

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.", 1+fact(10));
}
```

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。

```
data Expr = Const Target | Var String | If Expr Expr Expr | While Expr Expr
          | Let [Decl] Expr | Val Decl Expr
          | Lambda String Expr | Delay Expr | App Expr Expr
-- ここまでは、Util1と同じ
          | Begin [LabeledExpr]      -- ブロック
          | Break                    -- break 文
          | Continue                  -- continue 文
          | Goto String               -- goto 文
          deriving Show

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。

```
type K r a = _____  
unitK :: a -> K r a  
unitK a = _____  
bindK :: K r a -> (a -> K r b) -> K r b  
m 'bindK' k = _____  
abortK :: r -> K r a  
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。

```
type KST s v a = _____  
{- = (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)  
  
putStrK :: String -> KST (s, i, String) any ()  
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` の定義は以下ようになる。

```

comp (Goto l)          = mkGoto l
comp Break            = mkGoto "_break"
comp Continue        = TApp1 (TVar "abortK")
                      (TApp1 (TVar "_while") (TVar "_break"))

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。

```

comp (While e1 e2)    = compWhile e1 e2

compWhile e1 e2 = TLambda1 "_break"
                 (TLet [(PVar "_while", body)]
                      (TApp1 (TVar "_while") (TVar "_break")))
  where body = comp e1 'TBindM' TLambda1 "_b"
              (TIf (TVar "_b") (comp e2 'TBindM' TLambda0 (TVar "_while"))
                  (TUnitM (TVar "()")))

```

ソース (Util)	ターゲット (Haskell)
while c do t	<pre> _break -> let _while = c 'bindM' _b -> if _b then t 'bindM' _ -> _while else () 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):

```

foo = \ y -> begin
  setM xP 1; setM yP y;
  while getM yP > 0 do begin
    val x = getM xP in val y = getM yP in
    if y==10 then break
    else if y==3 then begin
      setM yP (y-1); continue
    end else ();
    setM xP (x*y);
    setM yP (y-1)
  end;
  getM xP
end

```

```

int foo(int y) {
  int x=1;
  while (y>0) {
    if (y==10) break;
    else if (y==3) {
      y--; continue;
    }
    x=x*y;
    y--;
  }
  return x;
}

```

をコンパイルすると、次の Haskell プログラムが得られる。

```

foo = \ y ->
  setK xP 1          'bindK' \ _ ->
  setK yP y          'bindK' \ _ ->
  (\ _break ->
    let _while
      = getK yP      'bindK' \ y ->
        if y > 0 then
          getK xP    'bindK' \ x ->
          getK yP    'bindK' \ y ->
          (if y == 10 then abortK (_break ()) else
            if y == 3 then
              setK yP (y-1) 'bindK' \ _ ->
              abortK (_while _break)
            else unitK ()) 'bindK' \ _ ->
          setK xP (x*y) 'bindK' \ _ ->
          setK yP (y-1) 'bindK' \ _ ->
          _while
            else unitK ()
          in _while _break) 'bindK' \ _ ->
    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, "", "")) 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)
<pre> begin label1: s₁ label2: s₂ label3: s₃ end </pre>	<pre> \ _end -> let label1 = \ () -> s'₁ label2 label2 = \ () -> s'₂ label3 label3 = \ () -> s'₃ _end in label1 () </pre>

s₁, s₂, s₃ 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):

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

```
void 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 プログラムにコンパイルされる。

```
bar = \ _ ->
  setK xP 1 'bindK' \ _ ->
  \ _end ->
  let label1 = \ _ ->
    (getK xP
     (if x > 100 then abortK (label2 ())
      else unitK ())) 'bindK' \ x ->
    getK xP 'bindK' \ _ ->
    setK xP (x*2) 'bindK' \ x ->
    abortK (label1 ())) label2 'bindK' \ _ ->
  label2 = \ _ -> getK xP _end
  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、`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、そのとき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、

```
baz = \ x ->
  callccM (\ k ->
    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。

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。

```
callccK :: ((a -> K r b) -> K r a) -> K r a
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。

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 callcc (\ 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。

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
接続に関する文献は数多くあるが、この論文は接続の「発見」について、振り返っている珍しいものである。
- [2] Andrzej Filinski, 「Representing Monads」
21st ACM Symposium on Principles of Programming Languages. 1994 年
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] 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/>
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] Levent Erkök, and John Launchbury, 「Recursive Monadic Bindings」
Proc. of the International Conference on Functional Programming. 2000 年
mfixUなどの不動点演算子について解説している。