shift/reset を用いた stepper の実装に向けて

叢悠悠 [1], 浅井健一 [1], 戸次大介 [2]

[1] お茶の水女子大学
[2] お茶の水女子大学, 国立情報学研究所, 独立行政法人科学技術振興機構, CREST
{so.yuyu, asai, bekki}@is.ocha.ac.jp

概 要 stepper とは、プログラムの実行を 1 ステップごとに止めて、表示するツールのことである。プログラミングの初学者にとって、直接値が返される通常のインタプリタを用いてプログラムの挙動を理解したり、デバッグをしたりすることは必ずしも簡単ではないが、stepper を使ってプログラムを実行すれば、計算の過程を一つ一つ追うことができる。また、shift/reset (Danvy and Filinski, 1990) のような限定継続命令を含むものなど、複雑なプログラムのステップ実行は、熟練者にとっても有益である。しかし、現時点で stepper を提供している言語は Racket のみである (Clements et al., 2001)。そこで、本研究では OCaml の基本的な構文と限定継続命令 shift/reset を対象言語とする stepper の実装を試みる。stepper の実装では、注目している redex を取り囲む計算と、その redex の簡約結果を使って、もとの表現を再構成する必要があるが、本研究ではメタ言語において shift 命令を使用することでこの動作を実現している。

1 はじめに

λ 計算において項を簡約するときは、一度に一つの redex を簡約するという small-step の簡約を行うことが多い。たとえば、$((\lambda f.\, \lambda x.\, f\, x)(\lambda y.\, y))(\lambda z.\, z)$ という項は、call-by-value かつ left-to-right で簡約すれば、以下のようなステップを経て normal form となる。

$$
\begin{aligned}
& ((\lambda f.\, \lambda x.\, f\, x)(\lambda y.\, y))(\lambda z.\, z) \\
\rightarrow\ & (\lambda x.\, (\lambda y.\, y)\, x)(\lambda z.\, z) \\
\rightarrow\ & (\lambda y.\, y)(\lambda z.\, z) \\
\rightarrow\ & (\lambda z.\, z)
\end{aligned}
$$

一方、インタプリタを用いてプログラムを実行する際は、big-step の簡約が行われ、最終結果が直接返される。しかし、プログラミングの初学者にとっては、small-step の実行がプログラムの挙動を理解するのに役立つことがある。初学者がつまづきやすい難関の一つに再帰関数が挙げられるが、その代表的な例として、階乗を計算する関数 `fac` を考えてみよう。

```
let rec fac n = if n = 0 then 1 else n * fac (n - 1)
```

この関数をいろいろな引数で実行し、インタプリタが出力する答えが正しいことを観察するだけでは、計算過程を理解することは難しい。そこで、実行のステップをひとつずつ書き下してみる。

```
  fac 3
→ if 3 = 0 then 1 else 3 * fac 2
→ 3 * fac 2
→ 3 * (if 2 = 0 then 1 else 2 * fac 1)
→ 3 * (2 * fac 1)
```

```
→ 3 * (2 * (if 1 = 0 then 1 else 1 * fac 0))
→ 3 * (2 * (1 * fac 0))
→ 3 * (2 * (1 * (if 0 = 0 then 1 else 0 * (fac (-1)))))
→ 3 * (2 * (1 * 1))
→ 3 * (2 * 1)
→ 3 * 2
→ 6
```

このように small-step で考えると、4 回目に `fac` が呼び出された際に 1 が返り、それに 1, 2, 3 が順番にかけられて、期待する答えを求められていることが分かる。

次に、同じ計算をするための関数を、継続渡し形式で定義することを考える。継続渡し形式の `fac` は、正しくは以下のように定義される。

```
let rec fac2 n k = if n = 0 then k 1
                   else fac2 (n - 1) (fun x -> k (n * x))
```

ここで、ユーザがこのプログラムを意図しながら、次のように定義したとする。

```
let rec fac2 n k = if n = 0 then 1
                   else fac2 (n - 1) (fun x -> k (n * x))
```

この関数は、初期継続として恒等関数 `id` を渡した場合、どんな自然数の入力に対しても 1 を返す。その理由は、以下のように書き下すと明らかになる。

```
  fac2 3 id
→ if 3 = 0 then 1 else fac2 2 (fun x -> id (3 * x))
→ fac2 2 (fun x -> id (3 * x))
→ if 2 = 0 then 1 else fac2 1 (fun x -> id (3 * (2 * x)))
→ fac2 1 (fun x -> id (3 * (2 * x)))
→ if 1 = 0 then 1 else fac2 0 (fun x -> id (3 * (2 * (1 * x))))
→ fac2 0 (fun x -> id (3 * (2 * (1 * x))))
→ if 0 = 0 then 1 else fac2 (-1) (fun x -> id (3 * (2 * (1 * (0 * x)))))
→ 1
```

期待された答えが求まらないのは、与えられた引数が 0 だった場合に、その後の計算 `k`（この例では、1 をかけ、2 をかけ、3 をかけるという計算）を捨てて、直接 1 を返してしまっているためである。

このように、プログラムの実行を 1 ステップずつ書き出すことは、その挙動を理解したり、デバッグをしたりするのに有効である。しかし、多くのプログラマにとって、この作業は退屈なものである。そこで役立つのが stepper とよばれるものである。stepper とは、プログラムの実行を 1 ステップずつ止めて、その都度表示するためのツールである。初学者に限らず、上級者にとっても助けとなる stepper だが、現時点で stepper を提供している言語は Racket のみである [3]。

本研究では、OchaCaml [8] を用いて、OCaml の基本的な構文、および限定継続命令 `shift/reset` [4] をサポートする stepper の実装を試みる。以下、 `shift/reset` の解説をし（2 節）、これらを用いてどのようにステップ実行を行うかを述べる（3 節）。stepper 本体については、まず λ 計算に対応するものを実装し、その導き方と正しさについて議論する（4 節）。その後、対象言語を拡張し（5 節）、stepper を用いたプログラムの実行例を示す（6 節）。最後に、Racket の stepper との比較を行い（7 節）、まとめと今後の課題について述べる（8 節）。

2 限定継続

2.1 継続とは

継続とは、ある時点における残りの計算のことである。たとえば、$1 + (2 * 3) - 4$ というプログラムの $2 * 3$ の部分を評価しているときの継続は、$2 * 3$ が hole となったような計算、つまり「現在計算している部分の結果が返ってきたら、それに 1 を足し、4 を引く」という計算である。これは、$\lambda x.(1 + x - 4)$ という関数ととらえることができる。

2.2 `shift/reset`

一般に「継続」というと、その後の計算全体のことを指す。一方で、その後の計算のうち、一部のみを扱いたい場合は、限定継続命令を使うことができる。本研究では、限定継続を処理するための命令として `shift/reset` [4] を採用する。`shift` は継続を切り取る命令、`reset` は `shift` が切り取る継続の範囲を定める命令である。`shift/reset` を使うことのできる言語として、OchaCaml [8] がある。OchaCaml は Caml Light [7] に `shift/reset` を直接実装した言語である。OchaCaml において、これらの命令はそれぞれ `shift (fun k -> M)` および `reset (fun () -> M)` という構文で表される。以下は、`1 + reset (fun () -> shift (fun k -> k (k (2 * 3))) - 4)` というプログラムを OchaCaml インタプリタで実行した様子である。

```
# 1 + reset (fun () -> shift (fun k -> k (k (2 * 3))) - 4) ;;
- :  int = -1
```

`shift` は、`reset` に囲まれた計算で、自身が hole となった計算を `k` に束縛する。この場合、`k` は `reset (fun () -> [ . ]  - 4)` という計算となる[1]。「1 を足す」という部分が含まれていないのは、それが `reset` の外側にあるためである。この `k` が $2 * 3$ に 2 回適用され、最後に 1 が足されて、結果は -1 となる。

3 memo 関数

本研究で実装する stepper は、プログラムを受け取ったら、その計算の過程を全て入れたリストを返すという方針を取る。このために必要な動作は、

(1) redex を見つけ、

(2) その redex を簡約する前のプログラム全体を現在のステップとして記録し、

(3) redex の部分を簡約したものを返す

という流れの繰り返しである。このうち、(2) において、注目している redex だけでなく、そのまわりを取り囲む表現までを含めたプログラム全体を記録する必要がある。同様に、(3) においても、redex の部分を、その簡約結果に置き換えたプログラム全体を返したい。このように、redex あるいはその簡約結果と、それを取り囲む表現からもとのプログラム全体を組み立てることを、本稿では「プログラムを再構成する」という。プログラムの再構成において必要となるのは、もとのプログラムの中で redex だった部分が hole となったものである。一方、2 節で述べたように、`shift` 命令を使うと、`reset` に囲まれた計算で、自身が hole となったようなものを取り出すことができる。そこで、本研究では `shift` 命令を使ってプログラムを再構成する。以下の `memo` は、この考えに基づいて定義された OchaCaml のプログラムである。

[1] `shift` が切り取る継続には `reset` が含まれる。5.3 節参照。

```
let memo e f = shift (fun k -> fun lst ->
                  (reset (fun () -> k (f ()))) (k e lst))
```

memo は名前の通り、現在のステップを記録するための関数である。この関数は redex e と簡約後の計算 f を thunk の形で受け取り、shift 命令を実行する。shift 節の中では、それまでの計算のステップが逆順に入ったリスト lst を受け取っており、コンテキストが高階になっている。これは、shift/reset による state monad の実装となっている [6]。次節で詳しく述べるが、stepper 本体は、部分表現の評価中に値以外の表現が返ってきたら、評価文脈を再構成するように作られている。そのため、shift 命令で捕捉される継続 k は、通過した評価文脈、つまり注目している redex を取り囲む表現であり、関数適用 k e によって、現在のプログラムが再構成される。さらに lst を適用すると、lst に現在のプログラムが加わったものが得られる。これは、stepper 本体の関数が 4.1 節に示す go を使って呼び出されるためである。これを reset の外側から渡すことで、現在の状態が新しい状態に更新される。このようにして状態を更新したうえで、簡約規則を適用したあとの計算 f を実行する。新しい状態を作る際に、shift で捕捉した継続を適用している点が、通常の state monad と異なっている。

4 λ 計算の stepper

本節では、λ 計算を対象とした stepper の実装と、その性質について述べる。

4.1 実装

図 1 に対象言語の構文、評価文脈、および簡約規則を示した。式 e は、値 v か関数適用 $e_1\ e_2$ のいずれかであり、値 v は変数 x か λ 抽象 $\lambda x.\, e$ である。評価文脈は call-by-value かつ left-to-right の評価に対応している。また、redex $(\lambda x.\, e)\, v$ は、e 中に現れる x を v で置き換えたものに簡約される。

$$
\begin{array}{rcll}
e & ::= & v \mid e_1\ e_2 & \text{(式)} \\
v & ::= & x \mid \lambda x.\, e & \text{(値)} \\
E & ::= & E\ e \mid v\ E \mid [\,] & \text{(評価文脈)} \\
E[(\lambda x.\, e)\, v] & \rightarrow & E[e[v/x]] & \text{(簡約規則)}
\end{array}
$$

図 1. λ 計算の構文、評価文脈、簡約規則

ここで、1 節で取り上げた $((\lambda f.\, \lambda x.\, f\, x)(\lambda y.\, y))(\lambda z.\, z)$ のステップ実行を考えてみよう。この表現全体は関数適用の形なので、評価文脈 $E\ e$ に従って、まずは関数部分 $(\lambda f.\, \lambda x.\, f\, x)(\lambda y.\, y)$ を評価する。これも関数適用の形なので、再び評価文脈に従って、$\lambda f.\, \lambda x.\, f\, x$ を評価する。これは λ 抽象、すなわち値なので、今度は評価文脈 $v\ E$ に従って、引数 $\lambda y.\, y$ を評価する。これも値になったので、現在注目している部分が redex となる。今は実行のステップをすべて記録したいため、この redex を簡約する前に、現在のプログラムを状態に追加しなければならない。そのために、部分表現の評価中に値ではない表現を見つけたら、プログラムを再構成する必要がある。この場合は、redex $(\lambda f.\, \lambda x.\, f\, x)(\lambda y.\, y)$ を $\lambda z.\, z$ に適用する、という表現を再構成し、それをリストに追加したうえで、簡約規則を適用する。すると、1 ステップ簡約した表現 $(\lambda x.\, (\lambda y.\, y)\ x)(\lambda z.\, z)$ が得られる。この表現も redex の形であるので、再びリストに追加し、簡約規則を適用して $(\lambda y.\, y)(\lambda z.\, z)$ の形にする。これを同じように記録したうえで簡約すると、$\lambda z.\, z$ という normal form になり、これを最後にリストに加えることで、簡約のステップがすべて入ったリストが得られる。このように考えると、ステップ実行は、

(1) 評価文脈に従って redex を探す

(2) redex を見つけたら、現在のプログラムをリストに追加する

(3) 部分表現の評価結果が値でなければ、プログラムを再構成する

という原則に従うことで実現できる。この 3 つの役割を持つのが、以下に示す stepper 関数である。なお、`subst body x a2` は `body` における `x` の自由な出現を `a2` に置き換えることを意味する。

```
let rec stepper e = match e with
  Value (Lam (x, body)) -> e
| Value (Var (x)) -> e
| App (f, arg) ->
  let a1 = stepper f in
    begin match a1 with
      Value (Lam (x, body)) ->
        let a2 = stepper arg in
          begin match a2 with
            Value (Lam (x2, body2)) ->
              memo (App (a1, a2)) (fun () ->
              stepper (subst body x a2))
          | _ -> App (a1, a2)
          end
    | _ -> App (a1, arg)
    end
```

`stepper` に渡された表現が値、すなわち変数または λ 抽象であったら、それ以上計算する必要はないので、そのまま返す。関数適用の場合は、評価文脈にしたがって、まず関数部分を評価する。その結果 `a1` が λ 抽象であれば、引数を評価する。その結果 `a2` が値であれば、簡約規則 $E[(\lambda x.\, e)\, v] \to E[e[v/x]]$ を適用できる形になったので、もとのプログラム中の `App (f, arg)` を `App (a1, a2)` に置き換えたものをリストに追加したうえで、簡約規則を適用し、その結果に関して再帰呼び出しを行う。簡約を行ったあとに `stepper` をもう一度呼び出しているのは、関数の本体部分（ここでは `body`）が関数適用の形である場合に、`App (a1, a2)` を簡約した結果が再び redex となることがあるからである。たとえば、ここで `stepper` をもう一度呼び出さずに簡約結果を返すようにすると、先ほどの表現は $(\lambda y.\, y)\, (\lambda z.\, z)$ になった時点で実行が終了してしまう。

一方、引数を評価した結果 `a2` が λ 抽象でなかった場合は、関数適用の形が再構成される（変数の場合を考えていないのは、閉じた項であれば変数で関数を適用することはないからである）。これは、評価文脈 $v\ E$ を再構成しているのに相当する。同様に、関数を評価した結果 `a1` が λ 抽象でなかった場合も、関数適用の形を返す。こちらは、評価文脈 $E\ e$ の再構成に対応している。これらのケースを設けることによって、`memo` が簡約前の redex を受け取ったときに、プログラムが再構成されるようになる。通常の（big-step の）インタプリタを実装する際には、関数部分と引数部分の再帰呼び出しで必ず値が返るので、これらのケースは必要ない。

`stepper` の実行には、以下の関数を使う。

```
let go e = reset (fun () ->
                    let res = stepper e in
                      fun lst -> res :: lst) []
```

`go` は、初期状態として空リストを `reset` の外側で渡す。`stepper` の中で呼び出された `memo` は、このリストに現在のプログラムを追加していく。`memo` が `reset` の内側からこのリストにアクセス

できるのは、shift で捕捉した継続を fun lst -> ... という関数で包み込んでいるからである。

この stepper を用いて先ほどの表現を実行すると、評価文脈にしたがって、$(\lambda f.\,\lambda x.\,f\,x)(\lambda y.\,y)$、$\lambda f.\,\lambda x.\,f\,x$ とプログラムの中に入って行く。これがそのまま値として返され、引数 $\lambda y.\,y$ も値として返されると、redex の形となるので、$(\lambda f.\,\lambda x.\,f\,x)(\lambda y.\,y)$ を memo に渡す。このときに shift 命令で捕捉される継続は、次のような計算である。

```
reset (fun () ->
        let res =
          let a1 = [ . ] in
            match a1 with
              Value (Lam (x, body)) -> ...
            | _ -> App (a1, λz.z)
        in res :: [])
```

hole に $(\lambda f.\,\lambda x.\,f\,x)(\lambda y.\,y)$ を代入すると、簡約規則が適用される前の $((\lambda f.\,\lambda x.\,f\,x)(\lambda y.\,y))(\lambda z.\,z)$ が再構成され、これが実行時に渡された空リストに加えられる。簡約規則を適用すると、もとの表現の関数部分が $(\lambda x.\,(\lambda y.\,y)\;x)$ という λ 抽象になり、$\lambda z.\,z$ を評価する。ここで再び memo が呼び出され、$(\lambda x.\,(\lambda y.\,y)\;x)\,(\lambda z.\,z)$ がリストに加えられたうえで関数適用が行われる。さらに、簡約した結果 $(\lambda y.\,y)(\lambda z.\,z)$ が自身の評価の中でメモされ、その簡約結果 $\lambda z.\,z$ が最後にリストに加わる。このような過程を経て、$[\lambda z.\,z;\,(\lambda y.\,y)(\lambda z.\,z);\,(\lambda x.\,(\lambda y.\,y)\;x)(\lambda z.\,z);\,((\lambda f.\,\lambda x.\,f\,x)(\lambda y.\,y))(\lambda z.\,z)]$ というリストが得られる。

4.2 stepper の導出

4.1 節の実装は、以下のように一般化することができる。

(1) 受け取った表現が値ならばそのまま返す。

(2) 値でなければ、評価文脈に従って部分表現を評価する。

(3) それらの結果をパターンマッチし、簡約規則が適用できる形になったら、memo の第一引数にその redex を渡し、第二引数に簡約規則を適用したあとの計算を thunk にして渡す。

(4) memo の中で現在のプログラムを再構成するために、評価中に値でない表現が返ってきた場合を設け、現在の評価文脈を再構成するコードを書く。

こうして見ると、stepper は評価文脈と簡約規則に忠実に従って定義されていることが分かる。よって、新たな構文を追加するときは、評価文脈と簡約規則を定義し、(1) – (4) の方針に従って stepper を拡張すれば良い。

4.3 stepper の性質

ここでは、λ 計算の stepper が以下の二つの性質を満たすことを示す。

(1) memo 関数を入れても、ステップ実行した結果がリストに入るのを除けば、プログラムの挙動は変わらない。

(2) あるプログラムを stepper で実行すると、その reduction sequence の要素が順にすべて得られる。

まず、(1) について考えよう。通常のインタプリタを使ってプログラムを実行する際は、redex が見つかったら、簡約規則を適用したあとの計算が直ちに行われる。一方、stepper は、redex を見つけたら memo を呼び出し、現在のプログラムを記録したうえで、第二引数の thunk に unit を渡

し、簡約後の計算を行う。つまり、stepper は通常のインタプリタに「redex を見つけたら、現在のプログラムをリストに追加する」というはたらきを加えたものである。したがって、stepper の中に現れる memo e (fun () -> e') を e' に置き換えると、stepper は通常のインタプリタと同一になり、これを用いてプログラムを実行すると、最後に normal form となった項一つからなるリストが得られる。つまり、あるプログラム e が v に評価されるならば、go はリスト $[v]$ を返す。

つづいて、(2) について考える。簡約のステップがすべてリストに入っていることを示すためには、簡約が起こるたびに、その時点の表現がリストに追加されることを保証しなければならない。λ 計算の中で、簡約が行われるのは関数適用のケースで、関数部分が λ 抽象、引数部分が値となったときのみである。4.1 節のプログラムを見ると、redex が見つかったら、memo に簡約前の表現を渡すことでその時点におけるプログラムを再構成し、リストに追加している。また、stepper が最終的に返す結果は、最後にリストに加わる。そのため、リストには簡約ステップ数 + 1 だけ要素が加わることになる。

次に、リストに加えられる表現が必ず正しく再構成されることを示す。プログラムの再構成が行われるのは、関数適用のケースのみである。その中で、関数部分について再帰呼び出しをしているとき、つまり現在の評価文脈が $E\ e$ であるときは、memo に redex が渡されて、stepper が値ではない表現を返したら、その redex を e に適用するという評価文脈が再構成される。同様に、引数を評価しているとき、すなわち現在の評価文脈が $v\ E$ であるときは、memo に redex が渡されると、その redex で v を適用するという評価文脈が再構成される。したがって、ある入力 e が $E[e']$ に分解され、stepper が e' の評価中に値ではない表現 e'' を返したならば、それは $E[e'']$ に再構成されることが言える。

最後に、reduction sequence の各要素が、その順でリストに加わることを示す。stepper は、評価文脈の定義に従って、左から右に redex を探していくので、各々のステップにおける表現は、逆順ではあるが、もとの sequence の順番通りにリストに加えられる。

以上より、あるプログラムを stepper で実行すると、各ステップにおける表現が順にすべて入ったリストが得られる。つまり、次の命題が成り立つ。

プログラム e が、$e = e_1 \to e_2 \to ... \to e_n = v$ という reduction sequence をもつとき、e を stepper で実行した結果のリストには、e_1, e_2, ..., e_n が逆順に入っている。すなわち、$go\ e = [e_n; ...; e_2; e_1]$ となる。

5 拡張した stepper

本節では、対象言語を図 2 に示す構文に拡張する。

$$
\begin{array}{rcl}
e & ::= & v \mid e\ e \mid e\ nop\ e \mid e\ cop\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{let rec}\ f\ x = e\ \texttt{in}\ e \\
& \mid & e :: e \mid (e, e) \mid \texttt{match}\ e\ \texttt{with}\ [\,] \to e \mid f :: r \to e \\
& \mid & \texttt{shift}\,(\lambda c.\, e) \mid \texttt{reset}\,(e) \\
v & ::= & n \mid b \mid x \mid \lambda x.\, e \mid \texttt{fix}\, f.\, x.\, e \mid [\,] \mid v :: v \mid (v, v) \\
nop & ::= & + \mid - \mid * \mid / \quad \text{（四則演算）} \\
cop & ::= & = \mid < \mid > \quad \text{（比較演算）}
\end{array}
$$

図 2. 拡張した対象言語の構文

これらの評価文脈と簡約規則を図 3 - 4 に示す。なお、`shift/reset` に関する定義は、[2] に従う [2]。F は hole を `reset` で囲んでいない文脈であり、pure な文脈とよばれる。

$$
\begin{array}{rcl}
E & ::= & [\,] \mid E\ e \mid v\ E \\
& \mid & E\ \mathit{nop}\ e \mid v\ \mathit{nop}\ E \mid E\ \mathit{cop}\ e \mid v\ \mathit{cop}\ E \\
& \mid & \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ x = E\ \texttt{in}\ e \\
& \mid & E :: e \mid v :: E \mid (E,\, e) \mid (v,\, E) \\
& \mid & \texttt{match}\ E\ \texttt{with}\ [\,] \to e \mid f :: r \to e \\
& \mid & \texttt{reset}\,(E) \\
F & ::= & [\,] \mid F\ e \mid v\ F \\
& \mid & F\ \mathit{nop}\ e \mid v\ \mathit{nop}\ F \mid F\ \mathit{cop}\ e \mid v\ \mathit{cop}\ F \\
& \mid & \texttt{if}\ F\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ x = F\ \texttt{in}\ e \\
& \mid & F :: e \mid v :: F \mid (F,\, e) \mid (v,\, F) \\
& \mid & \texttt{match}\ F\ \texttt{with}\ [\,] \to e \mid f :: r \to e
\end{array}
$$

図 3. 評価文脈

$$
\begin{array}{rcl}
E[(\lambda x.\, e)\, v] & \to & E[e[v/x]] \\
E[(\texttt{fix}\, f.\, x.\, e)\, v] & \to & E[e[\texttt{fix}\, f.\, x.\, e/f,\ v/x]] \\
E[v_1\ \texttt{Plus}\ v_2] & \to & E[v_1 + v_2] \\
 & & (\text{ただし、} v_1, v_2 \text{は整数。他の演算も同様。}) \\
E[\texttt{if true then}\ e_1\ \texttt{else}\ e_2] & \to & E[e_1] \\
E[\texttt{if false then}\ e_1\ \texttt{else}\ e_2] & \to & E[e_2] \\
E[\texttt{let}\ x = v\ \texttt{in}\ e] & \to & E[e[v/x]] \\
E[\texttt{let rec}\ f\ x = e_1\ \texttt{in}\ e_2] & \to & E[e_2[\texttt{fix}\, f.\, x.\, e_1/f]] \\
 & & (\text{ただし、} f \text{は今までに定義されていない。}) \\
E[\texttt{match}\ [\,]\ \texttt{with}\ [\,]\ \texttt{->}\ e_1 \mid f :: r\ \texttt{->}\ e_2] & \to & E[e_1] \\
E[\texttt{match}\ v_1 :: v_2\ \texttt{with}\ [\,]\ \texttt{->}\ e_1 \mid f :: r\ \texttt{->}\ e_2] & \to & E[e_2[v_1/f,\, v_2/r]] \\
E[\texttt{reset}\,(F[\texttt{shift}\,(\lambda c.\, e)])] & \to & E[\texttt{reset}\,(e[\lambda x.\, \texttt{reset}\, F[x]/c])] \\
E[\texttt{reset}\,(v)] & \to & E[v]
\end{array}
$$

図 4. 簡約規則

5.3 節で詳しく述べるが、簡約規則中の v は変数以外の値とする。また、`let rec` の簡約規則の注については、5.2 節で触れる。

拡張した対象言語は、`shift` と `reset` を含んでいる。これらの実行にメタ言語の `shift/reset` を使ってしまうと、`memo` の定義で使用しているメタ言語の `shift` と互いに干渉してしまう。そのため、`stepper` を継続渡し形式 [9] で定義した。よって、`stepper` の定義は以下のようになる。

[2] `shift/reset` の簡約規則としては、`shift` によって構文木における直近の親ノードまでの継続を取ってくるという方針 [1] もあるが、ここでは直近の `reset` までの継続を一度に取り出したいため、[2] の定義を採用する。

```
let rec stepper e k = match e with
...
```

これに伴い、go の中で stepper を呼び出すときは、初期継続として空の継続（恒等関数）を渡している。

拡張後の stepper も、λ 計算のときと同じように、評価文脈に従って再帰呼び出しを行い、簡約規則を適用する直前に現在のプログラムを記録し、値以外のものが返ってきた場合は表現を再構成するという方針で実装する。以下、四則演算、let rec、shift および reset の実装について、順次見ていく。

5.1 四則演算

四則演算の例として、足し算のステップ実行を見てみる。なお、NOp は四則演算を表すコンストラクタである。

```
NOp (n1, Plus, n2) ->
  stepper n1 (fun a1 ->
    begin match a1 with
      Value (Num (v1)) ->
        stepper n2 (fun a2 ->
          begin match a2 with
            Value (Num (v2)) ->
              memo (k (NOp (a1, Plus, a2))) (fun () ->
              k (Value (Num (v1 + v2)))
          | _ -> k (NOp (a1, Plus, a2))))
          end
    | _ -> k (NOp (a1, Plus, n2)))
    end
```

8 ページに示した評価文脈 $E\ nop\ e$ に従い、まずは n1 を評価する。通常の CPS インタプリタのように、n1 の計算結果は a1 として自身の継続に渡される。a1 が数字になったら、今度は評価文脈 $v\ nop\ E$ に従って、n2 を評価する。その結果 a2 も数字になったら、簡約規則 $E[v_1\ \texttt{Plus}\ v_2] \to E[v_1 + v_2]$ が適用できるので、memo を呼び出して現在のプログラムをリストに追加し、最終的に返したい答え、すなわちメタ言語の足し算を行った結果を k に渡す。k はその表現を取り囲む直近の Reset までの継続である。したがって、ある表現を k に渡すことで、直近の Reset までのプログラムが再構成される。それより外側の継続は、memo 関数の中で呼び出される shift 命令によって捕捉される。リストにプログラムを追加する際は、この継続を適用することで、全体を再構成している。一方、n2 の評価中に数字以外の表現が返ってきた場合は、評価文脈 $v + E$ を再構成する。同様に、n1 の評価で数字以外のものが返されたら、評価文脈 $E + e$ を再構成する。

5.2 let rec

再帰関数を入力する際の構文としては、多くのユーザにとって使いやすい let rec を採用しているが、内部においては新たに追加した値 fix を用いて再帰関数を表現している。また、プログラム中に現れる再帰関数の名前、パラメータおよび定義の組を入れるためのリスト rec_fun を用意した。ステップ実行の結果を表示する際には、最初にプログラム中の関数定義を let rec の形ですべて表示し、それらを呼び出すときは、関数名（fix $f.x.e$ の f）のみを表示する形にした。

```
LetRec (name, x, e1, e2) ->
  if (not_contain name !rec_fun)
  then (rec_fun := (name, x, e1) :: !rec_fun;
        stepper (subst e2 name (Value (Fix (name, x, e1)))) k)
  else (let f = gensym () in
        rec_fun := (f, x, subst e1 name (Value (Var (f)))) :: !rec_fun;
        stepper (subst e2 name
                 (Value (Fix (f, x, subst e1 name (Value (Var (f))))))) k)
```

関数定義 `LetRec (name, e1, e2)` を見つけたら、同じ名前の関数がすでに定義されていないかをチェックする。名前の衝突がなければ、簡約規則 $E[\texttt{let rec}\ f\ x = e_1\ \texttt{in}\ e_2] \to E[e_2[\texttt{fix}\, f.\, x.\, e_1/f]]$ が適用できるので、`e2` の中の変数 `name` を `fix name. x. e1` に置き換えたものを評価する。もし同じ名前の関数が `rec_fun` に入っていたら、新しい名前に付け替えたうえで、`e2` を評価する。最終的に再帰関数の定義をすべて表示しているため、このケースでは `memo` を呼び出さない。

5.3 shift

`shift` のステップ実行は以下のように行われる。

```
Shift (c, m) ->
  memo (k (Shift (c, m))) (fun () ->
  let x = gensym () in
    stepper (subst m c (Value (Lam (x, Reset (k (Value (Var (x))))))))
            (fun a -> a))
```

`Shift (c, m)` については簡約規則 $E[\texttt{reset}\,(F[\texttt{shift}\,(\lambda c.\, e)])] \to E[\texttt{reset}\,(e[\lambda x.\, \texttt{reset}\, F[x]/c])]$（$F$ は pure な文脈）が適用できるので、現在のプログラムをリストに加えたうえで、`m` 中の変数 `c` を「何か引数を渡されたら、それに現在（`shift` が呼び出された時点）の継続を適用する」という関数に置き換え、空の継続で評価する。前述の通り、`k` は直近の `reset` までの計算なので、`k (Value (Var (x)))` は簡約規則の中の $F[x]$ に相当する。

ここで、いまは call-by-value で評価を行うことにしているため、λ 抽象の内側にある `k (Value (Var (x)))` という計算の中で、簡約規則が適用されるのは望ましくない。これを防ぐために、図 4 の簡約規則を適用できるのは、各規則に含まれる v が変数以外のときのみとし、v が変数である場合は、簡約を行わないようにした。`go` に渡されたプログラムが閉じた項であれば、`stepper` が変数を返すことはないため、このようなパターンにマッチし得るのは、`Shift` の本体を評価するときのみである。

なお、`k (Value (Var (x)))` を取り囲む `Reset` は、`shift` で切り取られる継続が非明示的な `reset` で囲まれることを表している。また、簡約規則の右辺を取り囲む `reset` は、次に述べる `reset` の実装の中で、本体を評価した結果が値以外だったときに表現を再構成することで対応している。

5.4 reset

`reset` は以下のように定義した。

```
Reset (m) ->
  let a = stepper m (fun a' -> a') in
    begin match a with
      Value (Var (x)) ->
        k (Reset (a))
```

```
| Value (v) ->
    memo (k (Reset (a))) (fun () ->
    k (Value (v)))
| _ -> k (Reset (a))
end
```

評価文脈 reset (E) に従い、本体 m を評価する。このときに渡される継続は、空の継続（恒等関数）である。これによって、本体に対する継続が文字通り“リセット”されるため、中で shift が呼び出されたときに切り取られる継続は、reset に囲まれた計算に限定される。本体を評価した結果が変数以外の値であれば、簡約規則 $E[\texttt{reset}\,(v)] \to E[v]$ が適用できるので、現在のプログラムをリストに追加し、自身を値に置き換えた表現を構成する。それ以外の場合は、Reset の形を再構成する。本体を評価した結果をパターンマッチする際にシステムのスタックを使用しているため、この部分の実装は 4.1 節のものに近い形となっている。

5.5 拡張した stepper の性質

前述の通り、拡張した stepper も、（let rec を除いて）簡約可能な項を見つけたら簡約前のプログラムをリストに追加し、値以外の表現が返されたら評価文脈を再構成するように実装されている。そのため、4.3 節で示した二つの命題は、拡張後の stepper についても成り立つ。

6 実行例

6.1 factorial

1 節に挙げた fac を stepper で実行すると、次のような結果となる。

```
go fac ;;
["let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 6";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * 2)";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * 1))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (1 * 1)))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (1 * (if true then 1 else (0 * (fac (0 - 1))))))))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (1 * (if (0 = 0) then 1 else (0 * (fac (0 - 1))))))))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (1 * (fac 0))))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (1 * (fac (1 - 1)))))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (if false then 1 else (1 * (fac (1 - 1)))))))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (if (1 = 0) then 1 else (1 * (fac (1 - 1)))))))";
```

```
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (fac 1)))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (2 * (fac (2 - 1))))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (if false then 1 else (2 * (fac (2 - 1)))))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (if (2 = 0) then 1 else (2 * (fac (2 - 1)))))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (fac 2))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (3 * (fac (3 - 1)))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (if false then 1 else (3 * (fac (3 - 1))))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (if (3 = 0) then 1 else (3 * (fac (3 - 1))))";
 "let rec fac = (λn.(if (n = 0) then 1 else (n * (fac (n - 1))))) in
 (fac 3)"]
```

図 5. fac の実行

リストを後ろから見ていくと、それぞれの簡約ステップにおいて再構成されたプログラムが一つずつ入っていて、最終的に期待された計算結果が得られていることが分かる。なお、ここで let rec fac = ... を毎回表示しているのは、この定義がないと、関数を展開したときに else 節の fac が自由変数となってしまうからである。Racket の stepper も、このようにプログラムの先頭に関数定義列を残した状態で、ステップ実行を行っている。

6.2 times

times は、リスト中の要素をかけ合わせた値を返す関数である。この関数は、渡されたリストの中に一つでも 0 が含まれていたら、結果として 0 を返す。そこで、0 が見つかった時点で、残りの計算を行うことなく、直接 0 を返すようにしたい。shift/reset を使わない場合、そのプログラムは以下のように書かれる。

```
let rec times lst = match lst with
  [] -> 1
| first :: rest -> if first = 0 then 0
                   else first * times rest
in times [1; 0; 2]
```

このように書くと、リストの先頭要素が 0 であったら、残りの要素を見ることなく 0 が返される。しかし、これでは「0 よりも前に入っていた要素をかけ合わせる」という計算が行われてしまう。この計算も行わないようなプログラムは、shift 命令を使って書くことができる。

```
let rec times2 lst = match lst with
  [] -> 1
| first :: rest -> if first = 0 then shift (fun k -> 0)
                   else first * times2 rest
in reset (times2 [1; 0; 2])
```

lst の先頭要素が 0 だった場合、shift 命令が呼び出され、その時点の継続が変数 k に束縛される。この継続は適用されないまま破棄されるため、0 の前に入っていた要素をかける計算は行われない。限定継続に慣れていないプログラマにとって、ここで単に 0 を返した場合との違いをすぐに理解するのは難しいが、両者に同じリストを与えて stepper で実行した結果を見ると、その違いが明らかになる（ただし、見やすいように適宜改行している）。

```
go times ;;
["let rec times = (λlst.(match lst with
    [] -> 1
  | (f :: r) -> (if (f = 0) then 0 else (f * (times r))))) in 0";
 "let rec times = (λlst.(match lst with
    [] -> 1
  | (f :: r) -> (if (f = 0) then 0 else (f * (times r))))) in
  (1 * 0)";     (* ここで 1 * 0 が行われている *)
 "let rec times = (λlst.(match lst with
    [] -> 1
  | (f :: r) -> (if (f = 0) then 0 else (f * (times r))))) in
  (1 * (if true then 0 else (0 * (times (2 :: []))))))";
 "let rec times = (λlst.(match lst with
    [] -> 1
  | (f :: r) -> (if (f = 0) then 0 else (f * (times r))))) in
  (1 * (if (0 = 0) then 0 else (0 * (times (2 :: []))))))";
  ...
 "let rec times = (λlst.(match lst with
    [] -> 1
  | (f :: r) -> (if (f = 0) then 0 else (f * (times r))))) in
  (times (1 :: (0 :: (2 :: []))))"]
```

図 6. times の実行

```
go times2 ;;
["let rec times2 = (λlst.(match lst with
    [] -> 1
  | (f :: r) -> (if (f = 0) then (shift (λk.0)) else (f * (times2 r))))) in
  0";
 "let rec times2 = (λlst.(match lst with
    [] -> 1
  | (f :: r) -> (if (f = 0) then (shift (λk.0)) else (f * (times2 r))))) in
  reset (0)";
 "let rec times2 = (λlst.(match lst with
    [] -> 1
  | (f :: r) -> (if (f = 0) then (shift (λk.0)) else (f * (times2 r))))) in
  reset (((λk.0) (λx1.reset ((1 * x1)))))";
  (* λk.0 は本体に k を含まないため、1 * [ . ] は捨てられる *)
 "let rec times2 = (λlst.(match lst with
    [] -> 1
```

```
 | (f :: r) -> (if (f = 0) then (shift (λk. 0)) else (f * (times2 r))))) in
 reset ((1 * (shift (λk. 0))))";
"let rec times2 = (λlst. (match lst with
   [] -> 1
 | (f :: r) -> (if (f = 0) then (shift (λk. 0)) else (f * (times2 r))))) in
 reset ((1 * (if true then (shift (λk. 0))
              else (0 * (times2 (2 :: [])))))))";
"let rec times2 = (λlst. (match lst with
   [] -> 1
 | (f :: r) -> (if (f = 0) then (shift (λk. 0)) else (f * (times2 r))))) in
 reset ((1 * (if (0 = 0) then (shift (λk. 0))
              else (0 * (times2 (2 :: [])))))))";
 ...
"let rec times2 = (λlst. (match lst with
   [] -> 1
 | (f :: r) -> (if (f = 0) then (shift (λk. 0)) else (f * (times2 r))))) in
 reset ((times2 (1 :: (0 :: (2 :: [])))))"]
```

図 7. `times2` の実行

`times` では、リスト `[1; 0; 2]` の二つ目の要素を見た時点で、残りの要素を見ることなく 0 を返しているが、1 をかけるという計算は行われている。一方、`times2` では、`shift` 命令によって 1 をかける計算が破棄されているため、0 が直接返される。

7 関連研究

Clements et al. [3] は、Scheme（現在の Racket）の stepper を実装している。彼らは、ある時点における実行の状態を取り出す命令をプログラムに挿入するだけで、ステップ実行を行うことができるということを主張している。この状態を取り出す命令を breakpoint とよび、Scheme に breakpoint を加えた高レベルの言語を source 言語とする。breakpoint によるステップ実行は、低レベルの target 言語によって実装されている。target 言語は、continuation mark というメカニズムに基づいており、構文としては、Scheme に with-continuation-mark、current-continuation-marks、および output の 3 つの命令を加えたものになっている。with-continuation-mark は通過した評価文脈などをスタックに記録する命令、current-continuation-marks はそれらを取り出す命令、output は取り出された評価文脈を出力する命令である。

continuation mark とは、スタックに付けたマークのことである。stepper では、各 continuation mark に、もとの表現を再構成するための情報を値として持たせている。たとえば、$e_1 + e_2$ という計算の e_1 を評価しているとき、もとの表現を再構成するためには、現在計算している部分が足し算の第一引数であり、第二引数が e_2 であるという情報が必要となる。これは、target 言語における評価文脈 $E + e_2$ を通過した、と言い換えることができる。この通過した評価文脈の情報が、continuation mark の値として記録される。これらの値は、対応する Scheme の表現を再構成するのに使われる。

ステップ実行を行う際には、まず Scheme のプログラムを source 言語に変換する関数によって、プログラム中の簡約可能な箇所すべてに breakpoint を挿入する。次に、annotation 関数 $\mathcal{A}$ によって、source 言語を target 言語にコンパイルする。この中で、値ではない部分表現は (with-continuation-marks < 現在の評価文脈 > < その後の計算 >) という形に変換される。 ここで、変換したい表現が

breakpoint であれば、current-continuation-marks を使ってそれぞれの continuation mark に入っている評価文脈を取り出し、output でそれらを出力する。出力された評価文脈は、breakpoint を取り除く関数と翻訳関数によって、Scheme の表現に再構成される。

彼らの stepper は、以下の二つの定理を満たすことが示されている。

定理 1 (Elaboration Theorem)
annotation 関数 $\mathcal{A}$ を適用する前後で、プログラムの挙動が変わらない。

定理 2 (Stepping Theorem)
stepper でプログラムを実行すると、reduction sequence の要素が順にすべて得られる。

定理 1 は、breakpoint に期待される動作を、continuation mark のメカニズムで実装できていることを表す。定理 2 は、彼らの実装した stepper が正しく動作することを示している。

ここで、Clements らの stepper と、本研究の stepper を比較してみよう。Clements らの主張は、breakpoint 命令を入れるだけでステップ実行ができる、というものである。そのような高レベルの言語を実装するために、低レベルの言語を使って continuation mark に通過した評価文脈を記録し、必要なときにそれらを取り出している。一方、本研究の stepper では、Clements らが breakpoint 命令を入れている位置に、memo 関数を挿入している。また、部分表現を評価した結果が値ではなかったケースを設けて、そこに評価文脈を再構成するコードを書き、それを memo 関数の中で取り出している。つまり、`shift/reset` をうまく使って memo 関数を実装し、その中でプログラムを正しく再構成できるようにコードを書くことで、stepper に期待される動作を実現している。Clements らが示した定理 1, 2 は、それぞれ 4.3 節で示した二つの性質と対応しているが、レベルの異なる二つの言語を用いている Clements らの stepper と比べると、本研究の stepper は、それが満たす性質を証明するのはそれほど難しくない。

プログラムの動きを理解するためのツールとしては、他にも Chez Scheme [5] 等の trace がある。これは、ユーザが指定した関数の引数と返り値を表示するものである。たとえば、関数 `fac` を trace に渡し、引数 3 で実行すると、以下のような結果が得られる。

```
> (define (fac n) (if (= n 0) 1 (* n (fac (- n 1)))))
> (trace fac)
(fac)
> (fac 3)
|(fac 3)
| (fac 2)
| |(fac 1)
| | (fac 0)
| | 1
| |1
| 2
|6
6
>
```

このように、trace はそれぞれの再帰呼び出しにおける引数と返り値を表示することで、ユーザがプログラムの挙動を理解したり、間違いを見つけたりするのを助ける。しかし、個々の返り値がどのような過程を経て計算されているのかを見ることはできない。この点が、stepper と根本的に異なっている。

8　まとめと今後の課題

本論文では、OCaml の基本的な構文および `shift/reset` に対応する stepper の実装について述べた。ステップ実行を行う際には、現在注目している redex と、そのまわりを取り囲む表現から、もとの表現全体を再構成する必要があるが、ここでは memo 関数の中でメタ言語の `shift` 命令を使って redex のまわりの表現を取り出す、という方針で実装した。また、stepper が満たすべき性質として、memo 関数を挿入しても、ステップ実行の結果がリストに加わる以外はプログラムの挙動が変わらないこと、および stepper を使ってプログラムを実行すると、簡約のステップがすべて逆順に入ったリストが得られることを示した。

今後の課題として、一つ目に構文の拡張が挙げられる。4 節で述べたように、stepper 本体の関数は評価文脈と簡約規則に従って実装されているため、新たな構文を追加する際は、適切な評価文脈と簡約規則を定義すれば、コードを書くのはそれほど難しくないと考えられる。二つ目の課題として、実行結果を表示する方法の改良が挙げられる。現在はステップ実行した結果を一つのリストに入れて返しているが、Racket の stepper のように、1 ステップずつ実行を止めて、現在注目している redex とその簡約結果をハイライトして表示するように改良を行う予定である。

謝辞 多くの有益なコメントを下さった査読者の皆様に深く感謝致します。

参考文献

[1] Z. M. Ariola, H. Herbelin, and A. Sabry. A Proof-Theoretic Foundation of Abortive Continuations. *Higher-Order and Symbolic Computation*, Vol. 20, No. 4, pp. 403–429, 2007.

[2] K. Asai, and Y. Kameyama. Polymorphic Delimited Continuations. In *5th Asian Symposium on Programming Lnaguages and Systems, APLAS 2007, Lecture Notes in Computer Science 4807*, pp. 239-254, 2007.

[3] J. Clements, M. Flatt, and M. Felleisen. Modeling an Algebraic Stepper. *Programming Languages and Systems*. Springer Berlin Heidelberg, pp. 320-334, 2001.

[4] O. Danvy, and A. Filinski. Abstracting Control. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and Functional Programming*, pp. 151-160, ACM, 1990.

[5] R. K. Dybvig. *Chez Scheme User's Guide*. Cadence Research Systems, 1998.

[6] A. Filinski. Representing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 446-457, ACM, 1994.

[7] X. Leroy. The Caml Light system release 0.74. *URL: http://caml.inria.fr*, 1997.

[8] M. Masuko, and K. Asai. Caml Light + shift/reset = Caml Shift. *Theory and Practice of Delimited Continuations (TPDC 2011)*, pp. 33-46, 2011.

[9] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1.2, pp. 125-159, 1975.