第8章　Continuation-Passing Style (CPS)

この章では接続の概念の応用を説明する。

8.1　Continuation-Passing Style とは

Continuation-Passing Style（CPS）とは ______________________________ ____プログラムの書き方のことである。次のような使い途がある。

- call/cc のない言語でコルーチンなど ______________________を実現したいときに用いる
- プログラムを効率の良い形に変換したいときに、変換の途中の中間形式で用いる

また、JavaScript で非同期の関数を呼び出すときには、CPS でプログラムを書かざるを得ないときもある。

CPS のプログラムは次のような制限に従う。

- 関数呼び出しが ____________________。（つまり、関数呼び出しの引数は関数呼び出し[1]になっていることがない。）

例えば、

```
function prodPrimes(n) {
  if (n==1) return 1;
  else if (isPrime(n)) return n * prodPrimes(n-1);
  else return prodPrimes(n-1);
}
```

という関数を考える。これは 1 から n までの範囲に存在する素数の積を求める関数である。isPrime は素数かどうかを判定する関数とする。これを、CPS 変換すると、次のような関数 prodPrimesC に変換される。（ここには定義を示していないが、isPrime を CPS 変換した関数を isPrimeC とする。）

[1]ただし、+や*のようなプリミティブな関数の呼び出しは除く。

```
1  function prodPrimesC(n, c) {
2    if (n==1) return c(1);
3    else return isPrimeC(n, function (b) {
4      if (b)
5         return prodPrimesC(n-1,
6                            _____________________________________);
7      else return prodPrimesC(n-1, c);
8         });
9  }
```

（JavaScript の記法で紹介しているが、他のプログラミング言語でも同様の変換は可能である。）isPrimeC を呼び出すときに、戻ってきたときに行なうべき処理を接続:

```
function (b) {
   if (b)
      return prodPrimesC(n-1, function (p) { return c(n*p); });
   else return prodPrimesC(n-1, c);
}
```

として isPrimeC に渡している。さらにこの接続の中で、prodPrimesC を呼び出すときに、b の値に応じて、n を掛けてから c に渡すという接続: **function** (p) { **return** c(n*p); }、またはもとのままの接続である c を渡している。

CPS はコンパイラの中間言語として用いられることがある。これは関数の呼び出しの順番が明確になり、関数の呼び出しを単なるジャンプ命令で実現して良いという性質があるからである。

プログラムを CPS に変換するには、だいたい次のような手順で行なう。

1. すべての関数定義に _____を一つ追加する
   function prodPrimes(n) {...} $\Longrightarrow$ **function** prodPrimesC(n, c) {...}
2. 関数の戻り値に相当する位置にある単純な式は、_____。（ここで**単純な式**とは ···
   定数、変数、ラムダ式、プリミティブオペレータ（-, == など）を単純な式に適用した式、のいずれか）
   ... **return** 1; ... $\Longrightarrow$ ... **return** c(1); ...
3. 関数の戻り値に相当する位置にある（単純な式でない）関数適用は、_____。
   ... **return** prodPrimes(n-1); ... $\Longrightarrow$ ... **return** prodPrimesC(n-1, c)
4. その他の位置にある（単純な式でない）関数適用は、“適切[2]な” 接続を明示的に受け取る形に変換する。

[2]“適切な” 接続の正確な定義をここで与えることは断念する。要するに元のプログラムと意味が変わらず、CPS 変換を施す目的が達成できれば良い。

```
... return n*prodPrimes(n-1); ...
⟹ ... return prodPrimesC(n-1, function (p) { return c(n*p);
}) ...
```

正式な CPS 変換の定義は、UtilCont に対する変換 comp そのものである。つまり、UtilCont を Haskell にコンパイルし、unitM を "\ a c -> c a"、bindM を "\ c -> m (\ a -> k a c)" に置き換え、さらに見易いかたちにするための β 変換を実施すれば CPS 変換になる。ターゲット言語は Haskell でなくても、ラムダ式を持っていれば良いので、UtilCont から UtilCont への CPS 変換と見なすことも出来る。あるプログラミング言語（例えば JavaScript）が UtilCont と同等の制御構造を持っていれば、JavaScript と UtilCont の間の変換を考えて、JavaScript から JavaScript への CPS 変換を考えることも出来る。

8.2 CPS の応用—再帰呼出しの繰返しへの変換

CPS を利用してプログラムの変換を行なうことがある。例として再帰的関数を CPS を経由して繰返しへ変換する場合を取り上げる。

変換の対象は、次のように定義された階乗の関数である。

```
1  function fact(n) {
2    if (n==0) return 1;
3    else return n*fact(n-1);
4  }
```

これは数学的な記法の定義:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \quad (n > 0) \end{aligned}$$

に直接対応していてわかりやすいが、実行時には n に比例するスタック領域が必要にある。

この fact を CPS に変換すると次のようなプログラムになる。

```
1  function fact(n, c) {
2    if (n==0) return _____;
3    else
4      return fact(n-1, ____________________________________________);
5  }
```

さらに、これは末尾再帰なので、次のように繰返しに書き換えることができる。

```
1  function aux(n, c) { return function (r) { return c(n*r); }; }
2
3  function fact(n, c) {
4    while(n>0) {
5      c = aux(n, c);  n--; // 注[3]
6    }
7    return c(1);
8  }
```

繰返しに変換されたが、c がどんどん大きくなってしまうので、領域の節約にはならない。しかし、良く観察すると c は常に次のような形の関数であることがわかる。

つまり、fact の場合、第 2 引数として本当の接続を受け渡さなくても、この n*(n-1)* ... *m で接続を表現可能ということである。このことを考慮に入れてさらにプログラムを変換すると、次の定義が得られる。

```
1  function fact(n, m) {
2    while(n>0) {
3      __________ n--;
4    }
5    return m;
6  }
```

これは、通常の繰返しによる階乗関数の定義である。このように非末尾再帰を除去する場合、まず CPS に変換して末尾再帰のかたちにし、それから “接続” を同等のオブジェクトに入れ換えるとよい。

8.3 CPS の応用—Web プログラミング

Servlet や JavaScript などで WWW 上のインタラクティブなアプリケーションを作成するときに、プログラムの任意の場所でユーザの入力を待って、続きから実行するという書き方ができない（必ず doGet などの関数のはじめから実行されてしまう）という制約がある。

そこで、インタラクティブなプログラムを実現するために、さまざまなテクニックが必要になるが、CPS への変換はある意味でオールマイティな（つまり、どんな場合にも適用可能な）手段である[4]。

トリッキーな例として JavaScript のハノイの塔のプログラム:

```
1  function move(n, a, b) {      // 非 CPS 版
2    document.form.textarea.value
3      += ("move␣"+n+"␣from␣"+a+"␣to␣"+b);
4  }
5
6  function hanoi(n, a, b, c) { // 非 CPS 版
7    if (n>0) {
8      hanoi(n-1, a, c, b);
9      move(n, a, b);
10     hanoi(n-1, c, b, a);
11   }
12 }
```

[3] ここは c = function (r) { return c(n*r); }; と書くことはできない。JavaScript のセマンティクスでは、右辺の変数 c の値も変わってしまうからである。aux 関数を介すると c の値がコピーされるため安全である。

[4] もちろん、言語に最初から call/cc が用意されていれば、このような面倒なことをする必要がない。

を「ボタンを押したら 1 行表示する」というバージョンに書き換える、ということを考える。つまり、

```
<script type="text/javascript">
function move(n, a, b) {  // formの TextAreaに追加する。
  document.form.textarea.value
     += ("move␣"+n+"␣from␣"+a+"␣to␣"+b+"\n");
}
</script>

<form name="form">
<input type="button" onClick="exec()" value="実行"><br>
<textarea name="textarea" cols="20" rows="32"></textarea>
</form>
```

というフォームの「実行」ボタンを押せばテキストエリアに 1 行表示するようにする。

まず、hanoi を CPS に書き換える。

```
function move(n, a, b, k) {     // 暫定版（説明用）
  document.form.textarea.value
    += ("move␣"+n+"␣from␣"+a+"␣to␣"+b+"\n");
  return k();
}

function hanoi(n, a, b, c, k) { // 最終版
  if (n>0) {
    return hanoi(n-1, a, c, b,
                 function () {
                   return move(n, a, b,
                               function() {
                                 return hanoi(n-1, c, b, a, k);
                               });
                 });
  } else {
    return k();
  }
}
```

しかし、ここで、

```
function exec() {               // 暫定版（説明用）
  hanoi(5, 'a', 'b', 'c', function () { return null; });
}
```

のように、hanoi を呼び出しても、これまで通り一気に最後まで出力してしまうだけである。そこで move を次のように書き換える。

```
function move(n, a, b, k) {     // 最終版
  document.form.textarea.value
    += ("move␣"+n+"␣from␣"+a+"␣to␣"+b+"\n");
  return _;     // ____ ではない。
}
```

つまり、最後に接続を呼び出してしまわず、いったん呼び出し側に接続を戻り値として返す。（このような手法をトランポリンと言う。）これで call/cc と同じような接続を明示的に扱う効果が得られる。この接続を利用するために exec を次のように書き換える。

```
1  function doEnd() {                // 最終版
2    document.form.textarea.value += "end\n"; // 最後の処理
3    return doEnd;
4  }
5
6  // 最初のエントリポイント
7  var k = function() { return hanoi(5, 'a', 'b', 'c', doEnd); };
8
9  function exec() {                 // 最終版
10   __________
11 }
```

exec は k () の実行結果を新しい k の値として保存するだけである。これで「実行」ボタンを押すたびに move が 1 回ずつ実行されるようになる。

問 8.3.1 上のやり方にならって、次の関数を「ボタンを押したら 1 行表示する」というバージョンに書き換えよ。

```
1  function showArgument(m) {
2    document.form.textarea.value += ("argument␣=␣"+m);
3  }
4
5  function showResult(m, r) {
6    document.form.textarea.value
7      += ("result␣for␣argument:␣"+m+"␣=␣"+r);
8  }
9
10 function fib(m) {
11   showArgument(m);
12   var r;
13   if (m<2) {
14     r=1;
15   } else {
16     r=fib(m-1)+fib(m-2);
17   }
18   showResult(m, r);
19   return r;
20 }
```

接続の表現　JavaScript は匿名関数（ラムダ式）を持っているため、CPS への変換は比較的容易であったが、ラムダ式を持たない言語や効率を重視する場合では、__________を明示的に使用し、そのなかに接続に対応するデータを格納する必要がある。次のプログラムは、接続を n, a, b, c の各パラメータと次に実行を開始すべき場所（pc）から構成されるデータとして表現したものである。

```
1  function move(n, a, b) {          // 明示スタック版
2    document.form.textarea.value
3      += ("move␣"+n+"␣from␣"+a+"␣to␣"+b+"\n");
4  }
5
6  var stack = new Array();
7  stack.push(new Array(5, 'a', 'b', 'c', 0));
8
9  function hanoi(n, a, b, c, pc) { // 明示スタック版
10   while(n>0) {
11     switch (pc) {
12     case 0:
13       stack.push(new Array(n, a, b, c, 1));
14       var tmp=c; c=b; b=tmp; n--;
15       continue;
16     case 1:
17       stack.push(new Array(n-1, c, b, a, 0));
18       move(n, a, b);
19       return;
20     }
21   }
22   return exec();
23 }
24
25 function exec() {                 // 明示スタック版
26   if(stack.length>0) {
27     var args=stack.pop();
28     hanoi(args[0], args[1], args[2], args[3], args[4]);
29   } else {
30     document.form.textarea.value += "end\n";
31   }
32 }
```

ここまでやってしまうとプログラムの実行途中で “接続” をファイルに保存したり、別のコンピュータで起動することさえ可能になる。

8.4 さらに詳しく知りたい人のために ...

[1] は接続と CPS に関する重宝なリンク集のページである。

この章の参考文献

[1] Untyped Ltd. 「Continuations and Continuation Passing Style」http://library.readscheme.org/page6.html