

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:

```
1 int main(void) {
2     printf("The result is %d.\n", 1 + fact(10));
3     return 0;
4 }
```

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。

```
1 int baz(int n) {
2     int x = 0, y = 1;
3     label1:
4     if (x > n) goto label2;
5     x += y;
6     y++;
7     goto label1;
8     label2:
9     return x;
10 }
```

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`

```
1 newtype K r a = K (
2
3 unK :: K r a -> (a -> r) -> r
4 unK (K c) = c
5
6 instance Applicative (K r) where
7   pure a = K (
8     (<*>) = ap
9
10 instance Monad (K r) where
11   (K m) >>= k = K (
12     -- K, unK がなければ、
13     -- 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`

```
1 newtype KIO r s a
2   = KIO (
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 =
3     KIO (\ c (s, i, o) -> c (fst (p s)) (s, i, o))
4   set p v =
5     KIO (\ c (s, i, o) -> c () (snd (p s) v, i, o))
6
7 instance MyStream (KIO r s) where
8   readChar =
9     KIO (\ c (s, ch : i, o) -> c ch (s, i, o))
10  eof =
11    KIO (\ c (s, i, o) -> c (null i) (s, i, o))
12  writeStr v =
13    KIO (\ c (s, i, o) -> c () (s, i, o ++ v))
14
15 abort :: (WithIO s -> r) -> KIO r s a
16 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	<code>abort (label ())</code>
break	<code>abort (_break ())</code>
continue	<code>abort (_while _break)</code>

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 <i>c</i> do <i>t</i>	<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）：

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

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。ここでそのための関数 evalKIO を

```

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、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

一般的に説明するとか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
lb11: s_1	lb11 = \ _ -> unKIO $\llbracket s_1 \rrbracket$ lb12
lb12: s_2	lb12 = \ _ -> unKIO $\llbracket s_2 \rrbracket$ lb13
lb13: s_3	lb13 = \ _ -> unKIO $\llbracket s_3 \rrbracket$ _end
end	in lb11 ())

この s_1, s_2, s_3 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）：

1 bar _ = begin	1 int bar(void) {
2 xP := 1;	2 int x = 1;
3 label1:	3 label1:
4 if get xP > 100	4 if (x > 100)
5 then goto label2	5 goto label2;
6 else ();	6
7 xP := get xP * 2;	7 x = x * 2;
8 goto label1;	8 goto label1;
9 label2:	9 label2:
10 get xP	10 return x;
11 end	11 }

は次のような Haskell プログラムに翻訳される。

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. 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 }

```

2.

```

1 int fuga(int n) {
2     int i = 1, p = 1;
3     while (i <= n) {
4         if (i % 3 == 1) {
5             i++;
6             continue;
7         }
8         p = p * i;
9         i++;
10    }
11    return p;
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。上の翻訳結果からは、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))

```

Scheme では

```

1 (define (baz x)
2   (call/cc (lambda (k)
3     (+ 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。

もちろん、このような非実用的な例だけならば、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 の本当の価値は普通でな

い制御構造を一般ユーザーが定義できるところにある。実際の `callcc` の使用例については Scheme を使って説明する。

6.4 `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。
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 (
3
4   -- KIO, unKIO が必要
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。

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)
<code>callcc m</code>	<code>[[m]] >>= \ _x -> callcc _x</code>

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)
<code>purel m</code>	<code>[[m]] >>= \ _x -> return (purel _x)</code>

6.5 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、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
