPrompt` での挙動は、先行研究での `IShift`, `IReset` と挙動に特に違いはなかった。キャプチャされた継続が呼び出されたとき（`VCont` が呼び出されたとき）に、現在の継続やスタックを trail に積む（control）かメタ継続に積む（shift）かの違いのみであった。すでに直接実装されている shift/reset の手法を大幅に変更せずとも contol/prompt の実装ができることが導けた。

6　まとめと今後の課題

本研究では λ 計算に control/prompt を拡張したインタプリタからコンパイラと仮想機械を導出した。今回は先行研究の shift/reset の仮想機械導出とほとんど同じ方法で行った。得られた仮想機械は限定継続演算子 control/prompt の実装手法を提供する。具体的にはデータスタックの挙動や、ヒープへのコピーをするべき箇所を示している。現在 shift/reset のみ直接実装ができている OchaCaml に、control/prompt を実装していきたい。

また限定継続演算子には他にも shift0/reset0 や control0/prompt0 があり、現在それらを実装するインタプリタに対しても仮想機械の導出を行っている。これらの演算子は継続がキャプチャされる際に、reset0（prompt0）の一つ外側の継続にアクセスできる。そのため、これらの命令が呼び出されたときに、その次の命令列の処理に違いが出ることになる。これらの演算子は algebraic effects and handlers に近いと考えられており [10, 18]、その実装の基盤を提供する。

謝辞

有益なコメントをくださった査読者の皆様に感謝申し上げます。本研究は一部 JSPS 科研費 18H03218 の助成を受けたものです。

参考文献

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. *BRICS Report Series*, Vol. 03, No. 14, 2003.

[2] Kenichi Asai and Arisa Kitani. Functional derivation of a virtual machine for delimited continuations. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pp. 87–98, 2010.

[3] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, Vol. 84, No. 1, pp. 108–123, 2015.

[4] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 38, No. 1, pp. 1–25, 2015.

[5] Olivier Danvy. Defunctionalized interpreters for programming languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pp. 131–142, New York, NY, USA, 2008. ACM.

[6] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pp. 151–160, 1990.

[7] Olivier Danvy and Andrzej Filinski. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, Vol. 2, No. 4, pp. pp. 361–391, 1992.

[8] R Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, Vol. 17, No. 6, pp. 687–730, 2007.

[9] Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pp. 180–190, New York, NY, USA, 1988. ACM.

[10] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, Vol. 1, No. ICFP, pp. 13:1–13:29, August 2017.

[11] Martin Gasbichler and Michael Sperber. Final Shift for Call/Cc: Direct Implementation of Shift and Reset. *SIGPLAN Not.*, Vol. 37, No. 9, pp. 271–282, September 2002.

[12] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A Generalization of Exceptions and Control in ML-like Languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pp. 12–23, New York, NY, USA, 1995. ACM.

[13] Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect Handlers via Generalised Continuations. *Journal of Functional Programming*, Vol. 30, , 2020.

[14] Atsushi Igarashi and Masashi Iwaki. Deriving compilers and virtual machines for a multi-level language. In *Asian Symposium on Programming Languages and Systems*, pp. 206–221. Springer, 2007.

[15] Yukiyoshi Kameyama and Takuo Yonezawa. Typed dynamic control operators for delimited continuations. In *International Symposium on Functional and Logic Programming*, FLOPS '08, pp. 239–254. Springer, 2008.

[16] Moe Masuko and Kenichi Asai. Direct Implementation of Shift and Reset in the MinCaml Compiler. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, ML '09, pp. 49–60, New York, NY, USA, 2009. Association for Computing Machinery.

[17] Marek Materzok and Dariusz Biernacki. Subtyping Delimited Continuations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pp. 81–93, New York, NY, USA, 2011. ACM.

[18] Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Typed equivalence of effect handlers and delimited control. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[19] John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pp. 717–740, 1972.

[20] John C Reynolds. Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation*, Vol. 11, No. 4, pp. 363–397, 1998.

[21] Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, Vol. 20, No. 4, pp. 371–401, 2007.