shift/reset プログラミング入門

浅井 健一

平成 23 年 9 月 23 日

概 要

継続の概念はどのプログラミング言語においても現れる普通の概念である。条件文は、条件に従ってふたつの継続（＝未来）からひとつを選ぶ操作だし、例外は継続の一部を捨てる操作と考えることができる。また、末尾呼び出し（goto 文）は継続に実行を移す操作である。このように、継続はどこにでも出てくる当たり前の概念だが、これまで継続を明示的に扱うのはあたかも難しいことかのように考えられて来た。

このチュートリアルでは、プログラムの中で継続を明示的に扱うというのがどのようなことなのかを限定継続命令 shift と reset を使って解説する。継続に関する基礎知識はゼロから始めて、最終的には簡単なコルーチンや非決定的な探索プログラムを書くところまでを目指す。継続は初めてでも CW 2011 における発表に興味があるという人には、継続へのよい導入になると期待している。

チュートリアルは、OCaml, Standard ML, Scheme, Haskell といった関数型言語の一般的な基礎知識を仮定するが、継続についての知識は一切、仮定しない。チュートリアルでは種々の練習問題を用意しており、参加者は各自のコンピュータを使ってその場でプログラミングを行う。

1 はじめに

継続とは「その後の計算」を表す概念である。例外処理を一般化したものととらえることもできるが、例外処理よりもはるかに強力である。複雑な計算を書こうと思うと、しばしば普通のブロック構造では書きにくい制御構造を扱わなくてはならなくなる。継続を操作できると多くの場合、そのようなプログラムを全体の見通しを損なうことなく書くことができるようになる。

継続を明示的に扱うには、プログラム全体を継続渡し形式 [13] に変換するという方法がある。しかし、この方法ではプログラム全体を継続渡し形式で書かなくてはならない。直接形式でプログラムを書きつつ継続を操作したい場合には何らかの継続を扱う命令が必要である。

継続を扱う命令で最も有名なのは Scheme や Standard ML の call/cc である。しかし、call/cc は「その後の計算」として未来永劫の計算を全て取って来るため使いにくい。実際、継続渡し形式でプログラムを書いたときの継続は、未来永劫の計算を指すのではなく、その一部のみを指す場合がほとんどである。このように範囲が限定されている継続のことを限定継続と呼ぶ。

限定継続を取って来る命令は Felleisen の `control/prompt` [6]、Danvy と Filinski の `shift/reset` [4]、Gunter, Rémy, Riecke の `cupto/prompt` [8] などがある。この中で、本稿では Danvy と Filinski による `shift/reset` を用いる。これは、`shift/reset` には健全かつ完全な公理系が存在すること [9]、多相の型システムが存在すること [2]、継続渡し形式のプログラムとの間に厳密な対応関係があるため使いやすく、各種の応用があることなどの理由による。

本稿では、`shift/reset` を使った種々のプログラミングを行い、限定継続を使ったプログラミングがどのようなものなのかを体得する。

1.1 実装

`shift/reset` を使うことのできる処理系としては以下があげられる。

- call/cc と変更可能なセルがある言語では、このふたつを使って `shift/reset` を模倣できることを Filinski が示している [5]。Scheme や Standard ML ではこの方法で `shift/reset` を使うことができる。ただし、answer type は固定となる。

- Gasbichler と Sperber は Scheme48 上で `shift/reset` を直接実装した [7]。この実装は call/cc を使った方法よりも効率が良いことが報告されている。
- Racket には、`shift/reset` を含む限定継続命令を用意している。
- Kiselyov は `shift/reset` を含む限定継続命令を Delimcc library として OCaml 用に実装している [10]。
- MinCaml を拡張して `shift/reset` を使えるようにした研究がある [11]。answer type が変化する多相の型システムがサポートされている。より汎用の言語に対する実装として Caml Light を `shift/reset` で拡張した OchaCaml [12] がある。

本稿では OchaCaml を使って各種の `shift/reset` プログラミングを行う。

1.2 本稿の構成

次の節で `shift/reset` プログラミングを基礎から説明する。より基礎的な理論を知りたい方のために、3 節では `shift/reset` の基礎理論を概観するが、より詳細を知りたい方は論文を参照されたい。

1.3 前提とする知識

本稿では、OCaml, Standard ML, Scheme, Haskell など関数型言語についての一般的な知識を仮定しているが、継続についての知識は仮定しない。また、3 節では、多相の let 文を含む型付き λ 計算、その実行規則と型システムについての一般的な知識を仮定している。

2 `shift/reset` プログラミング

2.1 継続とは

継続とは、一言でいうと「その後の計算」のことである。プログラムの実行は、プログラム中の一部を次に実行すべき式（redex と呼ばれる）として選び出して実行し、その結果を使ってその後の計算を進める。この「その後の計算」が継続である。従って、継続はどのようなプログラムの実行にも現れる基本的な概念である。

継続が何であるかを明示するため、現在、着目している部分、これから実行しようとしている部分を `[...]`（hole と呼ばれる）で表そう。例えば $3+5*2-1$ という式の $5*2$ をこれから実行しようとしているなら $3+[5*2]-1$ となる。このときの継続は $3+[\cdot]-1$ となる。つまり、`[·]` の値（$5*2$ の結果である 10）が得られたら、「その結果に 3 を加え、1 を引く」が継続である。継続は、「hole の値を受け取ったら、その後の計算を行う」という意味で関数と似たような概念である。

継続は「例外が発生したときに捨てられる部分」という理解もできる。例えば、上の例で `[·]` に `raise Abort` という式を入れて例外を発生させたとしよう。このときに捨てられる計算 $3+[\cdot]-1$ がそのときの継続である。

継続というのは、現在、どこに着目しているかによって時々刻々と変わっていく。例えば、$5*2$ の計算が終了し、プログラムが $3+10-1$ になったとしよう。このプログラムは、次に $3+10$ を実行するので $[3+10]-1$ と表現できる。したがって、このときの継続は `[·]` -1、つまり「結果から 1 を引く」である。$3+10$ の計算が終了するとプログラムは $13-1$ になる。この時点で次に実行するべき式は $13-1$ なので $[13-1]$ と表現される。この時点での継続は空の継続 `[·]` つまり「何もせずに値をそのまま返す」である。

練習問題 1 次のプログラムに `[·]` を入れて、「次に実行すべき式」と「その継続」に分解せよ。前者の型は何か。その型の値を渡されたら、継続は最終的に何型の値を返すか。

(1) `5 * (2 * 3 + 3 * 4)`

(2) `(if 2 = 3 then "hello" else "hi") ^ " world"`

(3) `fst (let x = 1 + 2 in (x, x))`

(4) `string_length ("x" ^ string_of_int (3 + 1))`

2.2 限定継続とは

継続は「その後の計算」だが、限定継続というのは「その後の計算」の中でもその範囲が定まっている（限定されている）もののことである。$3 + [5 * 2] - 1$ の場合、現在の継続は $[\cdot]$ の外側の計算すべて、つまり $3 + [\cdot] - 1$ になるが、限定継続と言った場合にはその一部、ある区切られた範囲内の継続のみになる。

どこまでで継続を区切るかは限定子 $\langle \cdots \rangle$ によって指定する。例えば $\langle 3 + [5 * 2] \rangle - 1$ のように限定したなら、現在の継続は $\langle 3 + [\cdot] \rangle$ となり -1 は含まれない。

2.3 継続を限定する命令 reset

継続を区切るには、reset という命令を使う。本稿では OchaCaml の文法にあわせて

`reset (fun () -> `M`)`

という構文を使う。この文の意味は M と同じだが、M を実行中の継続はこの reset までで区切られる。

例えば、

```
reset (fun () -> 3 + [5 * 2]) - 1
```

という式を考えてみよう。次に実行するのは $5 * 2$ だが、その時点での限定継続は「結果に 3 を加える」のみであり「1 を引く」は含まれない。この例では $5 * 2$ の計算で継続を取って来たりはしないので、計算はそのまま reset がなかったのと同じように進む。つまり、$5 * 2$ を計算して 10 となり、それに 3 を加えて 13 が得られ、それが reset の返す値となり、最後に 1 を引いて 12 が返ってくる。

限定継続は「reset を try ... with Abort -> ... で置き換え、現在の式を raise Abort で置き換えたときに捨てられる部分」と理解することもできる。例えば、

```
reset (fun () -> 3 + [5 * 2]) - 1
```

という式の限定継続は、

```
(try 3 + [raise Abort] with Abort -> 0) - 1
```

を実行したときに捨てられる計算、つまり $3 + [\cdot]$ となる。

練習問題 2 次のプログラムにおける限定継続とその型は何か。

(1) `5 * reset (fun () -> [2 * 3] + 3 * 4)`

(2) `reset (fun () -> if [2 = 3] then "hello" else "hi") ^ " world"`

(3) `fst (reset (fun () -> let x = [1 + 2] in (x, x)))`

(4) `string_length (reset (fun () -> "x" ^ string_of_int [3 + 1]))`

2.4 限定継続をとって来る命令 shift

OchaCaml で限定継続を取って来るには shift を使う。shift 命令は、次のような形で使う。

`shift (fun k -> `M`)`

このように書くと、shift は

(1) 現在の限定継続を取り除き、

(2) それを関数にして k に束縛し、

(3) M を実行する。

という 3 ステップの動作を行う。すぐにはわかりにくいので、使用例をいろいろ見てみよう。

2.5 継続の破棄

以下のような式を考えてみる。

`shift (fun _ -> M)`

ここで _ は他にどこにも現れない変数のことである。この式は、他に現れない変数 k を使って `shift (fun k -> M)` と書いたのと同じことである。この式の実行は以下のように進む。

(1) 現在の限定継続を取り除き、

(2) それを関数にしたものが `(fun _ -> M)` に渡されるが、M の中ではその関数は使われない。結果として継続は破棄された上で、

(3) M を実行する。

従って、この式を実行すると現在の実行を直近の reset まで破棄して、代わりに M を実行することができる。

具体的にやってみよう。$3 + [5 * 2] - 1$ の継続部分を破棄して $5 * 2$ を全体の結果にするには、全体を reset でくくった上で `[·]` の部分に `shift (fun _ -> 5 * 2)` を入れれば良い。

```
# reset (fun () -> 3 + shift (fun _ -> 5 * 2) - 1) ;;
- : int = 10
#
```

この例では M は $5 * 2$ なので結果は 10 となる。しかし、ここで返す結果は別に整数でなくても構わない。

```
# reset (fun () -> 3 + shift (fun _ -> "hello") - 1) ;;
- : string = "hello"
#
```

ここで、破棄された継続は `int -> int` 型の $3 + [\cdot] - 1$ である。もともとは整数が返って来る予定だったのに、実際に返って来たのは文字列になっている。にもかかわらず上記の式は正しく型付けされている。この型の変化については後に触れる。

破棄されるのは reset でくくられている継続のみである。例えば、

```
# reset (fun () -> 3 + shift (fun _ -> 5 * 2)) - 1 ;;
- : int = 9
#
```

では、破棄される継続は $3 + [\cdot]$ のみで、1 を引く部分は破棄されないことになる。なお、この例では整数以外の値を返すことはできない。というのは reset から値が返って来たらその値から 1 を引かなくてはならないからである。

```
# reset (fun () -> 3 + shift (fun _ -> "hello")) - 1 ;;
Toplevel input:
> reset (fun () -> 3 + shift (fun _ -> "hello")) - 1 ;;
> ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
This expression has type string,
but is used with type int.
#
```

練習問題 3 次の式の `[·]` に `shift (fun _ -> M)` の形の式を入れてその時点での継続を破棄して適当な値を返してみよ。

(1) `5 * reset (fun () -> [·] + 3 * 4)`

(2) `reset (fun () -> if [·] then "hello" else "hi") ^ " world"`

(3) `fst (reset (fun () -> let x = [·] in (x, x)))`

(4) `string_length (reset (fun () -> "x" ^ string_of_int [·]))`

練習問題 4 次のような関数を考える。この関数は、整数のリストを受け取ったら、その要素全ての積を返す関数である。

```
(* times : int list -> int *)
let rec times lst = match lst with
    [] -> 1
  | first :: rest -> first * times rest
```

例えば、この関数を `[2; 3; 5]` で呼び出すと 30 を返す。今、この関数に `[2; 3; 0; 5]` を渡したとしよう。リストの要素の中に 0 が含まれているので、結果は積を全く計算しなくても 0 であることがわかる。これは、リストの中に 0 が見つかった時点で、そのときの継続を破棄して 0 を返すことで実現できる。上記のプログラムに以下のような行を加えてこの動作を実現せよ。

```
  | 0 :: rest -> ...
```

できあがったプログラムは、どのように呼び出せば良いだろうか。

2.6 継続の取り出し

以下の式を考える。

```
shift (fun k -> k)
```

この式の実行は以下のように進む。

(1) その時点での限定継続を取り除き、

(2) それを関数にして `k` に束縛し、

(3) `k` を実行する。

この最後の `k` の実行は単に `k` の値（捕捉した継続）を返すだけなので、`shift (fun k -> k)` を実行することで、その時点での継続を関数として得ることができる。

具体的にやってみよう。$3 + [5 * 2] - 1$ の継続部分を得るには全体を `reset` でくくった上で `[·]` の部分に `shift (fun k -> k)` を入れ、得られた結果を適当な変数に入れておけば良い。

```
# let f x = reset (fun () -> 3 + shift (fun k -> k) - 1) x ;;
f : int -> int = <fun>
#
```

ここでは、返ってくる値は（継続を表す）関数なので、`x` という引数を明示的に使っている。実は、以下のように引数なしでも書くことができる。

```
# let f = reset (fun () -> 3 + shift (fun k -> k) - 1) ;;
f : int => int = <fun>
#
```

しかし、このようにすると、`f` の定義が（「値」ではなく）`reset` 文になっているため、`f` の「答の型」が弱い多相になってしまう。（これは OchaCaml の値制限による。）このような普通とは違う関数を OchaCaml では `=>` という記号を使って表す。（3.4 節 参照。）ここでは別に弱い多相でも問題ないのだが、引数を明示することで簡単に回避できるので、そのようにしている。

さて、これで $3 + [\cdot] - 1$ という継続が（関数の形で）`f` という名前で得られる。この関数に値（例えば 10）を渡すと、それが `[·]` の部分の計算結果だと思ってその後の計算が実行される。

```
# f 10 ;;
- : int = 12
#
```

この場合、f は fun x -> 3 + x - 1 と同じ意味である。

練習問題 5 次の式の [·] に shift (fun k -> k) を入れて、継続を取り出してみよ。得られた継続はどのような関数か。その継続にいろいろな値を渡して実行してみよ。これらの継続の型は何か。

(1) reset (fun () -> 5 * ([·] + 3 * 4))

(2) reset (fun () -> (if [·] then "hello" else "hi") ^ " world")

(3) reset (fun () -> fst (let x = [·] in (x, x)))

(4) reset (fun () -> string_length ("x" ^ string_of_int [·]))

練習問題 6 次のような関数を考える。この関数は、受け取ったリストを再帰的に見ていくが、結果的には何もせずに受け取ったリストをそのまま返す恒等関数である。

```
(* id : 'a list -> 'a list *)
let rec id lst = match lst with
    [] -> []
  | first :: rest -> first :: id rest ;;
```

今、このリストを [1; 2; 3] というリストで以下のように呼び出したとしよう。

```
reset (fun () -> id [1; 2; 3]) ;;
```

すると、再帰呼び出しをして行き、最後に lst が空リストになる。この時点で関数 id は空リストを返すが、その時点での継続はどのようなもので、どのような型を持つか。実際に shift (fun k -> k) を入れて継続を取り出し、実行して確かめてみよ。

2.7 継続の保存

継続を自由に切り出して使うことができるようになると、実行を一時的に中断して取っておくことができるようになる。例えば、次のように定義される木構造を考える。

```
type tree_t = Empty
            | Node of tree_t * int * tree_t ;;
```

今、この木を次のように左から深さ優先で traverse しているとしよう。

```
(* walk : tree_t -> unit *)
let rec walk tree = match tree with
    Empty -> ()
  | Node (t1, n, t2) ->
      walk t1;
      print_int n;
      walk t2
```

このプログラムでは訪れた Node の値を順に表示する関数である。この関数を実行すると、例えば、次のようになる。

```
# let tree1 = Node (Node (Empty, 1, Empty), 2, Node (Empty, 3, Empty)) ;;
tree1 : tree_t = Node (Node (Empty, 1, Empty), 2, Node (Empty, 3, Empty))
# walk tree1 ;;
123- : unit = ()
#
```

この関数は、木の全ての Node を一気に見ていくが、そうではなく Node を訪れるたびに一時、実行を止めて、その Node の値を使った処理を行い、あとで再び実行を再開する、といったことをしたい場合には次のようにする。

```
let rec walk tree = match tree with
    Empty -> ()
  | Node (t1, n, t2) ->
      walk t1;
      yield n;
      walk t2 ;;
```

ここで yield は次のように定義される。

```
(* yield : int => unit *)
let yield n = shift (fun k -> Next (n, k)) ;;
```

このように変更すると、関数 walk は Node が見つかると実行を中断して n の値とその時点での継続を Next という構成子に入れて返す。従って、このプログラムを呼び出す側には、Node の値 n が（その先の traverse を中断して）すぐに返って来ることになる。その値を使った処理が終了して、さらに次の値が欲しくなったときには、組として渡されたもうひとつの値 k を () を引数にして呼び出せば良い。すると walk 中の yield 文の値が () になり、walk の実行が再開される。このように、継続をデータの中に格納して返すことで、関数の実行を一時的に止めることができるようになる。

このように変更すると walk は中で shift を実行するので、全体を reset で囲む必要がある。しかし、単に

```
reset (fun () -> walk tree1) ;;
```

のように囲むのでは型エラーとなる。というのは、walk は shift を使って Next (n, k) をコンテキストに返すからである。一方、tree1 が空の木だった場合には walk は () を返す。従って上のように walk の呼び出しを直接 reset で囲むと () と Next (n, k) の両方が reset の計算結果として返ってくる可能性があるので型エラーになる。

型エラーを避けるために「もうこれ以上、Node が存在しない」ことをしめす構成子 Done を導入する。そして、() ではなく Done を reset の結果として返すようにする。

```
(* start : tree_t -> 'a result_t *)
let start tree =
  reset (fun () -> walk tree; Done) ;;
```

ここで、reset の結果の型が、reset の中のはるか深いところに現れる yield 中の shift の影響を受けていることに注意をしよう。このように shift を含むプログラムに型をつけるためには、取り囲む reset の型（答の型）が必要になる。

ここまで来ると、あと定義しなくてはならないのはふたつの構成子 Next と Done のみである。これらは次のように定義される。

```
type 'a result_t = Done
                 | Next of int * (unit / 'a -> 'a result_t / 'a) ;;
```

Done は引数を持たないのに対して、Next は Node の値（整数）と継続を引数に持つ。継続の型は、() を受け取ったら Done か Next のどちらかを返すので、ほぼ unit -> 'a result_t である。しかし、shift/reset の入った体系では、それに加えて「答の型」も指定しなくてはならない。捕捉された継続の「答の型」は多相になるので型変数 'a が答の型となる。答の型については、順次、以降の節で触れていく。

実行例を見てみよう。次の関数は、受け取った木の中の値を全て順に表示する。

```
(* print_nodes : tree_t -> unit *)
let print_nodes tree =
  let rec loop r = match r with
      Done -> ()        (* no more nodes *)
    | Next (n, k) ->
        print_int n;    (* print n *)
```

```
        loop (k ()) in  (* and continue *)
  loop (start tree) ;;
```

最後の行で start を使って木の traverse を開始し、中で定義されている再帰関数 loop で traverse した結果を調べている。もし、それが Done ならこれ以上、Node はないということなので終了、一方、Next (n, k) なら、現在の値 n を処理した後、k に () を渡すことで traverse を再開している。この関数を tree1 で呼び出すと以下のようになる。

```
# print_nodes tree1 ;;
123- : unit = ()
#
```

同様にして、次の関数は木の中の値の合計を返す関数である。

```
(* add_tree : tree_t -> int *)
let add_tree tree =
  let rec loop r = match r with
      Done -> 0
    | Next (n, k) -> n + loop (k ()) in
  loop (start tree) ;;
```

実行例は以下。

```
# add_tree tree1 ;;
- : int = 6
#
```

このような実行の中断の考え方を使うと、コルーチンを実装することができる。次の練習問題は、その最も簡単な形と考えることができる。

練習問題 7 関数 same_fringe を作れ。この関数は tree_t 型の木をふたつ受け取ったら、両者を左から深さ優先で traverse したとき、Node に出てくる数字の並びが同じかどうかを判定する。例えば、次のふたつの木を same_fringe に渡すと true が返る。

```
let tree1 = Node (Node (Empty, 1, Empty), 2, Node (Empty, 3, Empty)) ;;
let tree2 = Node (Empty, 1, Node (Empty, 2, Node (Empty, 3, Empty))) ;;
```

この関数は、一度、ふたつの木をどちらもリストに展開すれば、簡単に作成できるが、このような実装では、たとえ最初の値が異なっていても木全体を見て展開しなくてはならない。そうではなく、最初から順に見ていって異なる部分が見つかった時点で（それ以上、木を見ることなく）false を返すように実装せよ。

2.8 継続の包み込み： printf

shift でとってきた継続 k に値を渡すと、それはその値を結果として k に保存されている継続が実行される。この実行をすぐに行うのではなく fun 文で包んであげると、その後の計算の実行を遅らせることができる。さらに、この考え方を使うと reset の外の引数にアクセスできるようになる。

例えば、次のようなプログラムを考えてみよう。

```
shift (fun k -> fun () -> k "hello")
```

このプログラムは、現在の継続を切り取り、それを k とした上で、一時的にその実行を中止する。そして、取り囲む reset の値として fun () -> k "hello" という thunk（0 引数関数）を返す。reset の外で引数 () を受け取ったら、実行を "hello" という値で再開する。つまり、このプログラムは全体としては "hello" を返すのと同じようなものだが、それを返す前に一時、実行を中断して () を受け取るようになっている。

このプログラムを [ ] ^ "world" というコンテキストに入れると、一時的に実行は中断して thunk が返ってくる。[1]

[1] f の答の型を多相にするために引数 x を明示している。

```
# let f x = reset (fun () ->
              shift (fun k -> fun () -> k "hello") ^ " world") x ;;
f : unit -> string = <fun>
#
```

この thunk は () を受け取ったら、"hello" で実行を再開するので、次のようになる。

```
# f () ;;
- : string = "hello world"
#
```

ここで thunk が現れるのは reset の（はるか）内側であるのに対し、() を渡すのは reset の外であることに注意をしよう。

```
shift (fun k -> fun () -> k "hello")
```

のような式を書くことで、reset の内側にいながら reset の外側で渡される () にアクセスできているのである。このように、継続を関数で包んで関数を返すようにすると、reset の外側の情報にアクセスすることができるようになる。この考え方を応用すると型付きの printf を実装することができる [1]。

練習問題 8 次の式の [...] の部分に "world" を入れたら、"hello world!" という文字列を得ることができる。

```
reset (fun () -> "hello " ^ [...] ^ "!")
```

では、[...] の部分に適当な式を入れて、次のように reset 全体の引数が [...] に入るようにするにはどうすれば良いだろうか。言い換えると [...] に適当な式を入れることで以下のようなやりとりを実現することはできるだろうか。

```
# reset (fun () -> "hello " ^ [...] ^ "!") "world" ;;
- : string = "hello world!"
#
```

ここで [...] は %s と同じような働きをしていると考えることができる。さらに（%d のように）引数として文字列ではなく整数を受け取って、それを出力文字列に埋め込むためにはどうすれば良いだろうか。（整数を文字列にする関数 string_of_int を使って良い。）さらに引数を複数、渡すようにすることはできるだろうか。（OchaCaml は引数を「右から」実行することに注意せよ。）

2.9 答の型の変化

ここで shift/reset を含む式の型がどのように決まっているのかを考えてみよう。以下のようなコンテキストを考えてみる。

```
reset (fun () -> [...] ^ " world")
```

このコンテキストが返す値（reset が返す値）は ^ の返り値なので文字列であるように見える。ところが、printf の例ではこの文字列に "hello" という文字列を渡した。そんなことがなぜ型エラーを起こすことなく可能なのだろうか。

printf の例にどのような型が付くかを理解するためには、「答の型」が何であるかを知らなくてはならない。「答の型」とは、着目している式を取り囲む reset が返すものの型のことである。例えば、reset (fun () -> 3 + [5 * 2]) という式における答の型は int である。また、reset (fun () -> string_of_int [5 * 2]), という式の答の型は string である。

着目している式に shift が含まれていない場合には、hole の型さえ合っていれば（コンテキストが返す型には関わらず）どのようなコンテキストにも入れることができる。上の例では、5 * 2 という式は reset (fun () -> 3 + [...]) にも reset (fun () -> string_of_int [...]) にも入れることができていた。このことを指して 5 * 2 の答の型は多相である（任意である）という。

ところが着目している式に shift が含まれていると話は違ってくる。shift は現在の継続を（取り囲む reset まで）切り取って来て、代わりに新しい式（shift の本体）を実行するので、reset の返すものの型がもととは同じではなくなる場合がある。再び、以下のコンテキストを考えよう。

```
reset (fun () -> [...] ^ " world")
```

このコンテキストがもともと reset の値として返す予定だったものは文字列である。なので、このコンテキストの型は string -> string である。ここで [...] の中で次の式を実行したとしよう。

```
shift (fun k -> fun () -> k "hello")
```

この式を実行すると、取り囲む reset から返って来るのは thunk である。thunk の中の k の型は string -> string なので、結局、返ってくる thunk の型は unit -> string となる。全体として、もともとこの reset からは string 型の値が返ってくる予定だったのが、shift を含む式の実行によって、実際に返って来たのは unit -> string 型の値ということになる。これが「答の型の変化」と呼ばれている現象であり、上の例で () を引数として受け取れた理由である。

このように shift を含む式を実行すると取り囲む reset の型が変化する可能性があるため、shift/reset を含む言語の型を推論する際には答の型を常に意識する必要があり、OchaCaml の型推論はそのように実装されている。詳しくは 3.4 節を参照。

2.10 継続の包み込み：状態モナド

継続の適用を関数で包みこむと reset の外側の引数にアクセスすることができる。この方法を上手に使うと「状態」をサポートすることができる。

今、状態として整数ひとつを持つことを考えよう。そのために、整数ひとつを（printf のときの "world" のように）コンテキストの引数として渡すようにする。

```
reset (fun () -> M) 3
```

この例では、状態の初期値は 3 である。M の中では、次の関数でこの状態にアクセスすることができる。

```
# let get () =
    shift (fun k -> fun state -> k state state) ;;
get : unit => 'a = <fun>
#
```

get は shift で現在の継続を取って来た後、実行を中断して fun state -> k state state という関数を取り囲む reset に返す。今、取り囲むコンテキストには状態（上の例では 3）が引数として渡されているので、その値が変数 state に入ることになる。その上で k に state が渡されるので、結局、get の値があたかも state であったかのようにその後の実行が進むことになる。

get の定義で、k に state が 2 度渡されているのは、k の実行を再開した後の状態をセットするためである。多少、技術的になるが、捕捉した継続 k の一番、外側には reset がつく。従って k state とすると、その後の k の実行は再び reset の中で行われることになる。その中で再び get が使われたときのために、新しい状態を渡しているのである。get を実行しても状態の値は変わらないとするなら、state をそのまま渡すことになる。もし、実行後に状態の値を変化させたいのなら、ここの部分に新しい状態の値を渡せば良い。例えば、次の関数は状態の値を 1 増やして () を返す。

```
# let tick () =
    shift (fun k -> fun state -> k () (state + 1)) ;;
tick : unit => unit = <fun>
#
```

計算を開始するには以下の関数を使う。

```
# let run_state thunk =
    reset (fun () -> let result = thunk () in
```

```
                    fun state -> result) 0 ;;
run_state : (unit => 'a) => 'b = <fun>
#
```

この関数は、受け取った thunk を初期状態 0 のもとで実行する。thunk の実行が終了したら、その時点での状態の値は無視して結果 result を返している。

実行例は以下。

```
# run_state (fun () ->
    tick ();            (* state = 1 *)
    tick ();            (* state = 2 *)
    let a = get () in
    tick ();            (* state = 3 *)
    get () - a) ;;
- : int = 1
#
```

練習問題 9 次のプログラムを実行すると結果は何になるだろうか。

```
run_state (fun () ->
  (tick (); get ()) - (tick (); get ())) ;;
```

実際に実行して確認せよ。(OchaCaml の実行順に注意せよ。)

練習問題 10 同様にして、状態の値を変更する関数 put を作成せよ。put x とすると、状態を x にした上で () を返すようにせよ。

ここで述べてきた方法は、状態モナドを継続を使って実装した形になっている。

2.11 継続を使った実行順序の変更(発展)

shift でとってきた継続 k に値を渡した後に別の計算を行うと、その計算と k の計算の順序を入れ替えることができる。例えば

```
# reset (fun () -> 1 + (shift (fun k -> 2 * k 3))) ;;
- : int = 8
#
```

のような式を実行すると 1 + [...] という継続の実行と 2 * [...] という計算の順序が入れ替わり、3 に 1 を加えてから 2 倍される。この考え方を応用すると A 正規形変換を実装することができる。

以下のようなλ計算の構文を考えよう。

```
type term_t = Var of string
            | Lam of string * term_t
            | App of term_t * term_t ;;
```

A 正規形変換とは、λ計算の項を受け取ったら、それと同じ意味の項で全ての関数呼び出しに他と重ならない名前をつけるような変換である。例えば、S コンビネータ $\lambda x.\lambda y.\lambda z.(xz)(yz)$ を A 正規形変換すると以下のようになる。

$$\lambda x.\lambda y.\lambda z.\mathsf{let}\ t_1 = xz\ \mathsf{in}\ \mathsf{let}\ t_2 = yz\ \mathsf{in}\ \mathsf{let}\ t_3 = t_1 t_2\ \mathsf{in}\ t_3$$

A 正規形変換を実装するには、まず、受け取ったλ項全体を traverse するけれども、何もせずにもとのλ項を再構成して返すような恒等関数を定義する。

```
(* id_term : term_t -> term_t *)
let rec id_term term = match term with
    Var (x) -> Var (x)
  | Lam (x, t) -> Lam (x, id_term t)
```

```
  | App (t1, t2) -> App (id_term t1, id_term t2) ;;
```

今、全ての関数呼び出しに対して名前を与えたいので、この関数の最後のところを以下のように変更しよう。

```
(* id_term' : term_t -> term_t *)
let rec id_term' term = match term with
    Var (x) -> Var (x)
  | Lam (x, t) -> Lam (x, id_term' t)
  | App (t1, t2) ->
      let t = gensym () in   (* generate fresh variable *)
      App (Lam (t, Var (t)), (* let expression *)
           App (id_term' t1, id_term' t2)) ;;
```

考えている構文に let 文が存在しないので、ここでは $\mathsf{let}\ t = M\ \mathsf{in}\ N$ を同じ意味の式 $(\lambda t.N)M$ で代用している。この関数を使って S コンビネータを変換すると以下のような結果を得る。

$$\lambda x.\lambda y.\lambda z.\mathsf{let}\ t_1 = (\mathsf{let}\ t_2 = xz\ \mathsf{in}\ t_2)(\mathsf{let}\ t_3 = yz\ \mathsf{in}\ t_3)\ \mathsf{in}\ t_1$$

この結果は、確かに関数呼び出しごとに別の名前が与えられているが、let 文がネストしてしまっている。そうではなく、欲しいのはこの let 文が平たく展開されたようなものである。

ここで、S コンビネータを上の `id_term'` に渡して実行を開始し、今ちょうど最初の関数呼び出し xz に差し掛かったところだとしよう。この時点での継続は、以下に示すような継続、つまり「yz を変換し、その後、外側の関数呼び出しを作り、最後に 3 回、λ を作る」ようなものである。

$$\lambda x.\lambda y.\lambda z.\mathsf{let}\ t_1 = [\cdot](\mathsf{let}\ t_3 = yz\ \mathsf{in}\ t_3)\ \mathsf{in}\ t_1$$

すべての let 文を平たく展開するには、ここで今、着目している xz に対する let 文を作る計算（つまり $\mathsf{let}\ t_2 = xz\ \mathsf{in}\ [\cdot]$）と継続の計算（$\mathsf{let}\ t_1 = [\cdot](\mathsf{let}\ t_3 = yz\ \mathsf{in}\ t_3)\ \mathsf{in}\ t_1$）を以下のように入れ替える。

```
(* a_normal : term_t => term_t *)
let rec a_normal term = match term with
    Var (x) -> Var (x)
  | Lam (x, t) -> Lam (x, reset (fun () -> a_normal t))
  | App (t1, t2) ->
      shift (fun k ->
        let t = gensym () in    (* generate fresh variable *)
        App (Lam (t,            (* let expression *)
                  k (Var (t))), (* continue with new variable *)
             App (a_normal t1, a_normal t2))) ;;
```

最後の `App` の場合を見てみよう。`shift` で継続を `k` に取って来た後、その継続は新しく取って来た変数 `t` で再開されている。従って、この関数適用の結果はこの新しい変数ということになる。この新しい変数を使って入力の項に対する A 正規形変換の結果が得られたら、その後に変数 `t` の定義を加えているのである。

最後に加える変数の定義は、λ 抽象の変数のスコープを超えてしまわないよう `Lam` のケースには `reset` が追加されている。以上の定義で A 正規形変換が実現できる。これは、部分評価の分野では let insertion と呼ばれる有名な方法である。

練習問題 11 上の A 正規形変換を使って、実際に S コンビネータを変換してみよ。どのようになるか。

2.12 継続の複製

これまでの例は、どれもとってきた継続を 1 度しか使わなかったが、複数回、使うようにすると、バックトラックを実現できる。

次のような関数を考えてみよう。

```
(* either : 'a -> 'a -> 'a *)
let either a b =
  shift (fun k -> k a; k b) ;;
```

この関数は、引数 a、b を受け取ったら、現在の継続 k を捕捉し、それを a、b に対して順に適用する。つまり、この関数は a、b というふたつの値を順に返す関数である。両方の値が返されていることは、次のようなプログラムを実行してみるとわかる。

```
# reset (fun () ->
    let x = either 0 1 in
    print_int x;
    print_newline ()) ;;
0
1
- : unit = ()
#
```

関数 either が、その時点での継続を 2 度、呼び出しているので、x の表示と改行が 2 度、行われている。

練習問題 12 引数の数をふたつと固定するのではなく、リストの形で引数を受け取ってくることで引数の数をいくつでも良くしてみよう。引数としてリストをひとつ受け取ったら、その各要素を順番に返す関数 choice を作れ。

関数 either を使うと、どちらの値を選んだら良いのかわからないときに、両方を試してみて正しかった方を採用するという generate and test の処理を簡単に書くことができるようになる。例えば、P と Q というふたつの論理変数を使った次のような論理式

$$(P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee \neg Q)$$

が充足可能かを調べたいとしよう。P と Q は true か false のどちらかの値をとるので、次のようなプログラムを書くことができる。

```
# reset (fun () ->
    let p = either true false in
    let q = either true false in
    if (p || q) && (p || not q) && (not p || not q)
    then (print_string (string_of_bool p);
          print_string ", ";
          print_string (string_of_bool q);
          print_newline ())) ;;
true, false
- : unit = ()
#
```

このプログラムは表面上、ループもバックトラックもない一本道のプログラムになっていることに注意しよう。単に p と q を定義し、それらが論理式を満たすかどうかをチェックしているだけである。実際には、関数 either はその後の計算を 2 回行うので、プログラムに現れる if は合計 4 回、実行されることになる。見方を変えると、関数 either は引数 a、b を非決定的に選ぶ関数と考えることもできる。

練習問題 13 Using choice, define a function that searches for three natural numbers between 1 and 5 that satisfy the Pythagorean theorem. In other words, find $1 \leq x, y, z \leq 5$ such that $x^2 + y^2 = z^2$.

練習問題 14 先に定義した関数 choice を使って、ピタゴラスの定理を満たす 1 から 5 までの整数、言い換えると $x^2 + y^2 = z^2$ を満たす 1 から 5 までの整数 x, y, z を求めよ。

3　限定継続命令の基礎理論

この節では、限定継続命令 `shift`/`reset` の基礎理論を概観する。対象とする言語は多相の let 文が入った left-to-right[2] で値呼び (call by value; CBV) の λ-計算を限定継続命令で拡張したものである。

3.1　構文

$$
\begin{array}{rrcl}
\text{(値)} & V & ::= & x \mid \lambda x.\, M \\
\text{(項)} & M & ::= & V \mid M\, M \mid \mathsf{let}\ x = M\ \mathsf{in}\ M \mid \mathcal{S}k.\, M \mid \langle M \rangle \\
\text{(純評価文脈)} & F & ::= & [\,] \mid F\, M \mid V\, F \mid \mathsf{let}\ x = F\ \mathsf{in}\ M \\
\text{(評価文脈)} & E & ::= & [\,] \mid E\, M \mid V\, E \mid \mathsf{let}\ x = E\ \mathsf{in}\ M \mid \langle E \rangle
\end{array}
$$

`shift (fun k -> M)` は $\mathcal{S}k.\, M$ と書き、`reset (fun () -> M)` は $\langle M \rangle$ と書く。通常の評価文脈はふたつにわかれている。純評価文脈というのは hole [ ] が `reset` で囲まれていないような評価文脈を示す。

3.2　簡約規則

$$
\begin{array}{rcll}
(\lambda x.\, M)\, V & \rightsquigarrow & M[x := V] & \\
\mathsf{let}\ x = V\ \mathsf{in}\ M & \rightsquigarrow & M[x := V] & \\
\langle V \rangle & \rightsquigarrow & V & \\
\langle F[\mathcal{S}\, V] \rangle & \rightsquigarrow & \langle V\, (\lambda x.\, \langle F[x] \rangle) \rangle & x \text{ is fresh}
\end{array}
$$

最初のふたつの規則はλ計算の普通の β-簡約、3つ目の規則は `reset` の中身が値になったら `reset` は外れるという規則、最後の規則は次に実行すべき式が `shift` だった場合、直近の `reset` までの継続をとって来る規則である。直近の `reset` までの継続は純評価文脈 $F[\,]$ で表現されている。この継続が $\lambda x.\, \langle F[x] \rangle$ という関数に変換されて V に渡される。右辺で全体の `reset` が残っている点と、切り取られた継続 $F[\,]$ のまわりにも `reset` が残っている点に注意しよう。（これらのどちらか、あるいは両方をなくすと、他の限定継続命令になる。）

3.3　実行規則と実行順序

実行規則は、簡約規則を評価文脈の中で行うという形で定義される。

$$E[M] \quad \rightarrow \quad E[M'] \quad \text{if} \quad M \quad \rightsquigarrow \quad M'$$

評価文脈の定義により、この体系では左から（関数部分を引数部分より先に）実行する。

実行規則をより詳しく述べると、式の実行は以下の3段階の繰り返しからなる。

(1) 式がすでに値であればそれが最終結果である。値でなければ式を $E[M]$ の形に分解する。ここで M は redex である。

(2) M を簡約規則に従って M' に簡約する。

(3) M' をもとの評価文脈 $E[\,]$ に戻して $E[M']$ とする。これが1ステップ簡約したあとの式である。

[2] OchaCaml は right-to-left である。

3.4 型規則

型変数を t とすると、型と型スキームは以下のように定義される。

$$\begin{array}{rrcl} \text{型} & \tau & ::= & t \mid \tau/\tau \to \tau/\tau \\ \text{型スキーム} & \sigma & ::= & \tau \mid \forall t.\sigma \end{array}$$

ここで $\tau_1/\alpha \to \tau_2/\beta$ は τ_1 から τ_2 への関数だが実行すると answer type が α から β に変化するような関数の型を示す [3]。関数が control effect を伴わない（`shift` を使っていない）なら α と β はともに同一の型変数となる。そのような関数のことを pure であると呼ぶ。answer type が polymorphic である、ということもある。answer type については次節で述べる。

OchaCaml では、pure な関数は（answer type は省略して）τ_1 `->` τ_2 と表現する。一方、（`shift` を行うような）pure でない関数はデフォルトでは answer type を省略するが answer type が polymorphic ではないことを示すため τ_1 `=>` τ_2 と表現する。answer type まで含めて見たいときには `#answer "all";;` というディレクティブを入れると τ_1 `/` α `->` τ_2 `/` β の形で表示されるようになる。

型判定は以下の形をしている。

$$\Gamma \vdash_p M : \tau$$
$$\Gamma, \alpha \vdash M : \tau, \beta$$

前者は「型環境 Γ のもとで式 M は pure で型 τ を持つ」ことを示し、後者は「型環境 Γ のもとで式 M は型 τ を持ち、M を実行すると answer type は α から β に変化する」ことを示す。型規則は以下のように与えられる。

$$\frac{(x:\sigma) \in \Gamma \quad \sigma \succ \tau}{\Gamma \vdash_p x : \tau}\ (\text{var}) \qquad \frac{\Gamma, x:\tau_1, \alpha \vdash M : \tau_2, \beta}{\Gamma \vdash_p \lambda x.\, M : \tau_1/\alpha \to \tau_2/\beta}\ (\text{fun})$$

$$\frac{\Gamma, \gamma \vdash M_1 : \tau_1/\alpha \to \tau_2/\beta, \delta \quad \Gamma, \beta \vdash M_2 : \tau_1, \gamma}{\Gamma, \alpha \vdash M_1\, M_2 : \tau_2, \delta}\ (\text{app}) \qquad \frac{\Gamma \vdash_p M : \tau}{\Gamma, \alpha \vdash M : \tau, \alpha}\ (\text{exp})$$

$$\frac{\Gamma \vdash_p M_1 : \tau_1 \quad \Gamma, x : \mathsf{Gen}(\tau_1, \Gamma), \alpha \vdash M_2 : \tau_2, \beta}{\Gamma, \alpha \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \tau_2, \beta}\ (\text{let})$$

$$\frac{\Gamma, k : \forall t.(\tau/t \to \alpha/t), \gamma \vdash M : \gamma, \beta}{\Gamma, \alpha \vdash \mathcal{S}k.\, M : \tau, \beta}\ (\text{shift}) \qquad \frac{\Gamma, \gamma \vdash M : \gamma, \tau}{\Gamma \vdash_p \langle M \rangle : \tau}\ (\text{reset})$$

ここで、$\sigma \succ \tau$ は型 τ が型スキーム σ のインスタンスになっていることを示し、$\mathsf{Gen}(\tau, \Gamma)$ は、型 τ 中の型変数のうち Γ に出てこない型変数について $\forall$ を付けることで得られる型スキームである。

3.5 Answer Type

Answer type というのは「現在のコンテキストの型」のことである。これは例を考えるとわかりやすい。例えば、

```
reset (fun () -> 3 + [5 * 2] - 1)
```

という式を考えてみよう。現在、着目している `[5 * 2]` という式の型は `int` でり、その式を取り囲むコンテキスト `3 + [5 * 2] - 1` 全体の型は（`5 * 2` の実行前も実行後も）`int` である。従って、`[5 * 2]` の answer type は（どちらも）`int` である。このことを次のような型判定で表す。

$$\Gamma, \texttt{int} \vdash \texttt{5 * 2} : \texttt{int}, \texttt{int}$$

最初の `int` が `5 * 2` を実行する前の answer type（コンテキストの型）、次の `int` が `5 * 2` 自身の型、最後の `int` が `5 * 2` を実行した後の answer type である。

上の例では、たまたま着目している式の型と answer type が同じになっているが、これらは異なることの方が普通である。例えば

```
reset (fun -> if [2 = 3] then 1 + 2 else 3 - 1)
```

ならば、現在、着目している `2 = 3` という式の型は `bool` である。一方、コンテキストの型は `1 + 2` または `3 - 1` の型になるので（`2 = 3` の実行前後どちらでも）`int` である。従って、型判定は以下のようになる。

$$\Gamma, \texttt{int} \vdash \texttt{2 = 3} : \texttt{bool}, \texttt{int}$$

着目している式の型と answer type は異なるのが普通だが、ふたつの answer type については pure な式については必ず等しくなる。さらに pure な式では answer type が着目している式の型に影響を及ぼすことはないので、answer type を完全に無視することができる。

次に、answer type が変化する例を考えてみよう。

```
reset (fun () ->
  [shift (fun k -> fun () -> k "hello")] ^ " world")
```

現在、着目している式 `shift (fun k -> fun () -> k "hello")` の型が何であるのかはすぐにはわかりにくいが、その場合は周りを見てみるとわかりやすい。`[...] ^ " world"` のように `[...]` の結果が `^` に渡されているので、`[...]` の部分は `string` のはずである。従って `shift (...)` の型は `string` である。次に、`[...]` を実行する前のコンテキストの型は `[...] ^ " world"` という形をしているので `string` である。一方、`[...]` を実行した後はどうなっているだろうか。実行後は、コンテキストは切り取られて `k` に束縛され、このコンテキストから返ってくるのは `fun () -> k "hello"` という関数である。この関数は `unit` を受け取ったら（`"hello world"` という）文字列を返すので、だいたい `unit -> string` 型である。従って型判定は次のような感じになる。

$$\Gamma, \texttt{string} \vdash \texttt{shift (fun k -> ...)} : \texttt{string}, \texttt{unit -> string}$$

（実際には、関数の型にも answer type が必要なので `unit -> string` は `unit / 'a -> string / 'a` とするのが正しい。）このように `shift` を使うと answer type がいろいろに変化することになる。この現象を指して answer type modification と呼ぶ。

同様にして関数の型も考えることができる。例えば、2.10 節に出て来た関数 `get` を考えてみよう。この関数は `()` を受け取って、現在の状態（整数）を返す関数なので answer type を無視すれば `unit -> int` 型である。では、answer type はどうなっているかというと、それには `get` が使われているコンテキストの方を考えればよい。例えば、

```
reset (fun () ->
  let result = [get ()] + 2 * get () in
  fun i -> result)
```

を考えると、コンテキストは最終的に `fun i -> result` を返しているので、関数の型 `int -> int` を持つ。コンテキストの型は `get` の実行に関わらずいつも状態を引数としてとる形なので、不変である。結局 `get` の型は `unit / (int -> int) -> int / (int -> int)` のような感じになる。実際には、`get` の定義を見ただけでは状態の型が整数に制限されることはなく、さらにコンテキストの型もより一般的なものとなって

```
unit / ('a / 'b -> 'c / 'd) -> 'a / ('a / 'b -> 'c / 'd)
```

になる。

このように `shift`/`reset` が入ってくると answer type を考えなくてはならなくなるが、これはコンテキストの型がどうなっているか、それがどう変化するかを考えることに相当する。

参考文献

[1] Asai, K. “On Typing Delimited Continuations: Three New Solutions to the Printf Problem,” *Higher-Order and Symbolic Computation*, Vol. 22, No. 3, pp. 275–291, Kluwer Academic Publishers (September 2009).

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

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

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

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

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

[7] Gasbichler, M., and M. Sperber "Final Shift for Call/cc: Direct Implementation of Shift and Reset," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pp. 271–282 (October 2002).

[8] Gunter, C. A., D. Rémy, and J. G. Riecke "A Generalization of Exceptions and Control in ML-Like Languages," *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pp. 12–23 (June 1995).

[9] Kameyama, Y., and M. Hasegawa "A Sound and Complete Axiomatization of Delimited Continuations," *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pp. 177–188 (August 2003).

[10] Kiselyov, O. "Delimited Control in OCaml, Abstractly and Concretely: System Description," In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (LNCS 6009)*, pp. 304–320 (April 2010).

[11] Masuko, M., and K. Asai "Direct Implementation of Shift and Reset in the MinCaml Compiler," *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, pp. 49–60 (September 2009).

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

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