第6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.¥n", 1+fact(10));
    return 0;
}
```

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系でも、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

接続（continuation）のモナドは単独では次のような型になる。

ファイル Cont.hs

```
1  newtype K r a = K (_____)
2
3  unK :: K r a -> (a -> r) -> r
4  unK (K c) = c
5
6  instance Monad (K r) where
7    return a      = K (_____)
8    (K m) >>= k   = K (\ c -> m (\ a -> unK (k a) c))
9    -- K, unK がなければ、
10   -- m >>= k = \ c -> m (\ a -> k a c)
```

直観的には r が “結果” の型、a -> r が接続（“以後実行すべき操作”）の型になる。return a は、接続（c）に _____渡す。m >>= k は、接続（c）の

_____という接続（\ 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。

6.2 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。

ファイル ContType.hs

```
1  data Expr = Const Target | Var String
2          | If Expr Expr Expr | While Expr Expr
3          | Let [Decl] Expr | Val Decl Expr
4          | Lambda String Expr | Delay Expr | App Expr Expr
5          -- ここまでは、Util1 と同じ
6          | Begin [LabeledExpr]     -- ブロック
7          | Break                   -- break 文
8          | Continue                -- continue 文
9          | Goto String             -- goto 文
10         deriving Show
11 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、

Expr	→	... | **begin** *LabeledExprSeq*
	|	**break** | **continue** | **goto** *Var*
LabeledExprSeq	→	*LabeledExpr* **end** | *LabeledExpr* ; *LabeledExprSeq*
LabeledExpr	→	*Expr* | *Var* : *Expr*

を想定する。

実際の UtilCont では接続とともに状態や入出力も扱いたいので、計算のモナドは、K そのものではなく、“状態の変化” を “結果” として持つ K (ST (WithIO s) v) a（と同型の型）とする。ここで、s は状態の型である。

ファイル Cont.hs

```
1  newtype KST v s a = KST ((a -> WithIO s -> (v,WithIO s))
2                           -> WithIO s -> (v,WithIO s))
3
4  unKST :: KST v s a -> (a -> WithIO s -> (v,WithIO s))
5                     -> WithIO s -> (v,WithIO s)
6  unKST (KST m) = m
```

set 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。

ファイル Cont.hs

```
1  instance MyState (KST v) where
2    get p    = KST (\ c (s,i,o) -> c (fst (p s)) (s,i,o))
3    set p v  = KST (\ c (s,i,o) -> c () (snd (p s) v,i,o))
4
```

```
5   instance MyStream (KST v s) where
6     readChar   = KST (\ c (s,ch:i,o) -> c ch (s,i,o))
7     eof        = KST (\ c (s,i,o) -> c (null i) (s,i,o))
8     writeStr v = KST (\ c (s,i,o) -> c () (s,i,o ++ v))
9
10  abort :: (WithIO s -> (v,WithIO s)) -> KST v s a
11  abort r = KST (__________)
```

set p v は状態の p で表される位置に v をセットし、() と新しい状態を接続に渡す。abort 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。

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。

ファイル ContCompiler.hs

```
1   comp (Goto l)          = mkGoto l "()"
2   comp Break             = mkGoto "_break" "()"
3   comp Continue          = mkGoto "_while" "_break"
4
5   mkGoto l v = TApp1 (TVar "abort") (TApp1 (TVar l) (TVar v))
```

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	abort (label ())
break	abort (_break ())
continue	abort (_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式に翻訳される。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。

ファイル ContCompiler.hs

```
1   comp (While e1 e2)     = compWhile e1 e2
2
3   compWhile e1 e2 = TApp1 (TVar "KST")
4     (TLambda1 "_break"
5       (TLet [(PVar "_while", TApp1 (TVar "unKST") body)]
6             (TApp1 (TVar "_while") (TVar "_break"))))
7     where body = comp e1 `TBind` TLambda1 "_b"
8             (TIf (TVar "_b") (comp e2 `TBind` TLambda1 "_"
9                    (TApp1 (TVar "KST") (TVar "_while")))
10                   (TReturn (TVar "()")))
```

ソース（Util）	ターゲット（Haskell）

while c **do** t	KST (\ _break ->
	  let KST _while = $\dot{c}$ >>= \ _b ->
	  if _b **then** $\dot{t}$ >>= \ _ ->
	  KST _while
	  else return ()
	  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）：

```
1  foo = \ y -> begin
2    set xP 1; set yP y;
3    while get yP > 0 do begin
4      val x = get xP in
5      val y = get yP in
6      if y == 10 then break
7      else if y == 3 then begin
8        set yP (y - 1); continue
9      end else ();
10     set xP (x * y);
11     set yP (y - 1)
12   end;
13   get xP
14 end
```

```
1  int foo(int y) {
2     int x = 1;
3     while (y > 0) {
4
5       if (y == 10) break;
6       else if (y == 3) {
7         y--; continue;
8       }
9       x = x * y;
10      y--;
11    }
12    return x;
13  }
```

をコンパイルすると、次の Haskell プログラムが得られる。

```
1  foo = \ y ->
2          set xP 1                >>= \ _ ->
3          set yP y                >>= \ _ ->
4          KST (\ _break ->
5            let KST _while
6                  = get yP          >>= \ y ->
7                    if y > 0 then
8                         get xP               >>= \ x ->
9                         get yP               >>= \ y ->
10                        (if y == 10 then abort (_break ()) else
11                           if y == 3 then
12                               set yP (y - 1    >>= \ _ ->
13                               abort (_while _break)
14                            else return ()) >>= \ _ ->
15                        set xP (x * y)       >>= \ _ ->
16                        set yP (y - 1)       >>= \ _ ->
17                        KST _while
18                   else return ()
19             in _while _break)  >>= \ _ ->
20         get xP
```

foo の型は Integer -> KST a (Integer,Integer) 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、\ a s -> (a,s)）、初期状態（((0,0),"","") など）を渡す必要がある。

```
1  fst (unKST (foo 9) (\ a s -> (a,s)) ((0,0),"",""))
```

の結果は、_____ に、9 を 11 に変えると結果は ___ になる。(参考: 9! = 362880)

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
  lbl1: s_1
  lbl2: s_2
  lbl3: s_3
end
	KST (\ _end -> **let**
	  lbl1 = \ _ -> unKST $\dot{s}_1$ lbl2
	  lbl2 = \ _ -> unKST $\dot{s}_2$ lbl3
	  lbl3 = \ _ -> unKST $\dot{s}_3$ _end
	  in lbl1 ())

s_1, s_2, s_3 の中には、**goto** lbl1, **goto** lbl2, **goto** lbl3 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 lbl1,lbl2, lbl3 に束縛されているのはそれぞれ、同名のラベル lbl1, lbl2, lbl3 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）:

```
1  bar = \ _ -> begin
2      set xP 1;
3    label1:
4      if get xP > 100 then goto label2
5                      else ();
6      set xP (get xP * 2);
7      goto label1;
8    label2:
9      get xP
10   end
```

```
1  int bar(void) {
2    int x = 1;
3   label1:
4    if (x > 100)
5      goto label2;
6    x = x * 2;
7    goto label1;
8   label2:
9    return x;
10 }
```

は次のような Haskell プログラムにコンパイルされる。

```
1  bar = \ _ ->
2          set xP 1     >>= \ _ ->
3          KST (\ _end ->
4            let label1 = \ _ -> unKST
5                          (get xP                >>= \ x ->
6                           (if x > 100 then abort (label2 ())
7                              else return ())     >>= \ _ ->
8                           get xP                 >>= \ x ->
9                           set xP (x * 2)         >>= \ _ ->
10                          abort (label1 ())) label2
11               label2 = \ _ -> unKST (get xP) _end
12             in label1 ())
```

そして、

```
fst (unKST (bar ()) (\ a s -> (a,s)) ((0,0),"",""))
```

を評価すると、結果は ____になる。

問 6.2.1 次の C の関数とほぼ同等な Haskell の関数をモナドを用いて作成せよ。

```
1  int hoge(int n) {
2    int i = 1, sum = 0;
3    while (i <= n) {
4      sum = sum + i;
5      if (sum > 21) {
6        sum = 0;
7        break;
8      }
9      i = i + 1;
10   }
11   return sum;
12 }
```

6.3 callcc とは

callcc は Scheme や Ruby などが採用している、プログラマが接続を直接操作することができるプリミティブである。(Scheme では call/cc と書く。)

1 引数の関数 *thunk* に対して、callcc *thunk* のように使用すると _____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、呼出しの接続は無視されて (= ジャンプして)、callcc 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 callcc 式全体の戻り値になる。例えば、

```
1  baz = \ x ->
2    callcc (\ k ->
3      100 + (if x == 0 then 1 else k x))
```

という関数を考える。baz 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、baz 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。

callcc のよくある使い方は、try〜catch と同じような大域脱出である。

```
1  multlist = \ xs ->
2    let aux = \ xs -> \ k -> begin
3          set xP 1; set yP xs;
4          while not (null (get yP)) do begin
5            val y = get yP in val n = head y in
6            if n == 0 then k 0 else
7              begin set xP (get xP * n); set yP (tail y);
8                    writeStr "␣";
9                    write n
```

```
            end
        end;
        get xP
    end in
  val result = callcc (\ k -> aux xs k) in begin
    writeStr ";␣result␣=␣";
    write result
  end
```

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 `callcc` のような大がかりな仕掛けをいれておく必要はない。`callcc` 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、＿＿＿＿＿＿＿＿＿＿＿＿ながら実行されていく方式のことである。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、

```
  increase = \ n -> \ k ->
       if n > 10 then ()
       else begin writeStr "␣i:"; write n;
       increase (n + 1) (callcc k) end;
  decrease = \ n -> \ k ->
       if n < 0 then ()
       else begin writeStr "␣d:"; write n;
       decrease (n - 1) (callcc k) end
```

という 2 つの関数を定義して

```
increase 0 (decrease 10)
```

という式を実行すると、

＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

と出力される [1]。`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。スレッドと似ているが、2 つのルーチンが同時に実行される訳ではない。

[1]実は、この Util プログラムを Haskell に変換すると型エラーになり、そのままではコンパイル・実行できない。これはラムダ式の Y コンビネーターに対応する `fix f = (\ x -> f (x x)) (\ x -> f (x x))` という式が型エラーになるのと同じように、自己適用が起こるのが原因である。いくつかトリッキーな変換をすると、意味を変えずに型付け可能な定義に書き換えが可能で、上記のような実行結果になる。しかし、ここはコルーチンのアイデアを説明するのが本旨なので、型付けをするためのトリックの詳細には立ち入らないことにする。

なお、Scheme では、

callcc は、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、callcc 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、その詳細の解説については、ここでは割愛する。

6.5 callcc の表現

我々の言語 UtilCont に callcc 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 に対応する関数の定義は次のようになる。

ファイル Cont.hs

```
1  callcc :: ((a -> KST v s b) -> KST v s a) -> KST v s a
2  callcc h = KST (\ c -> let k a = KST (\ d -> c a)
3                         in unKST (h k) c)
4  -- KST, unKST がなければ
5  -- callcc h = \ c -> let k a = \ d -> c a
6  --                    in h k c
```

callcc 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。

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

ソース（Util）	ターゲット（Haskell）
`callcc` *m*	*m* `>>= \ _x ->` `callcc _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* `>>= \ _x ->` `return (funWithOneArg _x)`

例えば、先に紹介した UtilCont プログラム multlist をコンパイルすると、次の Haskell プログラムが得られる。

```
1  (define (increase n k)
2     (if (> n 10) '()
3     (begin (display "␣i:") (display n)
4            (increase (+ n 1) (call/cc k)))))
5  (define (decrease n k)
6     (if (< n 0) '()
7     (begin (display "␣d:") (display n)
8            (decrease (- n 1) (call/cc k)))))
```

のように対応する関数を定義して

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

という式を実行すると上記の出力が得られる。

```
multlist = \ xs ->
  let aux = \ xs ->
        return (\ k ->
            set xP 1               >>= \ _ ->
            set yP xs              >>= \ _ ->
            KST (\ _break ->
               let KST _while
                     = get yP             >>= \ y ->
                       if not (null y) then
                           get yP           >>= \ y ->
                           (if head y == 0 then k 0 else
                              get xP        >>= \ x ->
                              set xP (x * head y)
                                            >>= \ _ ->
                              set yP (tail y) >>= \ _ ->
                              writeStr "␣"    >>= \ _ ->
                              write (head y)) >>= \ _ ->
                          KST _while
                        else return ()
                  in _while _break) >>= \ _ ->
            get xP)
    in callcc (\ k -> aux xs >>= \ _f ->
                      _f k)     >>= \ result ->
       writeStr ";␣result␣=␣"   >>= \ _ ->
       write result
```

このとき

```
let (_,(_,_,o)) = unKST (multlist [1,2,3,4,5])
                        (\ a s -> (a,s)) ((0,[]),"","") in o
```

の結果は“ 1 2 3 4 5; result = 120”となる。一方、

```
let (_,(_,_,o)) = unKST (multlist [1,2,3,0,4,5])
                        (\ a s -> (a,s)) ((0,[]),"","") in o
```

の結果は“ 1 2 3; result = 0”となる。つまり、0 が現れた時点で乗算を打ち切っていることがわかる。

6.6 さらに詳しく知りたい人のために ...

接続に関する文献は数多くあるが、[1] 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] は、call/cc がある意味で「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] は、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。[4] は mfixU などの不動点演算子について解説している。

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

[2] Andrzej Filinski, 「Representing Monads」21st ACM Symposium on Principles of Programming Languages. 1994 年

[3] 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/

[4] Levent Erkök, and John Launchbury, 「Recursive Monadic Bindings」Proc. of the International Conference on Functional Programming. 2000 年