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)の概念を導入する。

> **（補足）** “continuation” の日本語訳は「　　」のほうが一般的だが、ここでは「接続」を採用する。

接続は直観的には（ジャンプなどがなくて普通に実行するときに）　　　　を表す。例えば、次のような C のプログラムでは:

```c
int main(void) {
    printf("The result is %d.\n", 1 + fact(10));
    return 0;
}
```

2 行目の `fact(10)` の式の部分の接続は、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、接続をプログラマー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。

Q 6.0.1 次のように goto 文を使った C の関数 `baz` を定義する。

```c
int baz(int n) {
    int x = 0, y = 1;
  label1:
    if (x > n) goto label2;
    x += y;
    y++;
    goto label1;
  label2:
    return x;
}
```

この関数に対して、

1. `baz(50)`
2. `baz(100)`

の値を、まず実行する前に予想し、次に実際に実行して確認せよ。

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

```
newtype K r a = K (                    )

unK :: K r a -> (a -> r) -> r
unK (K c) = c

instance Monad (K r) where
  return a      = K (              )
  (K m) >>= k   = K (                                  )
  -- K, unK がなければ、
  -- 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、UtilCont の構文規則に、従来の Util に加えて次のような規則を追加する。

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

実際の UtilCont では接続とともに変数の破壊的代入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、"状態への操作"を"結果"として持つ `K (WithIO s -> r) 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

```
newtype KIO r s a
  = KIO (                                        )

unKIO :: KIO r s a -> (a -> WithIO s -> r)
                   -> WithIO s -> r
unKIO (KIO m) = m
```

この `KIO` の定義にあわせて、`set` など状態に関する関数も書き直しておく。
ファイル Cont.hs

```
instance MyState (KIO r) where
  get p   =
```

```
      KIO (\ c (s, i, o) -> c (fst (p s)) (s, i, o))
  set p v =
      KIO (\ c (s, i, o) -> c () (snd (p s) v, i, o))

instance MyStream (KIO r s) where
  readChar   =
      KIO (\ c (s, ch : i, o) -> c ch (s, i, o))
  eof        =
      KIO (\ c (s, i, o) -> c (null i) (s, i, o))
  writeStr v =
      KIO (\ c (s, i, o) -> c () (s, i, o ++ v))

abort :: (WithIO s -> r) -> KIO r s a
abort f = KIO (\ c -> f)
```

すると、`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。

また、`abort f` は現在の接続を無視して `f` 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（あるいはジャンプ）することに相当する。

ここで **goto**, **break**, **continue** について、変換前の Util と変換後の Haskell の対応は次の表のようになる。

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

つまり、 **goto** label と **break** はそれぞれ、現在の接続は無視して、 label と _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。

また **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。

ところで **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、翻訳がやや複雑になる。

ソース (Util)	ターゲット (Haskell)

while c **do** t	KIO (\ _break ->
	  let KIO _while = $[\![c]\!]$ >>= \ _b ->
	  if _b **then** $[\![t]\!]$ >>= \ _ ->
	  KIO _while
	  else return ()
	  in _while _break)

ここで、_break は [redacted] を表す接続で、_while _break は [redacted] 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）：

```
foo y = begin
```

```
int foo(int y) {
```

```
  xP := 1; yP := y;
  while get yP > 0 do
  begin
    val x = get xP in
    val y = get yP in
    if y == 10 then
     break
    else if y == 3 then
    begin
      yP := y - 1;
      continue
    end else ();
    xP := x * y;
    yP := y - 1
  end;
  get xP
end
```

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


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

は階乗の関数の変な変形であるが、これをコンパイルすると、次の Haskell プログラムが得られる。

```
foo = \ y ->
  set xP 1              >>= \ _ ->
  set yP y              >>= \ _ ->
  KIO (\ _break ->
    let KIO _while
        = get yP        >>= \ y ->
          if y > 0 then
              get xP            >>= \ x ->
              get yP            >>= \ y ->
              (if y == 10 then abort (_break ())
                 else if y == 3 then
                     set yP (y - 1)   >>= \ _ ->
                     abort (_while _break)
                  else return ()) >>= \ _ ->
              set xP (x * y)     >>= \ _ ->
              set yP (y - 1)     >>= \ _ ->
              KIO _while
            else return ()
      in _while _break)  >>= \ _ ->
  get xP
```

この `foo` の型は`Integer -> KIO 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`）、初期状態（`((0,0),"","")` など）を渡す必要がある。ここでそのための関数 `evalKIO` を

```
evalKIO m s = unKIO m (\ a s -> a) (s,"","")
```

と定義すると、`evalKIO (foo 9) (0,0)` の結果は、＿＿＿＿に、`evalKIO (foo 11) (0,0)` は＿＿になる。（参考: $9! = 362880$）

さらに **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、飛び先の**ラベル**に適切な接続を与える必要がある。つまり、ラベルの接続に変数として名前を与える、（再帰的に定義された）let 式として解釈する。一般的に説明するとかなり長くなってしまうので、例で示すにとどめることにす

る。 ここでは、ラベルが 3 つ使われている形を例として、翻訳前と翻訳後の形を示す。

ソース (Util)	ターゲット (Haskell)
begin	KIO (\ _end -> **let**
lbl1: s_1	lbl1 = \ _ -> unKIO ⟦s_1⟧ lbl2
lbl2: s_2	lbl2 = \ _ -> unKIO ⟦s_2⟧ lbl3
lbl3: s_3	lbl3 = \ _ -> unKIO ⟦s_3⟧ _end
end	**in** lbl1 ())

この 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）:

```
bar _ = begin
    xP := 1;
  label1:
    if get xP > 100
      then goto label2
      else ();
    xP := get xP * 2;
    goto label1;
  label2:
    get xP
  end
```

```
int 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 プログラムに翻訳される。

```
bar = \ _ ->
        KIO (\ _end ->
          let label1 = \ _ -> unKIO (
                get xP              >>= \ x ->
                (if x > 100 then abort (label2 ())
                   else return ()) >>= \ _ ->
                get xP              >>= \ x ->
                set xP (x * 2)      >>= \ _ ->
                abort (label1 ())) label2
              label2 = \ _ -> unKIO (get xP) _end
           in unKIO (set xP 1) label1)
```

そして、evalKIO (bar ()) (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) {
  int i = 1, sum = 0;
  while (i <= n) {
    sum = sum + i;
    if (sum > 21) {
      sum = 0;
      break;
    }
```

```
    i = i + 1;
  }
  return sum;
}
```

2.

```
int fuga(int n) {
  int i = 1, p = 1;
  while (i <= n) {
    if (i % 3 == 1) {
      i++;
      continue;
    }
    p = p * i;
    i++;
  }
  return p;
}
```

今日では構造化プログラミングが一般的になり、goto 文はほとんど使われることはない。上の翻訳結果からは、goto 文を多用したスパゲッティーコードは、結局、相互再帰（例えば、A の定義に B を利用し、B の定義に C を利用し、C の定義に A を利用するような状況）を多用することと同等であり、プログラムの意味がわかりにくくなると言える。

6.3 callcc とは

ここで紹介する callcc は Scheme や Ruby などが採用している、プログラマー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-with-current-continuation または、省略して call/cc と書く。（Scheme では "-" も "/" も関数名の一部として使えることに注意する。）

1 引数の関数 *f* に対して、callcc *f* のように使用すると＿＿＿＿＿＿を引数として、*f* を呼び出す。*f* 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。一方 *f* が接続を呼び出さなければ、*f* 自身の戻り値が callcc 式全体の戻り値になる。まず、トリビアルな例として Util の記法で

```
baz x = callcc (\ k ->
    100 + (if x >= 10 then x else k x))
```

Scheme では

```
(define (baz x)
  (call/cc (lambda (k)
    (+ 100 (if (=> x 10) x (k x))))))
```

という関数を考える。 ここで baz 10 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。

また 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。

```
multlist xs =
  let aux xs k = begin
        xP := 1; yP := xs;
        while not (null (get yP)) do
          val y = get yP in val n = head y in
          if n == 0 then k 0 else
            begin xP := get xP * n; yP := tail y;
                  writeStr " ";
                  write n
            end;
        get xP
    end in
  val result = callcc (\ k -> aux xs k) in begin
    writeStr "; result = ";
    write result
  end
```

Scheme では次のようになる。

```
(define (multlist xs)
   (call/cc (lambda (k)
      (define (aux xs)
         (if (null? xs)
            1
            (if (= 0 (car xs))
               (k 0)
               (begin
                    (display (car xs)) ;
                    (display " ")       ;
                  (* (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 全体は ＿＿ と出力する。

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。

注意: 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、`callcc` の引数の型と戻り値の型が同一になる必要があるからである。

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、

```
(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)))))
```

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。

ここでは `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。

また `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ーな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

```
callcc :: ((a -> KIO v s b) -> KIO v s a) -> KIO v s a
callcc h = KIO (
                                                )
-- KIO, unKIO がなければ
-- callcc h = \ c -> let k a = \ d -> c a
--                       in h k c
```

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

翻訳は単に callcc という名前の UtilCont の関数を Haskell の callcc に置き換えれば良い。

ソース (Util)	ターゲット (Haskell)
callcc m	⟦m⟧ >>= \ _x -> callcc _x

1 引数で副作用を持たない関数の head, tail, null, not, show などは、次のように翻訳される。

ソース (Util)	ターゲット (Haskell)
pure1 m	⟦m⟧ >>= \ _x -> return (pure1 _x)

例えば、先に紹介した UtilCont プログラム multlist を翻訳すると、次の Haskell プログラムが得られる。

```
multlist = \ xs ->
  let aux = \ xs ->
        return (\ k ->
          set xP 1                >>= \ _ ->
          set yP xs               >>= \ _ ->
          KIO (\ _break ->
             let KIO _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)) >>= \ _ ->
                       KIO _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、プログラムの出力を取り出すために、初期接続と初期状態を与える関数 evalKIO2 を

```

```

```
evalKIO2 m s = unKIO m (\ a (_,_,o) -> o) (s,"","")
```

と定義すると、 `evalKIO2 (multlist [1,2,3,4,5]) (0,[])` は、 `1 2 3 4 5; result = 120` となり、一方、`evalKIO2 (multlist [1,2,3,0,4,5]) (0,[])` 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、文献 (Reynolds 1993) 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。文献 (Filinski 1994) は、 `call/cc` がある意味で「オールマイティー」であることについての説明 を与えている。 文献 (Sekiguchi et al. 2001) 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ー を導入する方法を述べている。 文献 (Erköok & Launchbury 2000) は `mfixU` などの不動点演算子について解説している。

(Reynolds 1993) John C. Reynolds, “The Discoveries of Continuations” Lisp and Symbolic Computation, 6, (233–247). 1993 年

(Filinski 1994) Andrzej Filinski, “Representing Monads” 21st ACM Symposium on Principles of Programming Languages. 1994 年

(Sekiguchi et al. 2001) 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/

(Erköok & Launchbury 2000) Levent Erköok, and John Launchbury, “Recursive Monadic Bindings” Proc. of the International Conference on Functional Programming. 2000 年
