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.¥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。

```
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。

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。

```
1  type K r a = ________________
2
3  unitK :: a -> K r a
4  unitK a = _____________
5
6  bindK :: K r a -> (a -> K r b) -> K r b
7  m `bindK` k = _________________________________
8
9  abortK :: r -> K r a
10 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や入出力も扱いたいので、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。

```
1  type KST s v a = ________________
2    {- = (a -> s -> (v, s)) -> s -> (v, s) -}
```

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。

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

setK ::  Pos s a -> a -> KST (s, i, o) any ()
setK p v = \ c (s, i, o) -> c () (snd (p s) v, i, o)

getK :: Pos s a -> KST (s, i, o) any a
getK p = \ c (s, i, o) -> c (fst (p s)) (s, i, o)

readK :: () -> KST (s, String, o) any Char
readK () = \ c (s, ch:i, o) -> c  ch (s, i, o)

writeK :: Show v => v -> KST (s, i, String) any ()
writeK v = \ c (s, i, o) -> c () (s, i, o ++ show v)
```

```
15
16  putStrK :: String -> KST (s, i, String) any ()
17  putStrK str = \ c (s, i, o) -> c () (s, i, o ++ str)
```

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。setK 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。

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。

```
1  comp (Goto l)          = mkGoto l
2  comp Break             = mkGoto "_break"
3  comp Continue          = TApp1 (TVar "abortK")
4                                 (TApp1 (TVar "_while")
5                                        (TVar "_break"))
6
7  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。

```
1  comp (While e1 e2)     = compWhile e1 e2
2
3  compWhile e1 e2 = TLambda1 "_break"
4                    (TLet [(PVar "_while", body)]
5                          (TApp1 (TVar "_while") (TVar "_break")))
6    where body = comp e1 `TBindM` TLambda1 "_b"
7                 (TIf (TVar "_b") (comp e2 `TBindM`
8                                      TLambda0 (TVar "_while"))
9                                  (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）:

```
1  foo = \ y -> begin
2    setM xP 1; setM yP y;
3    while getM yP > 0 do begin
4      val x = getM xP in
5      val y = getM yP in
6      if y==10 then break
7      else if y==3 then begin
8        setM yP (y-1); continue
9      end else ();
10     setM xP (x*y);
11     setM yP (y-1)
12   end;
13   getM 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          setK xP 1              `bindK` \ _ ->
3          setK yP y              `bindK` \ _ ->
4          (\ _break ->
5            let _while
6                  = getK yP        `bindK` \ y ->
7                    if y > 0 then
8                         getK xP             `bindK` \ x ->
9                         getK yP             `bindK` \ y ->
10                        (if y == 10 then abortK (_break ()) else
11                           if y == 3 then
12                               setK yP (y-1)     `bindK` \ _ ->
13                               abortK (_while _break)
14                           else unitK ()) `bindK` \ _ ->
15                        setK xP (x*y)       `bindK` \ _ ->
16                        setK yP (y-1)       `bindK` \ _ ->
17                        _while
18                   else unitK ()
19             in _while _break)  `bindK` \ _ ->
20         getK xP
```

foo の型は `Integer -> KST ((Integer,Integer), i, o) 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),"","")` など）を渡す必要がある。`fst (fst (foo 9 unitST ((0,0),"","")))` の結果は、＿＿＿＿に、9 を 11 に変えると結果は＿＿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）:

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

```
1  void 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          setK xP 1      `bindK` \ _ ->
3          \ _end ->
4            let label1 = \ _ ->
5                           (getK xP                `bindK` \ x ->
6                            (if x > 100 then abortK (label2 ())
7                               else unitK ())       `bindK` \ _ ->
8                            getK xP                `bindK` \ x ->
9                            setK xP (x*2)          `bindK` \ _ ->
10                           abortK (label1 ())) label2
11               label2 = \ _ -> getK xP _end
12             in label1 ()
```

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

7.2 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、callccM *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）、callccM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     callccM (\ 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           setM xP 1; setM yP xs;
4           while not (null (getM yP)) do begin
5             val y = getM yP in val n = head y in
6             if n == 0 then k 0 else
7               begin setM xP (getM xP*n); setM yP (tail y);
8                     putStrM "␣";
9                     writeM n
10              end
11          end;
12          getM xP
13      end in
14    val result = callccM (\ k -> aux xs k) in begin
15      putStrM ";␣result␣=␣";
16      writeM result
17    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。

7.3 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、

```
1     increase = \ n -> \ k ->
2         if n > 10 then ()
3         else begin putStrM "␣i:"; writeM n;
```

```
4           increase (n+1) (callccM k) end;
5       decrease = \ n -> \ k ->
6           if n < 0 then ()
7           else begin putStrM "␣d:"; writeM n;
8           decrease (n-1) (callccM k) end
```

という 2 つの関数を定義して

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

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。

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。

7.4 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。

```
1   callccK :: ((a -> K r b) -> K r a) -> K r a
2   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 callccM という名前の UtilCont の関数を Haskell の callccM にコンパイルすれば良い。

ソース（Util）	ターゲット（Haskell）
`callccM` *m*	*m*' `` `bindM` `` `\ _x -> callccM _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。

[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 `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で、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。

ソース（Util）	ターゲット（Haskell）
`funWithOneArg` *m*	*m'* `` `bindM` \ _x -> unitM (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。

```
multlist = \ xs ->
  let aux = \ xs ->
        unitK (\ k ->
            setK xP 1                `bindK` \ _ ->
            setK yP xs               `bindK` \ _ ->
            (\ _break ->
               let _while
                     = getK yP              `bindK` \ y ->
                       if not (null y) then
                           getK yP            `bindK` \ y ->
                           (if head y == 0 then k 0 else
                              getK xP           `bindK` \ x ->
                              setK xP (x * head y)
                                                `bindK` \ _ ->
                              setK yP (tail y)  `bindK` \ _ ->
                              putStrK "␣"       `bindK` \ _ ->
                              writeK (head y))`bindK` \ _ ->
                           _while
                         else unitK ()
                 in _while _break) `bindK` \ _ ->
            getK xP)
    in callccK (\ k -> aux xs `bindK` \ _f ->
                       _f k)        `bindK` \ result ->
       putStrK ";␣result␣=␣"        `bindK` \ _ ->
       writeK result
```

`let (_, (_, _, o)) = multlist [1,2,3,4,5] unitST ((0, []), "", "") in o` の結果は “ `1 2 3 4 5; result=120`” となる。一方、`let (_, (_, _, o)) = multlist [1,2,3,0,4,5] unitST ((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。

7.5 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