第6章　UtilCont — 接続（continuation）の導入

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。

6.1　接続のモナド

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 Value | Let Decl Expr | Var String
          | Lambda String Expr | App Expr Expr | If Expr Expr Expr
          | Letrec Decl Expr
          | GetX | SetX Expr | GetY | SetY Expr | GetZ | SetZ Expr
          | While Expr Expr | _____________________________
          | _______ | ___________ | _______ | _______________ | _______________;

type LabeledExpr = (Maybe String, Expr);
```

これに対する具象構文としては、

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \ldots \mid \texttt{begin}\ \mathit{LabeledExprSeq} \\
& \mid & \texttt{break} \mid \texttt{continue} \mid \texttt{abort} \\
& \mid & \texttt{goto}\ \mathit{Var} \mid \texttt{callcc}\ \mathit{Expr} \\
\mathit{LabeledExprSeq} & \rightarrow & \mathit{LabeledExpr}\ \texttt{end} \mid \mathit{LabeledExpr}\ \texttt{;}\ \mathit{LabeledExprSeq} \\
\mathit{LabeledExpr} & \rightarrow & \mathit{Expr} \mid \mathit{Var}\ \texttt{:}\ \mathit{Expr}
\end{array}
$$

を想定する。

次に abort, break, continue などを解釈するために接続の概念をインタプリタに導入する。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 v = ____________;
```

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 v は現在の接続を無視して v 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。

実際に UtilCont で使用する計算の型 M では、接続とともに、状態を扱わなければいけないので、この r は状態の変化を表す State -> State とする。この State は状態の型である。いまのバージョンでは State は Value の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 State = (Value, Value, Value);
type Result = State -> State;
type M a = _____________;
           {- = (a -> State -> State) -> State -> State -}

unitM :: a -> M a;
unitM a = unitK a;

bindM :: M a -> (a -> M b) -> M b;
m `bindM` k = m `bindK` k;
```

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

```
failM :: String -> M a;
failM message = abortK (\ s -> (Str ("failure: "++message), Unit, Unit));

setX :: Value -> M Value;
setX v = ___________________________________________;
-- setY, setZ も同様に定義する。

getX :: M Value;
getX   = ___________________________________________;
-- getY, getZ も同様に定義する。


lookupM :: String -> Env -> M Value;
```

```
lookupM x ((n,v):rest) = if n==x then unitM v else lookupM x rest;
lookupM x [] = _______ ("Variable: "++x++" is not found");
```

failM はその時の環境・状態・接続はすべて無視して、“failure: . . . ” というメッセージをプログラムの結果とする。（便宜上、状態の第 1 要素にセットする。）setX v は状態の第 1 要素に v をセットし、Unit と新しい状態を接続に渡す。

interp を書き換える前に、接続を値として扱えるように Value 型を拡張しておく。これは break や continue や goto を実現するために、環境の中に接続を格納しておけると便利だからである。

```
data Value = ...  | _________________________________;
```

Const, Var, Letrec などに対しては interp は変更する必要はない。

Break, Continue は環境の中の break, continue という識別子に束縛されている接続を lookup し、現在の接続は無視して、その接続を起動する。これが “ジャンプ” に相当する。Abort は、接続を無視して現在の状態をそのまま計算の最終結果として返す。

```
interp Break    e = \ c0 -> lookupM "break" e
                                    ____________________________________________;
interp Continue e = \ c0 -> lookupM "continue" e
                                    ____________________________________________;
interp Abort    e = _____________________________;
```

While に対しては、適切な接続を環境に格納する必要があるため、定義がやや複雑になる。

```
interp (While m1 m2) e =
  \ c1 ->
  let { e1 = (("break", Cont c1) : e) } in
  interp m1 e (\ v ->
  case v of {
    Bool b -> if b then
                  let { c2 = \ _ -> interp (While m1 m2) e c1 } in
                  let { e2 = (("continue", Cont c2) : e1) } in
                  interp m2 e2 c2
              else c1 (Bool True);
    _      -> fairM "Boolean expected" c1
  });
```

ここで、c1 は _____________を表す接続で、c2 は _____________を表す接続である。これらの接続をそれぞれ、break, continue という識別子に束縛した環境（env）のもとで m2 を評価する。

これまでと同じように run という関数を

```
run :: String -> String;
run str = showValue (fst3 (interp (myParse str) initEnv
                                  (\ v (_, y, z) -> (v, y, z)) (Unit, Unit, Unit)));

fst3 (a, b, c) = a;
```

と定義すると、例えば、

```
run ("let foo = \ n -> begin              "++  -- C の記法では
     "  setX 1; setY n;                   "++  -- int foo(int n) {
     "  while getY > 0 do begin           "++  --   int r=1;
     "    if getY==10 then break          "++  --   while (n>0) {
```

```
"    else if getY==3 then begin
"      setY (getY-1); continue
"    end else 1;
"    setX (getX*getY);
"    setY (getY-1)
"  end;
"  getX
"end in
"foo 9
```

```
"++  --    if (n==10) break;
"++  --    else if (n==3) {
"++  --      n--; continue;
"++  --    }
"++  --    r=r*n; n--;
"++  --  }
"++  --  return r;
"++  -- }
")
```

の結果は、＿＿＿＿＿＿に、最後の行の foo 9 を foo 11 に変えると結果は ＿＿＿＿＿になる。

自由な飛躍命令である goto に対する意味を与えるために、まず、“ラベル”を解釈する必要があるが、これに対する interp を定義をここに示すと長くなってしまうので、アイデアのみを示す。

例えば、

```
begin
  l1:  Exprs1 /*  --- この中に goto l1; goto l2; を含むかもしれない */
  l2:  Exprs2 /*  --- この中に goto l1; goto l2; を含むかもしれない */
end
```

のような文の意味は、c_1, c_2 を l1, l2 の接続とすると、

$$\begin{aligned} c_1 &= \lambda \_ \to \texttt{interp}\ Exprs_1\ e'\ c_2 \\ c_2 &= \lambda \_ \to \texttt{interp}\ Exprs_2\ e'\ c \\ e' &= e + \{\texttt{l1} \mapsto \texttt{Cont}\ c_1, \texttt{l2} \mapsto \texttt{Cont}\ c_2\} \end{aligned}$$

のような式で意味を与えられる。そして、

$$\texttt{interp}\ (\texttt{l1:}Exprs_1\ \texttt{l2:}Exprs_2)\ e\ c = c_1\ \texttt{Unit}$$

と解釈する。この c, e はこの部分全体を解釈する時の接続と環境である。

goto はラベル名に対応する接続を環境から読み出して、現在の接続は無視して、単にその接続を起動する。

```
interp (Goto label) e = \ c0 -> lookupM label e ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿;
```

例えば、

```
run (
"begin
"    set 1;
"  l1:
"    if get > 100 then goto l2 else Unit;
"    set (get * 2);
"    goto l1;
"  l2:
"    get
"end
```

```
    -- /* C の記法では */
"++ -- {
"++ --    x = 1;
"++ -- l1:
"++ --    if (x>100) goto l2;
"++ --    x = x * 2;
"++ --    goto l1;
"++ -- l2:
"++ --    return x
")  -- }
```

を評価すると、結果は ＿＿＿＿＿になる。

6.2 Scheme超入門

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$_1$ ... 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。

6.3 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時もある。

```
(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 が呼ばれた時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 (foo x)                      ; let foo = \ x ->
  (call/cc (lambda (k)               ;     callcc (\ k ->
    (+ 100 (if (= x 0) 1 (k x))))))  ;          100 + (if x==0 then 1 else k x))
```

という関数を考える。（右側には Util 風の書き方を示す。）(foo 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、(foo 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)                ; let multlist = \ xs ->
   (call/cc (lambda (k)              ;   callcc (\ k ->
       (define (aux xs)              ;     letrec aux = \ xs ->
           (if (null? xs)     1      ;       if null xs then 1
               (if (= 0 (car xs))    ;       else if car xs == 0
                   (k 0)             ;            then k 0
                   (* (car xs) (aux (cdr xs))))))
                                     ;            else car xs * aux (cdr xs)
       (aux xs))))                   ;     in 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。

6.4 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)                 ; letrec increase = \ n -> \ k ->
   (if (> n 10) '()                    ;   if n > 10 then Unit
   (begin (display " i:") (display n) ;   else begin display " i:"; display n;
          (increase (+ n 1) (call/cc k))))) ;        increase (n+1) (callcc k) end;
(define (decrease n k)                 ;        decrease = \ n -> \ k ->
   (if (< n 0) '()                     ;   if n < 0 then Unit
   (begin (display " d:") (display n) ;   else begin display " d:"; display n;
          (decrease (- n 1) (call/cc k))));        decrease (n-1) (callcc k) end
```

(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)))
                                        ; in increase 0 (\ 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。

6.5 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。Callcc に対する interp の定義は次のようになる。

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

interp (Callcc m) e = interp m e `bindM` \ g ->
                      case g of {
                        Fun f -> ____________________________________;
                        _     -> failM ("Callcc: Function expected")
                      };
```

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。Callcc に対する interp の定義は、基本的に callccK を呼び出すだけである。

例えば、

```
run (
"let mult = \ xs -> \ k -> begin                                          " ++
"    setX 1; setY xs; setZ  "";                                            " ++
"    while isCons getY do begin                                            " ++
"      let n = car getY in                                                 " ++
"      if n == 0 then k 0 else                                             " ++
"        begin setX (getX*n); setY (cdr getY); setZ (getZ ++ " " ++ n) end " ++
"    end;                                                                  " ++
"    getX                                                                  " ++
"  end in                                                                  " ++
"let list   = cons 1 (cons 2 (cons 3 (cons 0 (cons 4 (cons 5 nil))))) in   " ++
"let result = callcc (\ k -> mult list k)  in                            " ++
"pair result getZ                                                         ")
```

の結果は (Num 0.0, Str " 1.0 2.0 3.0") となる。

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。