第7章　接続（continuation）

この章では、**goto** や **break**, **continue** などのジャンプ命令に意味を与えるために＿＿＿（continuation・＿＿＿ともいう）の概念を導入する。接続は直観的には＿＿＿＿＿＿＿＿＿＿＿＿を表す。例えば、次のようなCのプログラムでは:

```
int main(int argc, char** argv) {
    printf("The result is %d.", 1+fact(10));
}
```

下線の部分の接続は、プログラムの残りの部分 — 1を足してその結果を出力する、という操作である。

どのようなプログラム処理形でも、プログラムの実行中は何らかの形でこの接続の情報を保持しているはずである。機械語レベルでは、接続は＿＿＿＿＿＿＿＿＿＿＿＿（program counter）と＿＿＿＿＿の組に相当する。ジャンプ命令を解釈するためには、この接続の概念を明示的に扱う必要がある。

また、＿＿＿＿＿や＿＿＿＿＿など一部の言語は、接続をプログラマが明示的に扱うことを可能にしている。これによってコルーチン（coroutine）など、さまざまな自明でない制御構造を実現することができる。

この章では接続の概念を導入し、そのさまざまな応用を紹介する。

7.1　UtilCont – 接続の導入

Util に **break**, **continue** などを導入するために、Expr の定義に次のように構成子を追加する。また、**goto** 文を導入するため、ラベルも導入する。

```
data Expr = Const Target | Var String | If Expr Expr Expr | While Expr Expr
          | Let [Decl] Expr | Val Decl Expr
          | Lambda String Expr | Delay Expr | App Expr Expr
          -- ここまでは、Util1と同じ
          | Begin [LabeledExpr]     -- ブロック
          | Break                   -- break 文
          | Continue                -- continue 文
          | Goto String             -- goto 文
          deriving Show

type LabeledExpr = (Maybe String, Expr)  -- ラベル付きの式
```

これに対する具象構文としては、

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \ldots \mid \mathit{begin\ LabeledExprSeq} \\
&\mid \mathit{break} \mid \mathit{continue} \mid \mathit{goto\ Var} \\
\mathit{LabeledExprSeq} &\rightarrow \mathit{LabeledExpr\ end} \mid \mathit{LabeledExpr}\ ;\ \mathit{LabeledExprSeq} \\
\mathit{LabeledExpr} &\rightarrow \mathit{Expr} \mid \mathit{Var} : \mathit{Expr}
\end{aligned}
$$

を想定する。

次に **break**, **continue** などを解釈するために接続の概念を導入する。接続（continuation）のモナドは単独では次のような型になる。

```
type K r a = ____________________

unitK :: a -> K r a
unitK a = _______________

bindK :: K r a -> (a -> K r b) -> K r b
m `bindK` k = ___________________________________

abortK :: r -> K r a
abortK r = ____________
```

直観的には a -> r が接続（“以後実行すべき操作”）の型になる。unitK a は、____________ ____________。m `bindK` k は、____________（\ a -> k a c）を m に渡す。m は最後にこの接続を呼び出すのが普通だが、無視したり、他の接続を呼び出したりすることも可能である。これが、ジャンプなどの命令に対応する。例えば、abortK r は現在の接続を無視して r という値を全体の計算の結果としている。これは計算を途中で中止することに相当する。

実際の UtilCont では接続とともに状態も扱いたいので、計算のモナド KST では、型パラメータ r は状態の変化を表す ST s v とする。ここで、s は状態の型である。いまの UtilCont のバージョンでは s は三つ組であると定義しておく。（UtilST の時と同様、3 という数に特別の意味はない。）

```
type KST s v a = _________________
  {- = (a -> s -> (v, s)) -> s -> (v, s) -}
```

setX など状態に関する関数も、この KST の定義にあわせて書き直しておく。

```
failK :: String -> KST s () a
failK e = abortK (unitST ())

setXK ::  x -> KST (x, y, z) a ()
setXK v = ____________________________________
-- setYK, setZK も同様に定義する。

getXK :: () -> KST (x, y, z) a x
getXK () = ____________________________________
-- getYK, getZK も同様に定義する。
```

failK はその時の状態・接続はすべて無視して、 () をプログラムの結果とする。setXK v は状態の第 1 要素に v をセットし、() と新しい状態を接続に渡す。

Const, Var, Let などに対しては comp は変更する必要はない。変更された部分のうち、Goto, Break, Continue に対する comp の定義は以下のようになる。

```
comp (Goto l)          = mkGoto l
comp Break             = mkGoto "_break"
comp Continue          = TApp1 (TVar "abortK")
                               (TApp1 (TVar "_while") (TVar "_break"))

mkGoto l  = TApp1 (TVar "abortK") (TApp1 (TVar l) (TVar "()"))
```

goto, **break**, **continue** について、変換前と変換後をそれぞれ Util と Haskell の文法で記述すると次の表になる。

ソース（Util）	ターゲット（Haskell）
goto label	abortK (label ())
break	abortK (_break ())
continue	abortK (_while _break)

goto label, **break** はそれぞれ、現在の接続は無視して、label, _break という識別子に束縛されている接続を起動する。これが “ジャンプ” に相当する。**continue** も、現在の接続は無視して、_while という識別子に束縛されている計算に _break という接続を渡す。

while 〜 **do** 〜に対しては、**break** に対応する接続を変数に格納する必要があるため、定義がやや複雑になる。

```
comp (While e1 e2)     = compWhile e1 e2

compWhile e1 e2 = TLambda1 "_break"
                  (TLet [(PVar "_while", body)]
                        (TApp1 (TVar "_while") (TVar "_break")))
  where body = comp e1 `TBindM` TLambda1 "_b"
               (TIf (TVar "_b") (comp e2 `TBindM` TLambda0 (TVar "_while"))
                                (TUnitM (TVar "()")))
```

ソース（Util）	ターゲット（Haskell）

while *c* **do** *t*	\ _break ->
	  let _while = *c*' `bindM` \ _b ->
	  if _b **then** *t*' `bindM` \ _ ->
	  _while
	  else ()
	  in _while _break

ここで、_break は＿＿＿＿＿＿＿＿を表す接続で、_while _break は＿＿＿＿＿＿を表す接続である。これらの接続が、それぞれ **break**, **continue** に対応する。

例えば、UtilCont プログラム（右は対応する C プログラム）：

```
let foo = \ y -> begin
  setXM 1; setYM y;
  while getYM() > 0 do begin
    val x = getXM() in val y = getYM() in
    if y==10 then break
    else if y==3 then begin
      setYM (y-1); continue
    end else ();
    setXM (x*y);
    setYM (y-1)
  end;
  getXM()
end in
foo 9
```

```
int foo(int y) {
   int x=1;
   while (y>0) {

     if (y==10) break;
     else if (y==3) {
       y--; continue;
     }
     x=x*y;
     y--;
   }
   return x;
 }
```

をコンパイルすると、次の Haskell プログラムが得られる。

```
foo = \ y ->
        setXK 1                  `bindK` \ _ ->
        setYK y                  `bindK` \ _ ->
        (\ _break ->
          let _while
                = getYK ()         `bindK` \ y ->
                  if y > 0 then
                       getXK ()              `bindK` \ x ->
                       getYK ()              `bindK` \ y ->
                       (if y == 10 then abortK (_break ()) else
                          if y == 3 then
                              setYK (y - 1)     `bindK` \ _ ->
                              abortK (_while _break)
                            else unitK ())  `bindK` \ _ ->
                       setXK (x * y)         `bindK` \ _ ->
                       setYK (y - 1)         `bindK` \ _ ->
                       _while
                  else unitK ()
            in _while _break)  `bindK` \ _ ->
        getXK ()
```

foo の型は Integer -> KST (Integer,Integer,z) a Integer であるから、値を取り出すためには、整数と初期接続（通常 unitST）、初期状態（(0,0,0) など）を渡す必要がある。fst (foo 9 unitST (0,0,0)) の結果は、_____に、9 を 11 に変える (fst (foo 11 unitST (0,0,0))) と結果は __ になる。

goto に対する意味を与えるためには、ブロック（**begin**～**end**）のなかで、"ラベル" に適切な接続を与える必要があるが、Begin に対する comp の定義は長くなってしまうので、ここに示さず、変換前と変換後の形の例のみを示す。

ソース（Util）	ターゲット（Haskell）

begin
  label1: s_1
  label2: s_2
  label3: s_3
end
	\ _end -> **let**
	  label1 = \ () -> s'_1 label2
	  label2 = \ () -> s'_2 label3
	  label3 = \ () -> s'_3 _end
	in label1 ()

s_1, s_2, s_3 の中には、 **goto** label1, **goto** label2, **goto** label3 が含まれているかもしれない。ターゲット（Haskell）プログラム中の識別子 label1,label2, label3 に束縛されているのはそれぞれ、同名のラベル label1, label2, label3 に対応する接続である。

例えば、次の UtilCont プログラム（右は対応する C プログラム）：

```
bar = \ _ -> begin
    setX 1;
  label1:
    if getX () > 100 then goto label2 else ();
    setX (getX () * 2);
    goto label1;
  label2:
    getX ()
  end
```

```
void bar(void) {
  int x = 1;
label1:
  if (x>100) goto label2;
  x = x * 2;
  goto label1;
label2:
  return x;
}
```

は次のような Haskell プログラムにコンパイルされる。

```
bar = \ _ ->
        setX 1     `bindK` \ _ ->
        \ _end ->
          let label1 = \ _ ->
                       (getX ()                                  `bindK` \ x ->
                        (if x > 100 then abortK (label2 ())
                                    else unitK ())               `bindK` \ _ ->
                        getX ()                                  `bindK` \ x ->
                        setX (x * 2)                             `bindK` \ _ ->
                        abortK (label1 ())) label2
              label2 = \ _ -> getX () _end
            in label1 ()
```

fst (bar () unitST (0,0,0)) を評価すると、結果は ____になる。

7.2 Scheme 概説

Scheme は、Lisp の一方言である。Scheme は関数型言語であるが、Haskell と異なり、変数への代入など命令的な特徴を残している。このため __関数型言語と言える。また遅延評価ではなく、関数の引数を先に評価する、先行評価を採用している。

関数適用 関数適用 (function application) は次のような形である。

- (関数 引数$_1$ 引数$_2$ ... 引数$_n$) のような _____でくくった式の列

Scheme では + や × などの算術演算子に、通常の _____(infix notation) ではなく、_____ (prefix notation) を用いることが特徴的である。例えば、`(+ 1 2)` という式では、+が関数（function），1 と 2 が引数である。

変数と代入 例えば、

```
(define x 5)
```

という式で、5 という値の入った “x” という名前の変数を用意する。これ以降は x という変数は 5 に評価される。

Scheme の場合、変数名の中には、アルファベット、数字の他に

```
+ - . * / < = > ! ? : $ % _ & ~ ^
```

などの記号を用いることができる。（もちろん空白はダメ）アルファベットの大文字と小文字は _____ _____。（つまり、Japan と japan は _____変数である。）

set!という命令によって、変数の値を変更する（代入するという）ことができる。（C 言語の 「=」演算子に対応する。）

```
(set! x 4)          ; 変数 x の値を 4 に変更する。
                    ; それ以前に x を define しておく必要がある。
```

これは、Scheme が _____としての側面を持つことを示す。

リスト リストを入力するためには、組み込み関数 list を用いる。list は任意の数の引数を取ることができる。

```
> (list 1997 5 6)
(1997 5 6)
> (list "kagawa" "university")
("kagawa" "university")
```

単に (1997 5 6) と入力すると、Scheme の処理系は、1997 という関数を 5 と 6 という引数に適用しているのだと判断する。

このように、Scheme（一般に Lisp）では小括弧「(」、「)」が 2 つの意味に使われる。ユーザが入力するときは「_____」の意味に、処理系が出力するときは「_____」の意味になる。もっと正確に言うとユーザが「リスト」を入力すると、処理系はそれを「関数適用」だと解釈するのである。
このような処理系の振舞いは Lisp の強力さの源であるが、一方で混乱のもとでもある。

上記のデータは「’」（クォート記号・引用記号）を用いて次のようにも入力できる。

```
> '(1997 4 22)
(1997 4 22)
```

「' 式」は、「(quote 式)」とも書く。(むしろ、後者が正式な書き方である。)

```
> (quote (1997 5 6))
(1997 5 6)
```

quote は、＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿だから、*(1997 5 6)* は関数適用ではなくリストと解釈される。

空リスト（要素を 1 つも含まないリスト）は'() または (list) のように入力する。

```
> '()
()
> (list)
()
```

cons（＿＿＿＿＿と読む）, car（＿＿＿＿と読む）, cdr（＿＿＿＿＿と読む）などが、リストを操作するための最も基本的な関数である。 cons はリストを組み立てるための関数、car と cdr はリストを分解するための関数である。

cons — リストの先頭に 1 つ新たに要素を付け加えたリストを返す関数

car — リストの先頭の要素を返す関数

cdr — リストの先頭を除いた残り（のリスト）を返す関数

関数定義　関数の定義には次の形式の define を用いる。

(define (関数名 変数 $_1$ ... 変数 $_n$) 定義)

変数 $_1$ ... 変数 $_n$ はこの関数の仮引数である。

```
> (define (square x) (* x x))
square
> (square 4)
16
```

条件判断　条件判断は次のような形式で行なう。

(if 条件式 式 $_1$ 式 $_2$)

条件式が＿＿＿＿＿＿を、＿＿＿＿＿＿を評価（計算）する。(C の **if** 文と異なり、値を返すことに注意する。むしろ、C の?:オペレータに対応する。)

逐次実行

(begin 式 $_1$ 式 $_2$ ... 式 $_n$)

式$_1$から、式$_n$を順に評価し、最後の式$_n$の値を全体の値として返す。通常、____ C や JavaScript のブロック { 〜 }と意味は似ているが、C や JavaScript のブロックは “文” の一種であるので値を持たないのに対し、Scheme の begin 式は値を持つ。

なお、関数の定義の本体で、

```
(define (hen_na_square x)
     (begin (set! x (* x x))
             x))
```

のように順に式を評価するときは、上のように begin を使う必要はなく、

```
(define (hen_na_square x)
     (set! x (* x x))
      x)
```

のように単に式を並べて書くだけで良い。（これを “暗黙” の begin という。）

局所変数（let） 関数の定義の他に let という構文で局所変数を導入することができる。

```
(let ((変数1 式1)
      ...
      (変数m 式m))
  式0)
```

let 文では、式$_1$から式$_m$を評価した結果が、変数$_1$から変数$_m$に入れられ、最後に式$_0$を評価する。変数$_1, \ldots,$ 変数$_m$のスコープは式$_0$である。

ラムダ式（匿名関数）

```
(lambda (変数1 ... 変数n) 定義)
```

これは 変数$_1 \ldots$ 変数$_n$を引数とする関数である。例えば、(lambda (x) (* x x)) は 2 乗する関数である。((lambda (x) (* x x)) 2) は 4 になる。lambda はギリシャ文字の λ のことである。これらは define を用いて定義した名前付きの関数:

```
(define (sq x) (* x x))
```

の sq と同じ関数になる。つまり、(define (sq x) (* x x)) は (define sq (lambda (x) (* x x))) と同じ意味なのである。

7.3 Scheme の call-with-current-continuation

Scheme では、プログラマが接続を直接操作することができる。このことを Scheme は ____ という言い方をするときもある。

```
(call-with-current-continuation thunk)
(call/cc thunk)
```

call-with-current-continuation という名前は長いので、省略形の call/cc がよく使われる[1]。

[1]Scheme は、-や/のような文字も変数の名前の中でに使用できるので、call-with-current-continuation や call/cc でひとつの名前である。ただし、call/cc は Scheme の標準仕様には含まれていないので、処理形によっては、

thunk は 1 引数の関数であり、(call/cc *thunk*) は ＿＿＿＿＿＿を引数として、*thunk* を呼び出す。*thunk* のなかで、この接続を呼び出せば、そのときの接続は無視されて（= ジャンプして）、call/cc が呼ばれた—!とき—の接続にその値が返される。*thunk* が接続を呼び出さなければ、*thunk* 自身の戻り値が call/cc 式全体の戻り値になる。

例えば、

```
(define (bar x)
  (call/cc (lambda (k)
    (+ 100 (if (= x 0) 1 (k x))))))
```

という関数を考える。(bar 0) を評価すると普通に足し算が計算され、値は ＿＿になる。一方、(bar 1) の場合は、接続 k が 呼び出されるので 100 を足す部分はスキップされて、戻り値は ＿となる。

call/cc のよくある使い方は、try〜catch と同じような大域脱出である。（右側には Util 風の書き方を示す。）

```
(define (multlist xs)
   (call/cc (lambda (k)
       (define (aux xs)
           (if (null? xs)     1
               (if (= 0 (car xs))
                   (k 0)
                   (* (car xs) (aux (cdr xs))))))

       (aux xs))))
```

この関数はリストの要素の掛け算を求める。要素の中に 0 が見つかると、大域脱出して multlist 全体の値は ＿になる。

しかし、このような大域脱出だけならば、言語の仕様に call/cc のような大がかりな仕掛けをいれておく必要はない。call/cc の本当の価値はコルーチンなどの普通でない制御構造を実現できるところにある。

7.4 コルーチン（coroutine）

コルーチンとは、2 つ以上のプログラムの実行単位が、＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿のことである。サブルーチン（subroutine）のように、実行単位の間に主と副といった従属関係はなく、コルーチンを構成する個々のルーチンは互いに対等な関係である。

例えば、

```
(define (increase n k)
   (if (> n 10) '()
   (begin (display " i:") (display n)
          (increase (+ n 1) (call/cc k)))))
(define (decrease n k)
   (if (< n 0) '()
   (begin (display " d:") (display n)
          (decrease (- n 1) (call/cc k)))))
```

```
(define call/cc call-with-current-continuation)
```

のように自分で定義しておく必要がある。

という 2 つの関数を定義して

```
(increase 0 (lambda (k) (decrease 10 k)))
```

という式を実行すると、

というように画面へ出力される。increase と decrease という 2 つの関数が交互に実行されていることがわかる。

call/cc はひじょうに強力なプリミティブで、コルーチンの他にこれまでに紹介したエラー処理（try〜catch）や非決定性などのプリミティブも、call/cc を用いて定義できることがわかっている。ある意味でオールマイティのプリミティブである。

しかし、call/cc は効率的な実装の難しいプリミティブでもある。素直な実装では call/cc を実現するためには、スタック全体のコピーを行なう必要がある。一方、はじめからスタックをヒープの中に取り、スタックのコピーを行なわないという方式もある。この方式では不要になったスタック領域も _____で回収する。

7.5 call/cc の表現

我々の言語 UtilCont に call/cc を導入するには、接続を関数として渡すためのコードを用意すれば良い。call/cc に対応する関数の定義は次のようになる。

```
callccK :: ((a -> K r b) -> K r a) -> K r a
callccK h = ___________________________________________
```

callccK の定義中で用いられている k は現在の接続（d）を捨て、キャプチャされた接続（c）を呼び出すという関数である。

コンパイラは単に callccK という名前の UtilCont の関数を Haskell の callccK にコンパイルすれば良い。

ソース（Util）	ターゲット（Haskell）
`callccK` *m*	*m*’ ‘bindK‘ \ _x -> callccK _x

また、head, tail, null, not, show などの 1 引数で副作用を持たない関数は、次のようにコンパイルされる。

ソース（Util）	ターゲット（Haskell）
`funWithOneArg` *m*	*m*’ ‘bindK‘ \ _x -> unitK (funWithOneArg _x)

例えば、次の UtilCont プログラム:

```
multlist = \ xs ->
  let aux = \ xs -> \ k -> begin
        setXM 1; setYM xs; setZM "";
        while not (null (getYM())) do begin
          val y = getYM() in val n = head y in
          if n == 0 then k 0 else
            begin setXM (getXM()*n); setYM (tail y);
                  setZM (getZM() ++ " " ++ show n)
            end
        end;
        getXM ()
    end in
  val result = callccK (\ k -> aux xs k)  in
  "result = " ++ show result ++ "; z = \"" ++ getZM () ++ "\";"
```

をコンパイルすると、次の Haskell プログラムが得られる。

```
multlist = \ xs ->
  let aux = \ xs ->
        unitK (\ k ->
            setXK 1                ‘bindK‘ \ _ ->
            setYK xs               ‘bindK‘ \ _ ->
            setZK ""               ‘bindK‘ \ _ ->
            (\ _break ->
               let _while
                     = getYK ()                              ‘bindK‘ \ y ->
                       if not (null y) then
                           getYK ()                              ‘bindK‘ \ y ->
                           (if head y == 0 then k 0 else
                              getXK ()                           ‘bindK‘ \ x ->
                              setXK (x * head y)                 ‘bindK‘ \ _ ->
                              setYK (tail y)                     ‘bindK‘ \ _ ->
                              getZK ()                           ‘bindK‘ \ z ->
                              setZK (z ++ " " ++ show (head y)))
                                                              ‘bindK‘ \ _ ->
                           _while
                         else unitK ()
                 in _while _break) ‘bindK‘ \ _ ->
            getXK ())
    in callccK (\ k -> aux xs ‘bindK‘ \ _f ->
                       _f k)                  ‘bindK‘ \ result ->
       getZK ()                               ‘bindK‘ \ z ->
       unitK ("result = " ++ show result ++ "; z = \"" ++ z ++"\";")
```

fst (multlist [1,2,3,4,5] unitST (0, [], "")) の結果は “result = 120; z = " 1 2 3 4 5";”、fst (multlist [1,2,3,0,4,5] unitST (0, [], "")) の結果は “result = 0; z = " 1 2 3";” となる。

7.6 Continuation-Passing Style（CPS）

ここからは、Util から離れて、接続の概念の応用を説明する。

Continuation-Passing Style（CPS）とは ______________________________ のことである。次のような使い途がある。

- call/cc のない言語でコルーチンなど ______________ を実現したいときに用いる
- プログラムを効率の良い形に変換したいときに、変換の途中の中間形式で用いる

CPS のプログラムは次のような制限に従う。

- 関数呼び出しが ____________。（つまり、関数呼び出しの引数は関数呼び出し[2]になっていることがない。）

例えば、

```
prodPrimes n = if n==1 then 1
               else if isPrime n then n * prodPrimes (n-1)
                                 else prodPrimes (n-1);
```

という関数を考える。これは 1 から n までの範囲に存在する素数の積を求める関数である。isPrime は素数かどうかを判定する関数とする。これを、CPS 変換すると、次のような関数に変換される。（ここには定義を示していないが、isPrime を CPS 変換した関数を isPrime' とする。）

```
prodPrimes' n c = if n==1 then c 1
                  else isPrime' n (\ b ->
                         if b then prodPrimes' (n-1) ______________
                              else prodPrimes' (n-1) c);
```

（便宜上、Haskell の記法で紹介しているが、他のプログラミング言語でも同様の変換は可能である。）
isPrime' を呼び出すときに、戻ってきたときに行なうべき処理を接続:

```
\ b ->if b then prodPrimes' (n-1) (\ p -> c (n*p))
           else prodPrimes' (n-1) c
```

として isPrime' に渡している。さらにこの接続の中で、prodPrimes' を呼び出すときに、b の値に応じて、n を掛けてから c に渡すという接続: \ p -> c (n*p)、またはもとのままの接続である c を渡している。

CPS はコンパイラの中間言語として用いられることがある。これは関数の呼び出しの順番が明確になり、関数の呼び出しを単なるジャンプ命令で実現して良いという性質があるからである。

プログラムを CPS に変換するには、だいたい次のような手順で行なう。

1. すべての関数定義に ____________ を一つ追加する
   prodPrime n = ... ⟹ prodPrime' n c = ...

2. 関数の戻り値に相当する位置にある単純な式は、________。（ここで**単純な式**とは ···
   定数、変数、ラムダ式、プリミティブオペレータ（-, == など）を単純な式に適用した式、のいずれか）
   ... **then** 1 ... ⟹ ... **then** c 1 ...

[2]ただし、+や*のようなプリミティブな関数の呼び出しは除く。

3. 関数の戻り値に相当する位置にある（単純な式でない）関数適用は、＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿。
   ... **else** prodPrimes (n-1) ... ⟹ ... **else** prodPrimes' (n-1) c
4. その他の位置にある（単純な式でない）関数適用は、“適切[3]な” 接続を明示的に受け取る形に変換する。
   ... **then** n*prodPrimes (n-1) ... ⟹ ... **then** prodPrimes' (n-1) (\ p -> c (n*p)) ...

より正確には、プログラムを UtilCont のプログラムと見て Haskell にコンパイルし、unitM を “\ a c -> c a”、bindM を “\ c -> m (\ a -> k a c)” に置き換えれば CPS 変換になる。（ただし、実際には変換後、見易い形にするためのさらなる整理が必要になる。）

7.7 JavaScript 超概説

ここからは、JavaScript (ECMAScript) の記法を用いることにするので、JavaScript の基本をごく簡単に説明する。

変数 JavaScript には型チェックはないので、＿＿＿ というキーワードで変数を宣言する。

```
var i=0;
```

制御構造 条件判断 （**if** 文）, 繰返し （**while** 文, **for** 文）はほとんど C 言語と同じである。

関数の定義 関数の定義も C 言語と良く似ているが、JavaScript では型を気にする必要がないので、C 言語で関数の戻り値の型を書く部分に、キーワード ＿＿＿＿＿＿＿を用いるところだけが異なる。また、仮引数の型を宣言する必要もない。

```
function cube(n) {
  return n*n*n;
}
```

匿名関数 JavaScript でも無名の関数を定義することができる。JavaScript では次のような形を用いる。

function (変数$_1$, ... , 変数$_n$) { 定義 }

つまり、**function** というキーワードと括弧の間に関数名がない。

7.8 CPS の応用—再帰呼出しの繰返しへの変換

CPS を利用してプログラムの変換を行なうことがある。例として再帰的関数を CPS を経由して繰返しへ変換する場合を取り上げる。

[3]“適切な” 接続の正確な定義をここで与えることは断念する。要するに元のプログラムと意味が変わらず、CPS 変換を施す目的が達成できれば良い。

変換の対象は、次のように定義された階乗の関数である。

```
function fact(n) {
  if (n==0) return 1;
  else return n*fact(n-1);
}
```

これは数学的な記法の定義:

$$0! = 1$$
$$n! = n \times (n-1)! \quad (n > 0)$$

に直接対応していてわかりやすいが、実行時には n に比例するスタック領域が必要にある。

この fact を CPS に変換すると次のようなプログラムになる。

```
function fact(n, c) {
  if (n==0) return _____;
  else return fact(n-1, ____________________________________________);
}
```

さらに、これは末尾再帰なので、次のように繰返しに書き換えることができる。

```
function aux(n, c) { return function (r) { return c(n*r); }; }

function fact(n, c) {
  while(n>0) {
    c = aux(n, c);  n--; // 注[4]
  }
  return c(1);
}
```

繰返しに変換されたが、c がどんどん大きくなってしまうので、領域の節約にはならない。しかし、良く観察すると c は常に次のような形の関数であることがわかる。

つまり、fact の場合、第 2 引数として本当の接続を受け渡さなくても、この n*(n-1)* ... *m で接続を表現可能ということである。このことを考慮に入れてさらにプログラムを変換すると、次の定義が得られる。

```
function fact(n, m) {
  while(n>0) {
    ___________ n--;
  }
  return m;
}
```

これは、通常の繰返しによる階乗関数の定義である。このように非末尾再帰を除去する場合、まず CPS に変換して末尾再帰のかたちにし、それから “接続” を同等のオブジェクトに入れ換えるとよい。

[4] ここは c = function (r) { return c(n*r); }; と書くことはできない。JavaScript のセマンティクスでは、右辺の変数 c の値も変わってしまうからである。aux 関数を介すると c の値がコピーされるため安全である。

7.9 CPS の応用—Web プログラミング

Servlet や JavaScript などで WWW 上のインタラクティブなアプリケーションを作成するときに、プログラムの任意の場所でユーザの入力を待って、続きから実行するという書き方ができない（必ず doGet などの関数のはじめから実行されてしまう）という制約がある。

そこで、インタラクティブなプログラムを実現するために、さまざまなテクニックが必要になるが、CPS への変換はある意味でオールマイティな（つまり、どんな場合にも適用可能な）手段である[5]。

トリッキーな例として JavaScript のハノイの塔のプログラム:

```
function move(n, a, b) {     // 非 CPS 版
  document.form.textarea.value += ("move "+n+" from "+a+" to "+b);
}

function hanoi(n, a, b, c) { // 非 CPS 版
  if (n>0) {
    hanoi(n-1, a, c, b);
    move(n, a, b);
    hanoi(n-1, c, b, a);
  }
}
```

を「ボタンを押したら 1 行表示する」というバージョンに書き換える、ということを考える。つまり、

```
<script type="text/javascript">
function move(n, a, b) {  // form の TextArea に追加する。
  document.form.textarea.value += ("move "+n+" from "+a+" to "+b+"\n");
}
</script>

<form name="form">
<input type="button" onClick="exec()" value="実行"><br>
<textarea name="textarea" cols="20" rows="32"></textarea>
</form>
```

というフォームの「実行」ボタンを押せばテキストエリアに 1 行表示するようにする。

まず、hanoi を CPS に書き換える。

```
function move(n, a, b, k) {     // 暫定版（説明用）
  document.form.textarea.value += ("move "+n+" from "+a+" to "+b+"\n");
  return k();
}
```

```
function hanoi(n, a, b, c, k) { // 最終版
  if (n>0) {
    return hanoi(n-1, a, c, b,
                 function () {
                   return move(n, a, b,
                               function() {
                                 return hanoi(n-1, c, b, a, k);
                               });
                 });
  } else {
    return k();
  }
}
```

[5] もちろん、言語に最初から call/cc が用意されていれば、このような面倒なことをする必要がない。

しかし、ここで、

```
function exec() {                  // 暫定版（説明用）
  hanoi(5, 'a', 'b', 'c', function () { return null; });
}
```

のように、hanoi を呼び出しても、これまで通り一気に最後まで出力してしまうだけである。そこで move を次のように書き換える。

```
function move(n, a, b, k) {      // 最終版
  document.form.textarea.value += ("move "+n+" from "+a+" to "+b+"\n");
  return _;      // ____ ではない。
}
```

つまり、最後に接続を呼び出してしまわず、いったん呼び出し側に接続を戻り値として返す。（このような手法をトランポリンと言う。）これで call/cc と同じような接続を明示的に扱う効果が得られる。この接続を利用するために exec を次のように書き換える。

```
function doEnd() {                 // 最終版
  document.form.textarea.value += "end\n"; // 最後の処理
  return doEnd;
}

// 最初のエントリポイント
var k = function() { return hanoi(5, 'a', 'b', 'c', doEnd); };

function exec() {                  // 最終版

  ____________
}
```

exec は k () の実行結果を新しい k の値として保存するだけである。これで「実行」ボタンを押すたびに move が 1 回ずつ実行されるようになる。

問 7.9.1 上のやり方にならって、次の関数を「ボタンを押したら *1* 行表示する」というバージョンに書き換えよ。

```
function fib(m) {
  document.form.textarea.value += ("argument = "+m);
  var r;
  if (m<2) {
    r=1;
  } else {
    r=fib(m-1)+fib(m-2);
  }
  document.form.textarea.value += ("result for argument: "+m+" = "+r);
  return r;
}
```

接続の表現　JavaScript は匿名関数（ラムダ式）を持っているため、CPS への変換は比較的容易であったが、ラムダ式を持たない言語や効率を重視する場合では、_____を明示的に使用し、そのなかに接続に対応するデータを格納する必要がある。次のプログラムは、接続を n, a, b, c の各パラメータと次に実行を開始すべき場所（pc）から構成されるデータとして表現したものである。

```
function move(n, a, b) {          // 明示スタック版
  document.form.textarea.value += ("move "+n+" from "+a+" to "+b+"\n");
}
```

```
var stack = new Array();
stack.push(new Array(5, 'a', 'b', 'c', 0));

function hanoi(n, a, b, c, pc) { // 明示スタック版
  while(n>0) {
    switch (pc) {
    case 0:
      stack.push(new Array(n, a, b, c, 1));
      var tmp=c; c=b; b=tmp; n--;
      continue;
    case 1:
      stack.push(new Array(n-1, c, b, a, 0));
      move(n, a, b);
      return;
    }
  }
  return exec();
}
```

```
function exec() {                 // 明示スタック版
  if(stack.length>0) {
    var args=stack.pop();
    hanoi(args[0], args[1], args[2], args[3], args[4]);
  } else {
    document.form.textarea.value += "end\n";
  }
}
```

ここまでやってしまうとプログラムの実行途中で “接続” をファイルに保存したり、別のコンピュータで起動することさえ可能になる。

この章の参考文献

[1] John C. Reynolds, 「The Discoveries of Continuations」
Lisp and Symbolic Computation, 6, (233–247). 1993 年
接続に関する文献は数多くあるが、この論文は接続の「発見」について、振り返っている珍しいものである。

[2] Andrzej Filinski, 「Representing Monads」
21st ACM Symposium on Principles of Programming Languages. 1994 年
call/cc が「オールマイティ」であることについての説明を与えている。

[3] Richard Kelsey, William Clinger, and Jonathan Rees (Editors),
「Revised^5 Report on the Algorithmic Language Scheme」
http://www.schemers.org/Documents/Standards/R5RS/
Scheme の仕様書である。通常、略して R5RS と呼ばれる。call/cc の簡単な解説もある。

[4] T. Sekiguchi, T. Sakamoto, and A. Yonezawa,
「Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling」
Advances in Exception Handling Techniques. Springer-Verlag, LNCS 2022. 2001 年
http://www.yl.is.s.u-tokyo.ac.jp/amo/
Java などの命令型言語に、call/cc のような接続を扱うオペレータを導入する方法を述べている。

[5] Levent Erkök, and John Launchbury, 「Recursive Monadic Bindings」
Proc. of the International Conference on Functional Programming. 2000 年
mfixU などの不動点演算子について解説している。

[6] 「Continuations and Continuation Passing Style」
http://library.readscheme.org/page6.html
接続と CPS に関する重宝なリンク集のページである。