プログラム変換によるインタプリタからのコンパイラの導出

木谷　有沙
お茶の水女子大学
kitani.arisa@is.ocha.ac.jp

浅井　健一
お茶の水女子大学
asai@is.ocha.ac.jp

概 要　本研究の目的は，λ計算のインタプリタが与えられた際にそれと全く同じ動作をする仮想機械を導出する手法を確立し，shift/reset などのオペレータに対して正当性の保証された仮想機械の導出を可能にすることである．我々は既にプログラム変換により抽象機械を導出するまでの過程を示しているが，今回は導出した抽象機械をコンパイラと仮想機械に分割する手法に焦点をあてる．分割は Ager らによって提案された手法に則り，抽象機械を，項を受け取り解釈する部分と，その解釈結果とスタックを受け取り処理を行う部分に分かれるようカリー化し，前者をコンパイラ，後者を仮想機械にする．ただし本研究で用いているスタック導入，環境退避，戻り番地退避という変換の結果，カリー化前に抽象機械がコンビネータと再帰呼び出しに分解された形になっている点は異なっている．最終的に得られたコンパイラは，戻り番地をスタックに退避するなど，従来の手法に比べより現実のコンパイラに近いものとなっている．

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．たとえば “1 + 2” という計算を実行すると，まず「“1” と “2” の足し算である」と解釈され,「“1” とは整数の 1 である」「“2” とは整数の 2 である」と解釈が進み，最後に整数の 1 と 2 が足されて返ってくる．このとき,「“1” とは整数の 1 である」という部分を解釈している間,「あとで “2” を実行して，結果が分かったら足し算して返す」というのは計算システムが記憶しておかなければならない．この，あとで実行するために覚えている部分が継続である．

継続の切り取りや複製は，プログラムの制御フローの操作に等しい．継続処理オペレータがあれば，型の考慮された大域ジャンプが可能になる．切り取った継続を破棄することにより例外処理のような挙動をユーザが自分で定義することもできる．

しかし現在，継続処理を実装しているプログラミング言語は多くない．ユーザが機械語実装するにしても，継続処理にともなうスタック，ヒープの操作は自明になっておらず，困難である．

そこで本論文では，shift/reset を含むλ計算インタプリタから，それと全く同じ挙動を示すことが保証されているコンパイラおよび仮想機械を導出する．

shift/reset は限定継続を処理するオペレータで，CPS 形式のλ計算インタプリタ上において明確な定義が与えられている [4]．よって，shift/reset を定義するインタプリタに対して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導出した評価器は，shift/reset の正しい実装になっている．本論文では，shift/reset を定義するインタプリタに対しスタック導入，環境退避，戻り番地退避という変換を施してより低レベルな実装へ近づけていくが，これらの変換はいずれも，変換前後の評価器が同じ評価結果を返すものである．

本論文でインタプリタに施すスタック導入，環境退避といった変換は我々が先行研究において提案したもので，その変換前後の評価器の等価性についての詳細な証明も与えてある [10]．それに対し，戻り番地退避は本論文で新しく導入した変換である．この変換により，継続の中に自由変数として含まれる継続（戻り番地）が，スタックに積んで受け渡すように変わる．今回この変換を導入したのは，より低レベルな機械の挙動に近づける他に，スタック導入と環境退避だけでは継続の中に自由変数が残ってしまい，評価器を後述の手法でコンパイラと仮想機械に分割できなかったのを，解決するためでもある．

プログラム変換により低レベルな機械の挙動に近づいた評価器を，コンパイラと仮想機械に分割するには，Ager [1] らの手法を用いた．この手法では，まず評価器をカリー化し,「項を受け取ったら処理できる部分」と「それ以降の処理」とに分ける．前者がコンパイラ，後者が仮想機械の処理に相当するようになる．後者の処理は，そこまでに行ったプログラム変換により，コンビネータと再帰呼び出しの関数合成として書き換えることができる．各コンビネータを仮想機械の命令であると考えれば，前者の処理はコンパイラであると見なせる．

最終的に導出されるコンパイラと仮想機械は，コードポインタの受け渡しとスタックのコピーとして継続処理をモデル化できている．call/return に関しても，一度戻り番地をスタックに積んでおき後でスタックに積まれた番地に戻るという，実際のコンパイラ及び機械実装に近いものが得られている．

1.1 関連研究

Danvy らは λ 計算インタプリタを CPS 変換 [12]，非関数化 [13] することにより抽象機械を得られるとし，様々なインタプリタに対しそれと対応する抽象機械を導出している．CPS 変換および非関数化のような，変換前後のプログラムの振る舞いが等しくなることが保証される変換のみを用いれば，もとのインタプリタと最終的に導出された抽象機械は対応したものになるという考え方である．Ager ら [2] は，その手法で λ 計算インタプリタから CEK や CLS, SECD 等の抽象機械が得られることを示した．また Danvy と Millikin [6] は Landin の J オペレータを含む抽象機械を関数化，逆 CPS 変換することにより J オペレータを含む SECD マシンと λ 計算インタプリタの対応を示した．Danvy らのこの手法は “functional correspondence” と呼ばれているが，これは彼らが，用いているプログラム変換のうちでも特に非関数化（defunctionalization）がこの手法の核になると考えているからである．

本論文の，変換前後の等価性が保証された変換のみを用いてもとのインタプリタと対応する機械を導くというアプローチは，Danvy らのそれと同様である．しかし本論文では，彼らが重要視する非関数化は行わない．非関数化とはインタプリタ中のクロージャを一階のデータ構造へと書き換える変換で，インタプリタを抽象機械と見なして状態遷移規則を抽出するためには必要な過程だが，今回は変換途中のインタプリタから状態遷移規則を導くことはしない．

非関数化を用いて shift/reset を含む抽象機械を導出したものも，研究の一環として我々の研究グループで作成している [10]．shift/reset を含む λ 計算インタプリタにこの手法を適用する研究は Danvy ら自身によるものも既に存在していた [3] が，それが比較的抽象度の高い抽象機械の導出を目的及び成果としているのに対し，我々のグループではより現実の機械語実装に近い挙動をする抽象機械の導出を目指して，スタック導入や環境退避などの変換を新しく導入し，それらの変換前後の等価性の証明も行っている．このうちスタック導入と環境退避については本研究でも用いている．

コンパイラと仮想機械への分割を行っているのも，この研究がここまでで挙げた先行研究と大きく異なっている点のひとつである．インタプリタのコンパイラと仮想機械への分割は，Ager [1] らによって提案された．彼らは，インタプリタをカリー化し,「プログラムを受け取って命令列を返す」部分と「命令列と環境，継続を受け取り値を返す」部分に分割した．そして前者をコンパイラ，後者を仮想機械を位置づけ，ファンクタとモジュールに分けて実装したものを提示した．この手法の応用としては，五十嵐ら [9] が 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た研究が挙げられる．彼らの研究では，インタプリタを “functional correspondence” の先行研究同様 CPS 変換および非関数化した後，Ager らの手法を用いてコンパイラと仮想機械に分割する．本研究も，スタック導入，環境退避，戻り番地退避を行ったインタプリタに対して，Ager らの手法を用いて shift/reset を含む λ 計算に対する仮想機械とコンパイラを導出する．

また，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を作成している [11]．本研究の仮想機械は，定義に基づいて実装したインタプリタから導出したものでありながら，彼女らの実装のスタックコピーの振る舞いをモデル化しており，彼女らの shift/reset の実装に保証を与えるものとして期待されている．

1.2 概観

2 節において，shift/reset オペレータについて説明する．3 節では，shift/reset の定義から直接導かれる CPS インタプリタについて述べる．このインタプリタに対し施すプログラム変換を 4 節で説明する．その後の 5 節において抽象機械をファンクタとモジュールに分割したのち，6 節でコンパイラと仮想機械を得る．7 節では結論及び今後の課題を述べる．

2 shift/reset

継続を処理するためのオペレータとしては，shift/reset[4] の他にも call/cc [14]，control/prompt [7] などが挙げられる．前者は取って来られる継続の範囲が限定されず，継続が composable にならない．本論文では詳しく触れないが，call/cc を使用して shift/reset を間接的に実装した研究もある [8]．後者は shift/reset と同じく限定継続を処理できるオペレータだが，両者の間には挙動に若干の差がある．本研究では，CPS インタプリタにおいてその定義が明確に示されている shift/reset を扱う．

shift とは，現在の継続を切り取ってくるオペレータである．それに対し reset は，shift で切り取られる継

```
(* eval : t * string list * v list * c -> v *)
let rec eval (t, xs, vs, k) = match t with
    Var(x) -> k (List.nth vs (get(x, xs)))
  | Fun(x, t) -> k (VFun(fun (v1, k1) -> eval(t, x :: xs, v1 :: vs, k1)))
  | App(t0, t1) -> eval (t1, xs, vs, (fun v1
                    -> eval (t0, xs, vs, (fun v0
                         -> (match v0 with
                                VFun(f) -> f (v1, k)
                              | VCont(k') -> k (k' v1) )))))
  | Shift(t) -> eval (t, xs, vs, (fun v
                                   -> (match v with
                                          VFun(f) -> f (VCont(k), id)
                                        | VCont(k') -> k' (VCont(k)))))
  | Reset(t) -> k (eval (t, xs, vs, id))

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

図 1: eval1: shift/reset を定義する CPS インタプリタ

続の範囲を限定するオペレータである．

本研究では，shift(...) と書くと，括弧内の関数に現在の継続が渡されることにする．つまり，shift(fun k -> ...) と書くと，現在の継続が k に渡される．また，reset(...) と書くと，括弧内の shift で取って来られる継続を，括弧内の処理のみに限定することにする．

たとえば，reset(1 + shift(fun k -> k 2)) という計算は，shift オペレータによって「値を受け取ったら 1 を足して返す」という継続（shift の外側の "1 + "）が切り取られ，括弧内の k に渡される．その結果，1 + 2 となり，3 が返ってくる．

上の計算では shift する意味があまりないが，reset(1 + shift(fun k -> 3 * k 2)) という計算を考えると，継続の切り取りと呼び出しによって評価順序が変えられるのがより明確になる．shift がなければ 1 + (3 * 2) で 7 が返ってくるところだが，上の例と同様，shift によって「値を受け取ったら 1 を足して返す」という継続が切り取られ，括弧内の k に渡される．その結果，3 * (1 + 2) となり，9 が返ってくる．shift によって「1 を足す」場所が変更されたのである．このように，shift/reset は評価順序を操るオペレータであるとも捉えられる．

また，reset によって切り取る継続の範囲が限定されるのは，1 + reset(2 + shift(fun k -> 4 * k 3)) のような計算を考えると分かりやすい．reset の括弧内にある「値を受け取ったら 2 を足して返す」部分は shift 命令で取って来られるが，外側の「値を受け取ったら 1 を足して値を返す」部分は取って来られない．よって,「値を受け取ったら 2 を返す」という継続が k に渡され，1 + (4 * (2 + 3)) となり，21 が返ってくる．

本研究はこのような shift/reset オペレータを含む λ 計算インタプリタにプログラム変換を施していき，最終的には shift/reset を含む仮想機械を導出する．

3 shift / reset を含む λ 計算インタプリタ

本論文では，call-by-value の型なし λ 計算を以下のように shift/reset で拡張したものを対象とする．

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

変数，関数定義，関数呼び出しの他に shift と reset を記述できる．

値としては，関数定義の際に生成されるクロージャ $[\lambda(v, c).t]$ と，shift オペレータによって取り出される継続 $[c]$ の二つがある．

$$v \quad ::= \quad [\lambda(v, c).t] \mid [c]$$

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 ((v * c) -> v) (* クロージャ*)
       | VCont of c                (* 継続 *)
(* 継続 *)
and c  = (v -> v)
```

shift/reset を定義する評価器 `eval1` は図 1 のように実装される [5]．図中の関数 `id` は恒等関数であり，`(fun x -> x)` のことである．関数適用の際の引数の評価は左右どちらが先でもよいが，本論文では右から先に評価するものとする．

環境としては変数名のリスト `xs` と値のリスト `vs` の二つを持つようにした．`xs` の型は `string list`，`vs` の型は `v list` である．環境を二つに分割して実装したのは，後に評価器からコンパイラを抽出する際，項のみに依存する部分として変数名リストの処理も取り出したいからである．

関数 `get` は，参照したい変数の名前 `x` と変数名リスト `xs` を受け取ると，その変数の値が値リスト `vs` の何番目にあるかを調べて返す．今回は `xs` の n 番目の変数に対応する値が，`vs` の n 番目に積まれるように評価器を実装しているので，関数 `get` は「変数名リストを調べ，受け取った変数名と同じ文字列を見つけたら，それがリストの何番目かを返す」というだけの関数である．

この評価器は一般的な λ 計算の CPS インタプリタを，shift/reset の定義にしたがって拡張したものである [1]．`Shift(t)` を処理する際，評価器は現在の継続を `VCont(c)` という形にパッケージ化して `t`（正確には，`eval` で `t` を評価した結果得られるクロージャ）に渡している．`Reset(t)` の処理で現在の継続を `id` へと初期化しており，これにより shift オペレータで取り出される継続の範囲が限定されている．

以降の節で，この評価器に，変換前後の等価性が保証された変換のみを施していき，shift/reset を含む仮想機械を導出する．

4 プログラム変換

インタプリタに対し，変換前後のプログラムの等価性が保証されている変換のみを施したものは，変換前のインタプリタと全く同じ振る舞いを示すことが保証されている．

本研究では，インタプリタから，より機械語実装に近い挙動をする評価器を導出するために，スタック導入，環境退避，戻り番地退避という変換を用いる．このうちスタック導入と環境退避に関しては先行研究 [10] においてその詳細を述べており，変換前後の評価器の等価性も証明している．先行研究でのスタック導入，環境退避は，二回 CPS 変換及び非関数化された評価器に対して行われたものであり，一回しか CPS 変換せず非関数化はされていない評価器を扱う今回とは変換対象が異なっているが，変換の本質は同じである．等価性の証明を改めて行うにしても先行研究 [10] と全く同じ手順で証明できるため，今回用いるスタック導入，環境退避は既に変換前後の評価器の等価性が証明されているものとして扱う．

戻り番地退避は本研究で新しく導入したものであり，本節内で後に詳しく述べる．

本節で扱う変換は，次節以降でのコンパイラと仮想機械への分割を見越して，継続から自由変数を取り除く過程でもある．Ager らの手法 [1] を用いてコンパイラ及び仮想機械を導出するには，評価器をコンビネータの合成で書き換えられる状態にしておく必要があるため，この節でその準備をしているといえる．また，機械語実装における継続処理はコードポインタの受け渡しになると考えている以上，評価器によって受け渡しされる継続は単なるコードと同一視できるものになる必要があり，自由変数が含まれたままでは困るという動機もある．

変換前後の評価器の等価性が保証されている変換を用いてインタプリタから抽象機械を導出する研究として，Danvy らの研究が挙げられる．彼らは非関数化をその手法の中心に据えており，それもあって彼らはこの導出手法を “functional correspondence” と呼んでいる．しかし，非関数化はインタプリタを抽象機械と見なして状態遷移規則を得るのに用いられる変換であり，本研究では必要ない．また，非関数化すると継続を表現するための一階のオブジェクト及びこれを処理するための apply 関数を導入することになるが，低レベル実装における継続処理はコードポインタになると考えている本研究では，継続はそれ専用のオブジェクトとして扱うよりもコードと同一視できるようにしたい．よって非関数化は行わない．それよりも，機械語実装に近い挙動をする評価器を導出するためにスタック導入，環境退避，戻り番地退避という変換を用いている．

[1] 厳密に言うと，渡された式の一番外側を `reset()` で囲んであるものとして評価する．(例えば `shift(fun k -> k)` という式は `reset` が無く `shift` で取って来られる継続が限定されていないが，`eval1` に渡すと値が返ってくる．)

```
1 type v = VFun of ((s * c) -> s)  (* value *)
2         | VCont of s * c
3 and s = v list                    (* stack *)
4 and c = (s -> s)           (* continuation *)
5
6 (* eval : t * string list * v list * s * c -> v list *)
7 let rec eval (t, xs, vs, s, k) = match t with
8     Var(x) -> k ((List.nth vs (get(x, xs))) :: s)
9   | Fun(x, t) -> k ((VFun(fun ((v1 :: s1), k1) -> eval(t, x :: xs, v1 :: vs, s1, k1))) :: s)
10  | App(t0, t1) -> eval (t1, xs, vs, s, (fun s1 ->
11                   eval (t0, xs, vs, s1, (fun s0 ->
12                   (match s0 with
13                        VFun(f) :: v1 :: s0' -> f ((v1 :: s0'), k)
14                      | VCont(s', k') :: v1 :: s0' -> k (k' (v1 :: s' @ s0')))))))
15  | Shift(t) -> eval (t, xs, vs, s, (fun s0 ->
16                             (match s0 with
17                                  VFun(f) :: s0' -> f (VCont(s0', k) :: [], id)
18                                | VCont(s', k') :: s0' -> k' (VCont(s0', k) :: s'))))
19  | Reset(t) -> k ((eval (t, xs, vs, [], id)) @ s)
20
21 (* eval2 : t -> v *)
22 let eval2 t = eval (t, [], [], [], id)
```

図 2: eval2: eval1 にスタックを導入した評価器

4.1 スタック導入

eval1 の 6 〜 9 行目の fun 文は 6 行目の eval 呼び出し時に渡される継続である．この中の v1（8 行目）は，計算の中間結果であるが，fun 文の外側を参照しており，自由変数になっている．本研究では，低レベルな実装における継続処理はコードポインタの受け渡しになると考えており，インタプリタに変換を重ねた結果として，継続はコードと同一視できるようにしたい．そのため，継続に自由変数が含まれていては不都合であるし，計算の中間結果がコード中に積まれるというのも避けたい．また，後でコンパイラと仮想機械に分割するためにもここの自由変数は取り除いておく必要がある．よって，ここでは新しい引数としてスタック s を用意して，計算の中間値はそのスタックに積むようにする．そして，これまで値を一つだけ受け渡ししていた箇所は，その値を先頭に積んだスタックを受け渡しするようにする．

eval1 にスタックを導入した評価器を eval2 と呼ぶことにし，その定義を図 2 に示す．

スタック導入により，型も，これまで値 v 一つを引数として持っていたものが，スタック s を持つように変更されている．例えば継続を表す型 c は，eval1 では (v -> v) だったのが，eval2 では (s -> s) になる．

また評価器本体も，上述の通り変更されている．例えば，eval1 の 6 〜 9 行目の fun 文は値 v0 一つを受け取っていたが，それに対応する箇所である eval2 の 11 〜 14 行目の fun 文は，スタック s0 を受け取るようになっており，その中の match 文でスタックの中から計算の中間値を取り出している．

スタック導入は先行研究において，変換前後のプログラムの等価性が示された変換である．よって，eval1 と eval2 は同じ振る舞いをする評価器であるということが保証されている．

定理 1 (スタック導入前後の評価器の等価性).
任意の項 t に対し，eval1 と eval2 は，どちらも評価に失敗するか，または対応する値を返す [10].

以上のようなスタック導入によって，eval2 は計算の中間値がコード中に積まれることがなくなり，かわりに明示的に用意されたスタックに積まれるようになった．これは，より実際の機械語実装に近い振る舞いになっている．

しかし，まだ継続から自由変数が完全になくなったわけではない．そのため，次に示す環境退避によって，継続に残る自由変数をさらにスタックに積むように変換する．

4.2 環境退避

eval2 はまだ継続の中に自由変数がある．たとえば 11 行目の vs で，環境のうち変数の値を格納した部分である．先述の通り，本研究では継続処理は後に仮想機械の命令列の処理と同一視できるようにしたいこと，自由変数があってはコンパイラと仮想機械に分割できないことに加え，実際の機械語に近い命令列を持つ仮

```
1 type v = VFun of ((s * c) -> s)      (* value *)
2         | VCont of s * c
3         | VEnv of v list
4 and s = v list                       (* stack *)
5 and c = (s -> s)              (* continuation *)
6
7 let init_s = VEnv([]) :: []  (* initial stack *)
8
9 (* push_env : s -> s *)
10 let push_env s = match s with
11     VEnv(vs) :: s -> VEnv(vs) :: VEnv(vs) :: s
12 (* pop_env : s -> s *)
13 let pop_env s = match s with
14     v :: VEnv(vs') :: s' -> VEnv(vs') :: v :: s'
15
16 (* eval : t * string list * s * c -> v list *)
17 let rec eval (t, xs, s, k) = match t with
18     Var(x) -> (match s with VEnv(vs) :: s -> k ((List.nth vs (get(x, xs))) :: s))
19   | Fun(x, t) -> (match s with VEnv(vs) :: s
20                       -> k (VFun(fun (v1 :: s1, k1)
21                                   -> eval(t, x :: xs, VEnv(v1 :: vs) :: s1, k1)) :: s))
22   | App(t0, t1) -> eval (t1, xs, (push_env s), (fun s1 ->
23                    eval (t0, xs, (pop_env s1), (fun s0 ->
24                    (match s0 with
25                        VFun(f) :: v1 :: s0' -> f ((v1 :: s0'), k)
26                      | VCont(s', k') :: v1 :: s0' -> k (k' (v1 :: s' @ s0')))))))
27   | Shift(t) -> eval (t, xs, s, (fun s0 ->
28                              (match s0 with
29                                   VFun(f) :: s0' -> f (VCont(s0', k) :: [], id)
30                                 | VCont(s', k') :: s0' -> k' (VCont(s0', k) :: s'))))
31   | Reset(t) -> (match s with VEnv(vs) :: s -> k ((eval (t, xs, VEnv(vs) :: [], id)) @ s)
32
33 (* eval3 : t -> v *)
34 let eval3 t = List.hd(eval (t, [], init_s, id))
```

図 3: eval3: eval2 の環境をスタックに退避するようにした評価器

想機械の導出を目指している以上，環境（値リスト）が継続に積まれるのは避けたい．よって，環境 vs は評価器の引数としてではなく，スタックの先頭に積んで受け渡すように変換する．

eval2 の環境 vs をスタックに積むように変換したものを eval3 と呼ぶことにし，その定義を図 3 に示す．

eval2 からの主な変更点として，具体的にはまず値の型 v が拡張されていることが挙げられる．環境 vs をスタックに積むには vs を値として扱えるようにする必要があるためで，値の型に VEnv が追加されている．そして，環境をスタックに積む関数 push_env も新しく定義されている（10〜11 行目）．これはスタック中に保存された環境を読み取り複製してスタックに積む関数で，App 項の処理（eval3 の 22 行目）において，t1 の処理に進む直前に呼び出されている．ここで複製された環境は，後で t0 の処理（23 行目）をするときに使われる．t0 の処理に進む直前に pop_env という関数が呼び出され，この関数が，スタック中に保存された環境をスタック先頭に復帰させる．また，eval 関数は引数として vs を持たないように変わっている．これまで vs を参照していた箇所は，スタックに積まれた VEnv(vs) を参照するようになっている．例えば Var 項の処理（18 行目）は，(match s with VEnv(vs) :: s -> ... となっており，スタックの先頭に積まれた環境を参照している．

環境退避は先行研究において，変換前後のプログラムの等価性が示された変換である．よって，eval2 と eval3 は同じ振る舞いをする評価器であるということが保証されている．

定理 2 (環境退避前後の評価器の等価性).
任意の項 t に対し，eval2 と eval3 は，どちらも評価に失敗するか，または対応する値を返す [10].

以上に示した環境退避によって，環境が自由変数として継続に残る事がなくなった．評価器は関数呼び出し時に環境をスタックへと退避させるようになり，これは実際の機械語での関数呼び出しの際の環境の退避、復活を模倣していると見なせる．

しかし，まだ継続の中には自由変数がある．継続から自由変数を完全に取り除くために，次の変換を行う．

```
1 type v = VFun of ((s * c) -> s)                (* value *)
2         | VCont of s * c
3         | VEnv of v list
4         | VK of c
5 and s = v list                                 (* stack *)
6 and c = (s -> s)                            (* continuation *)
7
8 let init_s = VEnv([]) :: VK(id) :: [] (* initial stack *)
9
10 (* push_k : s -> c -> s *)
11 let push_k s k = match s with
12    VEnv(vs) :: s' -> VEnv(vs) :: VK(k) :: s'
13
14 (* eval : t * string list * s -> v list *)
15 let rec eval (t, xs, s) = match t with
16    Var(x) -> (match s with
17                  VEnv(vs) :: VK(k) :: s -> k ((List.nth vs (get(x, xs))) :: s))
18  | Fun(x, t) -> (match s with VEnv(vs) :: VK(k) :: s
19                     -> k (VFun(fun (v1 :: s1, k1)
20                                  -> eval(t, x :: xs, VEnv(v1 :: vs) :: VK(k1) :: s1)) :: s))
21  | App(t0, t1) -> eval (t1, xs, push_k (push_env s) (fun s1 ->
22                   eval (t0, xs, push_k (pop_env s1) (fun s0 ->
23                   (match s0 with
24                       VFun(f) :: v1 :: VK(k) :: s0' -> f ((v1 :: s0'), k)
25                     | VCont(s', k') :: v1 :: VK(k) :: s0' -> k (k' (v1 :: s' @ s0'))
26                   )))))
27  | Shift(t) -> eval (t, xs, push_k s (fun s0 ->
28                             (match s0 with
29                                 VFun(f) :: VK(k) :: s0' -> f (VCont(s0', k) :: [], id)
30                               | VCont(s', k') :: VK(k) :: s0' -> k' (VCont(s0', k) :: s'))))
31  | Reset(t) -> (match s with VEnv(vs) :: VK(k) :: s'
32                   -> k ((eval (t, xs, VEnv(vs) :: VK(id) :: [])) :: s))
33
34 (* eval4 : t -> v *)
35 let eval4 t = List.hd(eval (t, [], init_s))
```

図 4: eval4: eval3 の継続（戻り番地）をスタックに退避するようにした評価器

4.3 戻り番地退避

スタック導入と環境退避を経てなお，eval3 の継続にはまだ自由変数が残っている．25，26，29，30 行目の k である．これは関数呼び出し時の継続であり，低レベルな実装においては戻り番地に相当するものである．実際の機械語実装がそうであるように，戻り番地をスタックに積むように変更すれば，継続から自由変数を取り除くことができる．本研究では，このような，継続（戻り番地）をスタックに積むようにする変換を，戻り番地退避と呼ぶ．

eval3 を，継続 k をスタックに積むように変換したものを eval4 と呼ぶことにし，その定義を図 4 に示す．

この変換の手順は環境退避のときに似ている．まず，継続をスタックに積めるよう，値の型 v を拡張する．ここでは新しく，継続を引数に持つ VK が導入される（eval4 の 4 行目）．そして，スタックに継続を積むための関数 push_k が定義されている．この関数は，例えば App 項の処理（21 行目）において t1 の実行に進む前に呼ばれ，「t0 を実行する」という継続をスタックに積んでいる．また，評価器が引数として継続 k を持たなくなったので，これまで継続を受け取っていた箇所はスタックの中から継続を取り出すようになっている．例えば Var 項の処理は，eval2 では評価器の引数 k を参照していたところだが，eval3 では (match s with VEnv(vs) :: VK(k) :: s -> ... となっており，スタックに積まれた継続 k を参照している（16 〜 17 行目）．

戻り番地退避の結果，eval の再帰呼び出し前に必ず継続がスタックに保存されるようになった．継続をスタックに積む処理は，コードを複製しているように見えるが，実際にはコードの先頭番地を保存すればよい．この継続を保存する動作は低レベル実装で戻り番地を保存する動作に相当する．return に相当するのは，スタックから継続をとりだして実行している箇所である．

戻り番地退避前後の評価器の等価性は変換手順が類似している環境退避と同様の方法ですぐに検証できる．

```
(* eval : t * string list -> s -> s *)
let rec eval (t, xs) = match t with
    Var(x) -> (fun (VEnv(vs) :: VK(k) :: s) -> k ((List.nth vs (get(x, xs))) :: s))
  | Fun(x, t) -> (fun (VEnv(vs) :: VK(k) :: s) -> k (VFun(fun (v1 :: s1, k1)
                           -> eval(t, x :: xs) (VEnv(v1 :: vs) :: VK(k1) :: s1)) :: s))
  | App(t0, t1) -> (fun s -> eval (t1, xs) (push_k (push_env s) (fun s1 ->
                           eval (t0, xs) (push_k (pop_env s1) (fun s0 ->
                           (match s0 with
                                VFun(f) :: v1 :: VK(k) :: s0' -> f ((v1 :: s0'), k)
                              | VCont(s', k') :: v1 :: VK(k) :: s0' -> k (k' (v1 :: s' @ s0'))
                           ))))))
  | Shift(t) -> (fun s -> eval (t, xs) (push_k s (fun s0 ->
                        (match s0 with
                            VFun(f) :: VK(k) :: s0' -> f (VCont(s0', k) :: [], id)
                           | VCont(s', k') :: VK(k) :: s0' -> k' (VCont(s0', k) :: s')))))
  | Reset(t) -> (fun (VEnv(vs) :: VK(k) :: s)
                      -> k ((eval (t, xs) (VEnv(vs) :: VK(id) :: [])) @ s))

(* eval5 : t -> v *)
let eval5 t = List.hd((eval (t, [])) init_s)
```

図 5: eval5: eval4 をカリー化した評価器

定理 3 (戻り番地退避前後の評価器の等価性).
任意の項 t に対し，eval3 と eval4 は，どちらも評価に失敗するか，または対応する値を返す.

スタック導入，環境退避，戻り番地退避により，評価器はより実際の機械語実装に近い挙動をするものになり，継続から自由変数が完全に取り除かれた.

以降の節では，この評価器に Ager らの手法 [1] を用いてコンパイラ及び仮想機械を得る.

5 モジュール分割

Ager [1] らは，以下の手順でインタプリタを分割し，コンパイラと仮想機械を得る手法を提案した.

1 **評価器をカリー化**
評価器が項 t のみを受け取るようにカリー化する．カリー化により評価器は「項だけ受け取って処理する部分」となり,「環境や継続などを受け取ったら処理する部分」を出力するようになる．評価器の出力が仮想機械の命令列と見なせるようになれば，その評価器はコンパイラになっているといえる.

2 **関数合成に分解**
単にカリー化しただけでは，評価器の出力は入れ子状の fun 文になっているだけで，命令列になっているとは言いがたい．ここではそれぞれの fun 文で行う処理に名前を付けて関数（自由変数を含まない関数，コンビネータ）として定義し，評価器はこれらの関数を呼び出すことにする．前節までの変換で評価器から正しく自由変数が取り除かれていれば，評価器の処理はここで名付けた各関数と eval の再帰呼び出しの合成として書き換える事ができる．ここで出力する各関数が仮想機械の命令に相当すると考えると，評価器が仮想機械の命令列を出力していると見なせるようになる.

3 **ファンクタとモジュールに分けて実装**
ここまで変換を行った評価器を，ファンクタとモジュールに分けて実装する．評価器の eval 関数，つまり項を受け取ったら命令列を出す部分をファンクタとして実装する．各命令列が来たときに何をするかは別のモジュールとして実装する．これまでと全く同じように各関数を定義したモジュールを用意してファンクタに渡せば，これまでの評価器と全く同じものが生成される．ここで，各関数が仮想機械の命令列を出力するように定義したモジュールを作ってファンクタに渡せば，生成されるモジュールがコンパイラとなる．その場合，これまで各関数で行っていた処理を，命令列を受け取ったときに行うよう定義したモジュールを作成すれば，それが仮想機械になる.

本研究ではこの手法を応用して，前節までで導出した評価器から shift/reset を含む λ 計算のためのコンパイラ及び仮想機械を導出する.

```
(* function composition *)
let (>>) f g = (fun x -> g (f x))

let push_k k s = match s with VEnv(vs) :: s' -> VEnv(vs) :: VK(k) :: s'
let access i s = match s with VEnv(vs) :: VK(k) :: s -> k ((List.nth vs i) :: s)
let push_cls t' s = match s with
    VEnv(vs) :: VK(k) :: s -> k (VFun(fun (v1 :: s1, k1)
                                    -> t' (VEnv(v1 :: vs) :: VK(k1) :: s1)) :: s)
let call s = match s with
    VFun(f) :: v1 :: VK(k) :: s0' -> f ((v1 :: s0'), k)
  | VCont(s', k') :: v1 :: VK(k) :: s0' -> k (k' (v1 :: s' @ s0'))
let shift s = match s with
    VFun(f) :: VK(k) :: s0' -> f (VCont(s0', k) :: [], id)
  | VCont(s', k') :: VK(k) :: s0' -> k' (VCont(s0', k) :: s')
let reset t' s = match s with
    VEnv(vs) :: VK(k) :: s -> k ((t' (VEnv(vs) :: VK(id) :: [])) @ s)

(* eval: t * string list -> s *)
let rec eval (t, xs) = match t with
    Var(x) -> access (get(x, xs))
  | Fun(x, t) -> push_cls (eval (t, x :: xs))
  | App(t0, t1) -> (push_env
                  >> push_k(pop_env >> push_k(call) >> (eval (t0, xs)))
                  >> (eval (t1, xs)))
  | Shift(t) -> push_k(shift) >> (eval (t, xs))
  | Reset(t) -> reset (eval (t, xs))

(* eval6: t -> v *)
let eval6 t = List.hd((eval (t, [])) init_s)
```

図 6: eval6: eval5 を関数合成に分解した評価器

5.1 カリー化

前節までで導出した評価器をカリー化し，項だけ受け取れば処理できる部分とそれ以降に分割する．評価器のうち項だけあればできる処理を抽出したものがコンパイラであり，それ以外の処理は仮想機械が処理する部分となると考えられる．

本研究で扱う評価器は項 t と環境（変数名リスト）xs とスタック s を引数として持っていたので，このうち項 t と環境（変数名リスト） xs だけあれば処理できる部分を抽出する．項だけでなく変数名リストも受け取るようにするのは，この変数名リストは項のみに依存しており，先に処理したい部分だからである．カリー化により評価器は項 t と xs だけを受け取るようになり，スタック s を受け取ったあとにしかできない処理は eval 関数では行わず，あとでスタックを受け取ってから実行するように fun 文にして残すようにする．

eval4 をカリー化した評価器を eval5 と呼ぶことにし，その定義を図 5 に示す．

先に述べた通り，eval5 では評価器の引数が項 t と環境（変数名リスト） xs だけになっている．引数として受け取らなくなったスタック s は，出力結果の fun 文で受け取るようになっている．

カリー化は，変換前後の評価器の等価性は保証されている．よって，eval4 と eval5 は同じ振る舞いをする評価器であるということが保証されている．

定理 4 (カリー化前後の評価器の等価性).
任意の項 t に対し，eval4 と eval5 は，どちらも評価に失敗するか，または等しい値を返す．

5.2 関数合成に分解

項と変数名リストだけで処理できる部分を抽出しコンパイラとする意図でカリー化を行ったが，コンパイラというからにはその出力結果は仮想機械の命令列になってほしい．eval5 ではまだ，入れ子状になった fun 文をそのまま返すようになっており，仮想機械の命令列になっているとは言えない．

しかし前節でのプログラム変換の結果，この fun 文はは自由変数を含まないコンビネータになっており，独立した関数にすることができ，したがって，それらの関数の合成として書き表すことができる．ここでは各 fun 文に名前を付けて関数として定義し，それらの関数の合成として書き変える．

この変換を eval5 に施したものを eval6 と呼ぶことにし，その定義を図 6 に示す．

```
1 module type INTERPRETATION =
2 sig
3   type computation
4   type result
5   val push_env : computation
6   val pop_env : computation
7   val push_k : computation -> computation
8   val access : int -> computation
9   val push_cls : computation -> computation
10  val call : computation
11  val shift : computation
12  val reset : computation -> computation
13  val combine : computation -> computation -> computation
14  val compute : computation -> result
15 end
```

図 7: INTERPRETAION: ファンクタに渡すシグネチャ

変換の結果，eval 関数は再帰呼び出しと変換の際定義した関数，およびこれまでに定義した push_env，pop_env，push_k などの関数の合成になっている．例えば Shift 項（25 行）は，push_k(shift) という関数と，再帰呼び出し eval(t, xs) の合成になる．

Var 項の処理は eval5 では fun 文を返していたが，eval6 ではこの fun 文に access という名前をつけて関数として定義している（4〜5 行目）．そして，Var 項の処理としてはこの access 関数を返すように変えている．ここで，eval5 では Var 項の中で List.nth vs (get (x, xs)) となっていたところが，access 関数では List.nth vs i になっている．これは，get (x, xs) がスタック s に依存しない処理で，eval 関数内で実行してしまえるからである．

eval6 の評価器の出力結果の各関数が仮想機械の命令に相当すると考えると，この評価器は項を受け取って命令列を出力すると見なせるようになる．

関数を展開すれば，分解する前の評価器と全く同じものが得られることより，この変換前後の評価器の等価性は明らかである．

定理 5 (関数合成に分解する前後の評価器の等価性).
任意の項 t に対し，eval5 と eval6 は，どちらも評価に失敗するか，または等しい値を返す．

5.3 ファンクタとモジュールに分けて実装

ここまでの変換によって，評価器の処理はコンパイラと仮想機械に分離されている．eval6 の関数 eval がコンパイラに相当する部分で，access などの関数が行っている処理が仮想機械の行う処理である．これらはファンクタとモジュールに分けて実装することが可能である．eval 関数をファンクタで実装しておき，access などの関数が行う処理の定義は別のモジュールで行う．各関数の処理を eval6 と同じように定義したモジュールをファンクタに渡せば，eval6 と同じ評価器が得られる．ここで各関数の定義を,「仮想機械の命令列を出力する」にすれば，ファンクタに渡して生成される評価器はコンパイラになるが，これについては次節で説明する．本節ではひとまず eval6 の定義をファンクタとモジュールに分割するという意図のもと，eval6 と同じ評価器が得られるファンクタおよびモジュールの実装を説明する．

まず，ファンクタで実装するにはそのファンクタの定義に使うシグネチャを定義する必要があり，これは図 7 のようになる．このシグネチャでは関数の型などを定義するが，eval 関数の出力を表す computation という型と，処理結果を表す result という型を使っている．eval6 と同じ評価器を作る場合は computation は s -> s 型，result は v 型としてしまってもよさそうだが，こうして抽象化しているのは，後に同じファンクタを使ってコンパイラを生成する際に eval 関数の出力や結果の型が変わるからである．

次に，このシグネチャを使って定義したファンクタが図 8 の MkProcessor である．このファンクタ内に eval 関数を定義されている．この eval 関数の定義は eval6 の eval とほぼ同様である．

最後に，ファンクタに渡すモジュールを定義したものが図 9 の I_Eval である．このモジュールの中に，access などの関数の定義を記述する．ここも，eval6 での定義と全く同じにする．3〜4 行目で，computation 型および result 型も定義してある．eval6 と同じなので，eval 関数の出力の型は s -> s，最終結果の型は v となる．28 行目の Evaluator = MkProcessor(I_Eval) でファンクタにモジュールが渡され，eval6 と同じ評価器が生成される．(Evaluator.f が eval6 と同じ挙動を示す.) モジュールを関数展開すれば等し

```
module MkProcessor =
  functor (I: INTERPRETATION) -> struct
  let (>>) = I.combine

  let rec eval (t, xs) = match t with
      Var(x) -> I.access (get(x, xs))
    | Fun(x, t) -> I.push_cls (eval (t, x :: xs))
    | App(t0, t1) -> (I.push_env
                      >> I.push_k (I.pop_env >> I.push_k(I.call) >> (eval (t0, xs)))
                      >> (eval (t1, xs)))
    | Shift(t) -> I.push_k (I.shift) >> (eval (t, xs))
    | Reset(t) -> I.reset (eval (t, xs))
  let f t = I.compute (eval (t, []))
  end
```

図 8: MkProcessor: eval6 を分解して得られるファンクタ

```
module I_Eval =
  struct
    type computation = s -> s
    type result = v
    let id x = x
    let init_s = VEnv([]) :: VK(id) :: []

    let (>>) f g = (fun x -> g (f x))
    let push_env s = match s with VEnv(vs) :: s -> VEnv(vs) :: VEnv(vs) :: s
    let pop_env s = match s with v :: VEnv(vs') :: s' -> VEnv(vs') :: v :: s'
    let push_k k s = match s with VEnv(vs) :: s -> VEnv(vs) :: VK(k) :: s
    let access i s = match s with VEnv(vs) :: VK(k) :: s -> k ((List.nth vs i) :: s)
    let push_cls t' s = match s with VEnv(vs) :: VK(k) :: s
                          -> k ((VFun(fun ((v1 :: s1), k1)
                             -> t' (VEnv(v1 :: vs) :: VK(k1) :: s1))) :: s)
    let call s = match s with
        VFun(f) :: v1 :: VK(k) :: s0' -> f ((v1 :: s0'), k)
      | VCont(s', k') :: v1 :: VK(k) :: s0' -> k (k' (v1 :: s' @ s0'))
    let shift s = match s with
        VFun(f) :: VK(k) :: s0' -> f (VCont(s0', k) :: [], id)
      | VCont(s', k') :: VK(k) :: s0' -> k' (VCont(s0', k) :: s')
    let reset t' s = match s with
        VEnv(vs) :: VK(k) :: s -> k ((t' (VEnv(vs) :: VK(id) :: [])) @ s)
    let combine f g = (fun x -> g (f x))
    let compute t' = List.hd(t' init_s)
  end

module Evaluator = MkProcessor(I_Eval)
```

図 9: I_Eval: MkProcessor に渡すと eval6 と同じ評価器を生成するモジュール

い評価器が得られることより，この実装の正当性は自明である．

定理 6 (モジュール分割前後の評価器の等価性).
任意の項 t に対し，Evaluator.f と eval6 は，どちらも評価に失敗するか，または等しい値を返す．

この節では，これまでの評価器と同じ挙動をするものを生成するファンクタとモジュールの実装について説明した．次節では同じファンクタを使って，今度はランタイム処理（スタックを受け取って行う処理）をせず命令列を出力する，つまりコンパイルだけするモジュールの実装について説明する．そして，そのコンパイル結果を受け取りランタイム処理を行う仮想機械の実装を行う．

6 コンパイラと仮想機械

前節ではそれまでの評価器と同じものを生成するファンクタおよびモジュールを実装したが，本節では同じファンクタに別のモジュールを渡してコンパイルを得る．ファンクタで定義されている eval 関数は既に述べた通り，項を処理して関数合成にして返してくれるので，ほぼコンパイラの働きをしていると言って良い．access などをそれぞれ仮想機械の命令と見なして，関数を合成するかわりにリストの連結を行うように

```
module I_Compile =
  struct
    type computation = i list
    type result = i list

    let push_env = [IPushEnv]
    let pop_env = [IPopEnv]
    let push_k k = [IPushK(k)]
    let access i = [IAccess(i)]
    let push_cls t' = [IPushCls(t')]
    let call = [ICall]
    let shift = [IShift]
    let reset t' = [IReset(t')]
    let combine f g = f @ g
    let compute t' = t'
  end
module Compiler = MkProcessor(I_Compile)
```

図 10: I_Compile: MkProcessor に渡すとコンパイラになるモジュール

すれば，eval 関数は本当に仮想機械の命令列を出力するコンパイラになる．よって，そのように定義したモジュールを実装して I_Eval のかわりに MkProcessor に渡せば，コンパイラが得られる．

まずは，仮想機械の命令を以下のように定義する．

```
i = IPushEnv
  | IPopEnv
  | IPushK of i list
  | IAccess of int
  | IPushCls of i list
  | ICall
  | IShift
  | IReset of i list
```

処理にスタック以外の引数を持っていたものはその引数を保持するようになっている．IPushCls と IReset が持つ i list 型の引数は，より低レベルな実装ではコードポインタになると考えられる．

これらの命令列を出力するように各関数を定義したモジュールが図 10 の I_Compile である．先に述べた通り，I_Eval ではランタイム処理を行っていた関数が，仮想機械の命令列を返すだけになっている．関数合成のかわりにリストの結合が行われるようにも変更されている（14 行目）．また，eval 関数の出力とコンパイラの出力結果はともに命令列であるため，computation と result の型は i list になっている（3 〜 4 行目）．17 行目の Compiler = MkProcessor(I_Compile) において，コンパイラが生成される．(Compiler.f が項を受け取り命令列を返す関数になる.)

ここで出力する命令列を処理する仮想機械は，これまで access 等の関数が行っていた処理を IAccess 等の命令がきたときに行うよう定義したものであり，図 11 のように定義される．

仮想機械 VM の処理そのものは，I_Eval や eval6 の各関数が行っていた処理と全く同じである．Compiler.f で項 t をコンパイルした結果を VM.f で処理した結果は，I_Eval.f や eval6 で t を処理した結果と等しくなる．

本節で行った仮想機械の命令列の定義および仮想機械の実装は非関数化の一種であるともとらえることができる．非関数化とは他から独立している関数を一階のオブジェクトに置き換える変換で，関数を返す部分をそのオブジェクトを返すように変更し，そのオブジェクトを処理するための apply 関数を別に用意する，というものである．仮想機械の命令列が一階のオブジェクト，仮想機械がそのオブジェクトを処理するための apply 関数であると考えると，我々が 6 節で行ったことは非関数化であると見なせる．関連研究として紹介した五十嵐ら [9] はコンパイラと仮想機械への分割は非関数化であるとして，ファンクタとモジュールへの分割はせずに評価器（本論文の eval6 のようなもの）を非関数化している．

本節で行ったコンパイラ及び仮想機械の実装は非関数化の一種といえることにより，非関数化の正当性を根拠に，コンパイラと仮想機械の実装もその正当性が保証されたものであるといえる．

定理 7 (コンパイラと仮想機械への分割の正当性).
任意の項 t に対し，Compiler.f >> VM.f と Evaluator.f は，どちらも評価に失敗するか，または等しい

```
module VM =
  struct
    type v = VFun of ((s * c) -> s)
           | VCont of s * c
           | VEnv of v list
           | VK of c
    and s = v list
    and c = i list

    let id = []
    let init_s = VEnv([]) :: VK(id) :: []

    let rec run i s = match i with
        IPushEnv :: i' -> (match s with VEnv(vs) :: s -> run i' (VEnv(vs) :: VEnv(vs) :: s))
      | IPopEnv :: i' -> (match s with
                            v :: VEnv(vs') :: s' -> run i' (VEnv(vs') :: v :: s'))
      | IPushK(k) :: i' -> (match s with VEnv(vs) :: s -> run i' (VEnv(vs) :: VK(k) :: s))
      | IAccess(i) :: _ -> (match s with VEnv(vs) :: VK(k) :: s -> run k ((List.nth vs i) :: s))
      | IPushCls(t') :: _ -> (match s with VEnv(vs) :: VK(k) :: s
                                -> run k ((VFun(fun ((v1 :: s1), k1)
                                                -> run t' (VEnv(v1 :: vs) :: VK(k1) :: s1))) :: s))
      | ICall :: _ -> (match s with
                          VFun(f) :: v1 :: VK(k) :: s0' -> f ((v1 :: s0'), k)
                        | VCont(s', k') :: v1 :: VK(k) :: s0' -> run k (run k' (v1 :: s' @ s0')))
      | IShift :: _ -> (match s with
                           VFun(f) :: VK(k) :: s0' -> f (VCont(s0', k) :: [], id)
                         | VCont(s', k') :: VK(k) :: s0' -> run k' (VCont(s0', k) :: s'))
      | IReset(t') :: _ -> (match s with VEnv(vs) :: VK(k) :: s
                              -> run k ((run t' (VEnv(vs) :: VK(id) :: [])) @ s))
      | [] -> s

    let f i = List.hd(run i init_s)
  end
```

図 11: VM: 仮想機械

値を返す.

以上，shift/reset を含む λ 計算にスタック導入，環境退避，戻り番地退避といった変換を用い，それをファンクタとモジュールに分割して，コンパイラ及び仮想機械が得られた．その定義を以下に示す.

- 入力項と仮想機械の命令列

$$
\begin{array}{rcl}
t & ::= & x \mid \lambda x.\, t \mid t_0\ t_1 \mid \mathtt{shift}(t) \mid \mathtt{reset}(t) \\
i & ::= & \mathtt{IPushEnv} \mid \mathtt{IPopEnv} \mid \mathtt{IPushK}(il) \mid \mathtt{IAccess}(n) \mid \mathtt{IPushCls}(il) \mid \mathtt{ICall} \mid \mathtt{IShift} \mid \mathtt{IReset}(il) \\
il & ::= & [] \mid i :: il
\end{array}
$$

- コンパイラ

$$
\begin{array}{rcl}
[\![x, xs]\!] & = & [\mathtt{IAccess}(get(x, xs))] \\
[\![\lambda x.\ t,\ xs]\!] & = & [\mathtt{IPushCls}([\![t, x :: xs]\!])] \\
[\![t_0\ t_1,\ xs]\!] & = & \mathtt{IPushEnv} :: \mathtt{IPushK}(\mathtt{IPopEnv} :: \mathtt{IPushK}([\mathtt{ICall}]) :: [\![t_0,\ xs]\!]) :: [\![t_1,\ xs]\!] \\
[\![\mathtt{shift}(t),\ xs]\!] & = & \mathtt{IPushK}([\mathtt{IShift}]) :: [\![t,\ xs]\!] \\
[\![\mathtt{reset}(t),\ xs]\!] & = & [\mathtt{IReset}([\![t,\ xs]\!])]
\end{array}
$$

- 値

$$
\begin{array}{rcl}
v & ::= & [il, vs] \mid [s, il] \mid [vs] \mid [il] \\
s & ::= & [] \mid v :: s \\
vs & ::= & [] \mid v :: vs
\end{array}
$$

- 状態遷移規則
  仮想機械の状態遷移規則は図 12 のように定義される．⟨ ⟩ の中に ⟨ ⟩ が含まれる場合は，内側の ⟨ ⟩ から先に評価を行う．各状態に対して適用される状態遷移規則は決定的に定められる．

$$
\begin{array}{rcl}
il & \Rightarrow & \langle il, \mathtt{VEnv}([]) :: \mathtt{VK}([]) :: [] \rangle \\
\hline
\langle \mathtt{IPushEnv} :: il', \mathtt{VEnv}(vs) :: s \rangle & \Rightarrow & \langle il', \mathtt{VEnv}(vs) :: \mathtt{VEnv}(vs) :: s \rangle \\
\langle \mathtt{IPopEnv} :: il', v :: \mathtt{VEnv}(vs) :: s \rangle & \Rightarrow & \langle il', \mathtt{VEnv}(vs) :: v :: s \rangle \\
\langle \mathtt{IPushK}(il) :: il', v :: \mathtt{VEnv}(vs) :: s \rangle & \Rightarrow & \langle il', \mathtt{VEnv}(vs) :: \mathtt{VK}(il) :: s \rangle \\
\langle \mathtt{IAccess}(n) :: \_, \mathtt{VEnv}(vs) :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il', (List.nth\ vs\ n) :: s \rangle \\
\langle \mathtt{IPushCls}(il) :: \_, \mathtt{VEnv}(vs) :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il', \mathtt{VFun}(il, vs) :: s \rangle \\
\langle \mathtt{ICall} :: \_, \mathtt{VFun}(il, vs) :: v :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il, \mathtt{VEnv}(v :: vs) :: \mathtt{VK}(il') :: s \rangle \\
\langle \mathtt{ICall} :: \_, \mathtt{VCont}(s', il'') :: v :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il, \langle il'', v :: s' @ s \rangle \rangle \\
\langle \mathtt{IShift} :: \_, \mathtt{VFun}(il, vs) :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il, \mathtt{VEnv}(\mathtt{VCont}(s, il') :: vs) :: \mathtt{VK}(id) :: [] \rangle \\
\langle \mathtt{IShift} :: \_, \mathtt{VCont}(s', il'') :: v :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il'', \mathtt{VCont}(s, il') :: s' \rangle \\
\langle \mathtt{Reset}(il) :: \_, \mathtt{VEnv}(vs) :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il', \langle il, \mathtt{VEnv}(vs) :: \mathtt{VK}([]) :: [] \rangle @ s \rangle \\
\hline
\langle [], v :: \_ \rangle & \Rightarrow & \langle v \rangle
\end{array}
$$

図 12: eval6 から導かれる状態遷移規則

導出の結果得られた仮想機械では，継続と命令列の型がともに i list となっている．関数適用や shift のコンパイル結果を見ると，スタックに積まれる継続が命令列になっているのがわかる．継続を積む操作は命令列を積む操作となったのである．この実装では命令列を直接スタックに積んでいるが，これは命令列の先頭へのポインタを積んでいると見なすことができ，この仮想機械は継続処理のコードポインタの受け渡しとしての実装をモデル化している．shift の処理の際には，コードだけでなく，現在のスタックのコピーも渡しており，これは継続処理におけるスタックコピーをモデル化したものであるといえる．

例えば IShift のときの処理（図 12 の下から 4 行目）ではスタック中の VK から命令列を取り出して，現在のスタックを複製したものと一緒に値環境に積んでいるが，これは継続に相当する部分のコードへのポインタをスタックから取り出して来ていると見なせる．また，スタックのコピーを積んでいるが，これはスタックの中身をヒープへコピーし、ヒープのその箇所のアドレスを渡す操作であると考えられる．

スタック s の中に VCont(s, c) が含まれれば自己参照が起こり，スタックをコピーする際，自明でないアドレスの操作が要求されるのではないかということが先行研究では懸念されていたが，VCont が持ちうるのはスタックへのポインタではなくスタックの中身をコピーしたヒープへのポインタであると考えれば，この懸念は解消される．

また，CPS 形式のインタプリタでは処理が値まで落ちたら継続の処理に進むようになっていたが，ここまでのプログラム変換の結果，仮想機械では「スタックに積まれた戻り番地を読み取りジャンプする」という挙動になっていると見なせる．これは低レベルな機械での call/return の振る舞いを模倣しているといえる．

ただし今回導出した仮想機械では末尾再帰になっていない箇所があり，状態の中に状態が入っている等，通常状態遷移機械と言われて想像されるものと異なる部分がある．インタプリタを再度 CPS 変換すると全ての処理が末尾再帰になり，状態が入れ子状になるのは解消されるが，その場合は二度目の CPS 変換によって出現するもう一つの引数（メタ継続）もスタックに退避させる変換が必要になる．これについては今後さらに考察を行う予定である．

7　まとめと今後の課題

本研究では，shift/reset を含む λ 計算のインタプリタに対してスタック導入，環境退避，戻り番地退避という変換を施し，より低レベルな機械語実装に近く，継続から自由変数が取り除かれた実装を得た．そしてその評価器に Ager らの手法を用いて，コンパイラと仮想機械を得た．インタプリタに対し，変換前後の等価性が保証されている変換のみを用いているため，本研究で得たコンパイラと仮想機械は，もともとのインタプリタと全く同じ挙動をすることが保証されている．かつ，このコンパイラと仮想機械は実際の機械語実装における call/return に近い振る舞いを模倣できている上，shift/reset の処理をコードポインタの受け渡しとしてモデル化できている．

インタプリタに対して変換前後の等価性が保証されている変換を用いて低レベルな実装を導くというのは Danvy らの “functional correspondence” と同じであるが，彼らが非関数化をその変換の要であると捉えているのに対し，本論文では非関数化は抽象機械の遷移規則導出のための過程であると見なしており，用いていない．より低レベルな実装を目指して彼らとは別の変換，スタック導入，環境退避，戻り番地退避を行っている．

コンパイラと仮想機械の導出については，`Ager` らの手法にほぼ忠実に行ったが，各手順について先行研究よりも本論文内で詳しく解説した．

導出したコンパイラ及び仮想機械は，スタックに戻り番地を退避し，あとでスタックから戻り番地を復活させるといった，実際の機械語での call/return を模倣できている．また shift/reset についてもスタックから戻り番地を取り出して，現在のスタックがコピーされたヒープへのアドレスと一緒に shift の括弧内の関数に渡し，その関数内では渡された番地にジャンプする，といった挙動になっており，低レベルな機械における shift/reset の実装モデルを示しているといえる．

今回導出した仮想機械の処理は末尾呼び出しになっていない箇所が残っているが，評価器をもう一度 CPS 変換すればすべて末尾再帰になる．今後，CPS 変換したものや非関数化したものに対しても本研究のプログラム変換を施し，考察を行う予定である．

また，低レベルな機械の挙動をモデル化した仮想機械を得たが，これと実際の機械語の命令セットとの対応をより明らかにし，shift/reset の実際の機械語実装での正当性を保証するものにしたい．

謝辞

本研究を読んでくださり，有益なコメントを下さった査読者の皆様に感謝いたします．

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, BRICS, RS-03-14, 2003.

[2] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. Technical report, BRICS, RS-03-13, 2003.

[3] M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the cps hierarchy. *Logical Methods in Computer Science*, Vol. 1, 2:5, pp. 1–39, 2005.

[4] O. Danvy and A. Fillinski. A functional abstraction of typed contexts. Technical report, DIKU, 89/12, 1989.

[5] O. Danvy and A. Fillinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160, 1990.

[6] O. Danvy and K. Millikin. A rational deconstruction of landin’s secd machine with the j operator. *Logical Methods in Computer Science*, Vol. 4, 4:12, pp. 1–67, 2008.

[7] M. Felleisen. The theory and practice of first-class prompts. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 180–190, 1988.

[8] A. Filinski. Representing monads. In *Conference Record of the 21th Annual ACM Symposium on Principles of Programming Languages*, pp. 446–457, 1994.

[9] A. Igarashi and M. Iwai. Deriving compilers and virtual machines for a multi-level languages. In *Proceedings of the Fifth Asian Symposium on Programming Languages and Systems (APLAS’07), LNCS 4807*, pp. 206–221, 2007.

[10] 木谷有沙, 浅井健一. 限定継続処理の抽象機械導出のためのプログラム変換. コンピュータソフトウェア, 2010. 掲載予定.

[11] M. Masuko and K. Asai. Direct implementation of shift and reset in the mincaml compiler. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, pp. 49–60, 2009.

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

[13] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Proceedings of the ACM National Conference*, Vol. 1, No. 2, pp. 717–740, 1972. reprinted in *Higher-Order and Symbolic Computation*, Vol. 11, No. 4, pp. 363–397, 1998. Kluwer Academic Publishers.

[14] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. Revised^6 report on the algorithmic language scheme. http://www.r6rs.org/, 2007.