


```

4     if (b)
5         return prodPrimesCPS(n-1,
6                                     _____);
7     else return prodPrimesCPS(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。)

```
prodPrime(n) = prodPrimeCPS(n, function (x) { return x; })
```

という関係が成り立つ。isPrimeCPS 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 prodPrimesCPS(n-1, function (p) { return c(n*p); });
  else return prodPrimesCPS(n-1, c);
}

```

として isPrimeCPS に渡している。さらにこの接続の中で、prodPrimesCPS 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 はコンパイラの間言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 _____ (空欄 8.1.4) を一つ追加する
`function prodPrimes(n) { ... } ⇒ function prodPrimesCPS(n, c) { ... }`
2. 関数の戻り値に相当する位置にある単純な式は、**接続に渡す**。(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; ... ⇒ ... 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、**接続を引数として渡す**。
`... return prodPrimes(n-1); ... ⇒ ... return prodPrimesCPS(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。要するに恒等関数 ($\lambda x.x$) を接続として渡されたとき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 prodPrimesCPS(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、`return` を “`\ a c -> c a`”、`(>>=)` 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。主な部分を `return` と `(>>=)` を置き換えた上で、再構成すると次の表のようになる。この表のなかでソース中で *Italic* フォントで示されている m, n などは任意の Util の式で、ターゲット中で m', n' のように ' (プライム) が付いている式は、その CPS 変換後の式を表す。

ソース (Util)	ターゲット (CPS)
<code>x</code> (ただし <code>x</code> は定数・変数)	<code>\ _c -> _c x</code>
<code>val x = m in n</code>	<code>\ _c -> m' (\ x -> n' _c)</code>
<code>f a</code>	<code>\ _c -> f' (\ _g -> a' (\ _x -> _g _x _c))</code>
<code>\ x -> m</code>	<code>\ _c -> _c (\ x -> m')</code>
<code>begin</code> <code> s;</code> <code> t;</code> <code> u</code> <code>end</code>	<code>\ _c -> s' (\ _ -> t' (\ _ -> u' _c))</code>

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 の間の変換を介して、JavaScript から JavaScript への CPS 変換を考えることが出来る。

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 factCPS(n, c) {
2   if (n==0) return ____;
3   else
4     return factCPS(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 factCPS(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いうので、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。

```
function (r) { return n*(n-1)*...*m*r; }
```

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 $n*(n-1)* \dots *m$ 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 factCPS(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置き換えるとよい。

8.3 CPS の応用—Web プログラミング

Servlet や JavaScript などでも 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

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。

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 }
```

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、

```
1 <form name="form">
2 <input type="button" onClick="exec()" value="実行"><br>
3 <textarea name="textarea" cols="20" rows="32"></textarea>
4 </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。

```
1 function moveCPS(n, a, b, k) { // 暫定版(説明用)
2   document.form.textarea.value
3   += ("move_" + n + "_from_" + a + "_to_" + b + "\n");
4   return k();
5 }
6
7 function hanoiCPS(n, a, b, c, k) { // 最終版
8   if (n > 0) {
9     return hanoiCPS(n - 1, a, c, b, function () {
10      return moveCPS(n, a, b, function () {
11       return hanoiCPS(n - 1, c, b, a, k);
12      });
13    });
14  } else {
15    return k();
16  }
17 }
```

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。

しかし、ここで、

```
1 function exec() { // 暫定版 (説明用)
2   hanoiCPS(5, 'a', 'b', 'c', function () { return; });
3 }
```

のように、hanoiCPS 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 moveCPS を次のように書き換える。

```
1 function moveCPS(n, a, b, k) { // 最終版
2   document.form.textarea.value
3     += ("move_" + n + "_from_" + a + "_to_" + b + "\n");
4   return _; // ____ ではない。
5 }
```

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 hanoiCPS(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。

もう一つの例として、次のフィボナッチ数列を計算する関数を CPS 化してみる。

```
1 function showArgument(m) { // 非 CPS 版
2   document.form.textarea.value += ("argument_" + m);
3 }
4
5 function showResult(m, r) { // 非 CPS 版
6   document.form.textarea.value
7     += ("result_for_argument:" + m + "_=" + r);
8 }
9
10 function fib(m) { // 非 CPS 版
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 }
21
22 function exec() { fib(5); }

```

このプログラムは、計算途中の引数と戻り値を表示するようになっている。まずは、非 CPS 版と意味が変わらない、途中で止まらないバージョンを作成する。fib は戻り値を持つので、接続は値を受け取る必要がある。

```

1  function showArgumentCPS(m, k) { // 暫定版
2      document.form.textarea.value += ("argument_=_=" + m + "\n");
3      return k();
4  }
5
6  function showResultCPS(m, r, k) { // 暫定版
7      document.form.textarea.value
8          += ("result_for_argument:_=" + m + "_=" + r + "\n");
9      return k();
10 }
11
12 function fibCPS(m, k) { // 最終版
13     return showArgumentCPS(m, function () {
14         function tmp(r) {
15             return showResultCPS(m, r, function() {
16                 return k(r);
17             });
18         }
19         if (m<2) {
20             return tmp(1);
21         } else {
22             return fibCPS(m-1, function(r1) {
23                 return fibCPS(m-2, function(r2) {
24                     return tmp(r1+r2);
25                 });
26             });
27         }
28     });
29 }
30
31 function doEnd(n) { // 暫定版
32     document.form.textarea.value
33         += "final_result_is_" + n + "_end\n";
34 }
35
36 function exec() { fibCPS(5, doEnd); }

```

これをハノイの塔のときと同じテクニックを使って途中で止まるように、プログラムを書き換える。

```

1  function showArgumentCPS(m, k) { // 最終版

```

```

2     document.form.textarea.value += ("argument_=_=" + m + "\n");
3     return k; // k() ではない
4 }
5
6     function showResultCPS(m, r, k) { // 最終版
7         document.form.textarea.value
8             += ("result_for_argument:_=" + m + "_=_=" + r + "\n");
9         return k; // k() ではない
10    }
11
12    // fibCPS は変更なし
13
14    function doEnd(n) { // 最終版
15        document.form.textarea.value
16            += "final_result_is_=" + n + "_end\n";
17        return function () { return doEnd(n); };
18    }
19
20    var k = function() { return fibCPS(5, doEnd); };
21    function exec() { k = k (); }

```

これで、1行表示するたびに停止する。

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には、(空欄 8.3.1) 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ハノイの塔を表現したものである。

```

1     // move は非 CPS 版と同じなので省略
2
3     var stack = new Array();
4     stack.push(new Array(5, 'a', 'b', 'c', 0));
5
6     function hanoiStack(n, a, b, c, pc) { // 明示スタック版
7         while(n>0) {
8             switch (pc) {
9                 case 0:
10                stack.push(new Array(n, a, b, c, 1));
11                var tmp=c; c=b; b=tmp; n--;
12                continue;
13                case 1:
14                stack.push(new Array(n-1, c, b, a, 0));
15                move(n, a, b);
16                return;
17            }
18        }
19        return exec();
20    }
21

```

```
22 function exec() { // 明示スタック版
23     if(stack.length>0) {
24         var args=stack.pop();
25         hanoiStack(args[0], args[1], args[2], args[3], args[4]);
26     } else {
27         document.form.textarea.value += "end\n";
28     }
29 }
```

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 ...

文献 [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>