

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 (ジャンプなどが無いときに) 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。

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 ((a -> r) -> r)
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 (\ c -> c a)
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、 $a \rightarrow r$ 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 $(\backslash a \rightarrow k a c)$ 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
5           | App Expr Expr
6           -- ここまでは、Util1と同じ
7           | Begin [LabeledExpr]      -- ブロック
8           | Break                    -- break 文
9           | Continue                 -- continue 文
10          | Goto String               -- goto 文
11          deriving Show
12 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`

```

1 newtype KIO r s a
2   = KIO ((a -> WithIO s -> r) -> WithIO s -> r)
3
4 unKIO :: KIO r s a -> (a -> WithIO s -> r)
5         -> WithIO s -> r
6 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`

```

1 instance MyState (KIO r) where
2   get p    = KIO (\ c (s, i, o) -> c (fst (p s)) (s,
   i, o))
3   set p v  = KIO (\ c (s, i, o) -> c () (snd (p s) v,
   i, o))
4
5 instance MyStream (KIO r s) where
6   readChar = KIO (\ c (s, ch : i, o) -> c ch (s, i,
   o))
7   eof      = KIO (\ c (s, i, o) -> c (null i) (s, i,
   o))
8   writeStr v = KIO (\ c (s, i, o) -> c () (s, i, o ++
   v))
9
10 abort :: (WithIO s -> r) -> KIO r s a
11 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 lbl)      = mkGoto lbl "()"
2 comp Break          = mkGoto "_break" "()"
3 comp Continue       = mkGoto "_while" "_break"
4
5 mkGoto lbl v = TApp1 (TVar "abort")
6                (TApp1 (TVar lbl) (TVar v))

```

ここで `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)
<code>goto label</code>	<code>abort (label ())</code>
<code>break</code>	<code>abort (_break ())</code>
<code>continue</code>	<code>abort (_while _break)</code>

一方 `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。

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、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 "KIO")
4   (TLambda1 "_break"
5    (TLet [(PVar "_while", TApp1 (TVar "unKIO") body)]
6           (TApp1 (TVar "_while") (TVar "_break"))))
7   where body = comp e1 `TBind` TLambda1 "_b"
8             (TIf (TVar "_b") (comp e2 `TBind`
9                   TLambda1 "_"
10                  (TApp1 (TVar "KIO") (TVar "_while"))))
11             (TReturn (TVar "()))))

```

ソース (Util)	ターゲット (Haskell)
<code>while c do t</code>	<pre> KIO (\ _break -> let KIO _while = c >>= \ _b -> if _b then t >>= \ _ -> KIO _while else return () in _while _break) </pre>

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) :

<pre> 1 foo y = begin 2 xP := 1; yP := y; 3 while get yP > 0 do 4 begin 5 val x = get xP in 6 val y = get yP in 7 if y == 10 then 8 break 9 else if y == 3 then 10 begin 11 yP := y - 1; 12 continue 13 end else (); 14 xP := x * y; 15 yP := y - 1 16 end; 17 get xP 18 end </pre>	<pre> 1 int foo(int y) { 2 int x = 1; 3 while (y > 0) { 4 5 if (y == 10) 6 break; 7 else 8 if (y == 3) { 9 y--; 10 continue; 11 } 12 x = x * y; 13 y--; 14 } 15 return x; 16 } </pre>
---	--

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。

```

1 foo = \ y ->
2   set xP 1           >>= \ _ ->
3   set yP y           >>= \ _ ->
4   KIO (\ _break ->
5     let KIO _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 ()))
11            else 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            KIO _while
18            else return (())
19        in _while _break) >>= \ _ ->
20    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

```

1 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、“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	KIO (\ _end -> let
lb11: s ₁	lb11 = \ _ -> unKIO s ₁ lb12
lb12: s ₂	lb12 = \ _ -> unKIO s ₂ lb13
lb13: s ₃	lb13 = \ _ -> unKIO s ₃ _end
end	in lb11 (())

この s₁, s₂, s₃ の中には、goto lb11, goto lb12, goto lb13 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 lb11, lb12, lb13 に束縛されているのはそれぞれ、同名のラベル lb11, lb12, lb13 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）:

<pre> 1 bar _ = begin 2 xP := 1; 3 labell1: 4 if get xP > 100 5 then goto label2 6 else (); 7 xP := get xP * 2; 8 goto labell1; </pre>	<pre> 1 int bar(void) { 2 int x = 1; 3 labell1: 4 if (x > 100) 5 goto label2; 6 7 x = x * 2; 8 goto labell1; </pre>
---	--

9	label2:	9	label2:
10	get xP	10	return x;
11	end	11	}

は次のような Haskell プログラムにコンパイルされる。

```

1 bar = \ _ ->
2     KIO (\ _end ->
3         let label1 = \ _ -> unKIO (
4             get xP          >>= \ x ->
5                 (if x > 100 then abort (label2 ())
6                   else return ()) >>= \ _ ->
7             get xP          >>= \ x ->
8             set xP (x * 2)   >>= \ _ ->
9             abort (label1 ())) label2
10        label2 = \ _ -> unKIO (get xP) _end
11    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) {
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 }

```

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。つまり goto 文を多用したスパゲッティコードは、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

```
1 baz x = callcc (\ k ->
2   100 + (if x >= 10 then x else 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。

```
1 multlist xs =
2   let aux xs k = begin
3     xP := 1; yP := xs;
4     while not (null (get yP)) do
5       val y = get yP in val n = head y in
6       if n == 0 then k 0 else
7         begin xP := get xP * n; yP := tail y;
8           writeStr " ";
9           write n
10        end;
11      get xP
12    end in
13  val result = callcc (\ k -> aux xs k) in begin
14    writeStr "; result = ";
15    write result
16  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 writeStr " i:"; write n;
4   increase (n + 1) (callcc k) end;
5 decrease n k =
6   if n < 0 then ()
7   else begin writeStr " d:"; write n;
8   decrease (n - 1) (callcc k) end

```

という2つの関数を定義して

```

1 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、

```

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

```

1 (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](#)

```

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

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 multlist = \ xs ->
2   let aux = \ xs ->
3     return (\ k ->
4       set xP 1 >>= \ _ ->
5       set yP xs >>= \ _ ->
6       KIO (\ _break ->
7         let KIO _while =
8           get yP >>= \ y ->
9           if not (null y) then
10            get yP >>= \ y ->
11            (if head y == 0 then k 0 else
12             get xP >>= \ x ->
13             set xP (x * head y)
14             >>= \ _ ->
15             set yP (tail y) >>= \ _ ->
16             writeStr " " >>= \ _ ->
17             write (head y) >>= \ _ ->
18             KIO _while

```

```

19         else return ()
20         in _while _break) >>= \ _ ->
21         get xP)
22     in callcc (\ k -> aux xs >>= \ _f ->
23         _f k) >>= \ result ->
24         writeStr "; result = " >>= \ _ ->
25         write result

```

このとき、プログラムの出力を取り出すために

```

1 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 (Erkook & 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/>

(Erkook & Launchbury 2000) Levent Erkook, and John Launchbury, "Recursive Monadic Bindings" *Proc. of the International Conference on Functional Programming*. 2000 年