第7章 Continuation-Passing Style (CPS)

この章では接続の概念の応用を説明する。

7.1 Continuation-Passing Styleとは

Continuation-Passing Style（CPS）とは常に関数に接続（に相当するもの）を引数として受け渡す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 (`XMLHttpRequest` の `send` など) を呼び出すときには、CPS に準じてプログラムを書かざるを得ないときもある。（なお ECMAScript 2017 以降では、`async`, `await` を用いて非同期の処理が書き易くなった。）

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 (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。)になっていることがない。）だから、結果が評価順によらない

CPS 変換とは、接続として恒等関数（`\ x -> x`）を渡したときに、意味が同じになるような CPS のプログラムに変換することである。例えば、
ファイル ProdPrimes.js

```
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。ここで `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 `prodPrimesCPS` 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 `isPrimeCPS` とする。）
ファイル ProdPrimes.js

```
function prodPrimesCPS(n, c) {
  if (n == 1) return c(1);
```

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

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

という関係が成り立つ。 ここで `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 はコンパイラー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) { … }` ⇒ **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ー（「-」, 「==」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、 "適切な" (("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。))接続を明示的に受け取る形に変換する。
   … **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、ただし通常は`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 簡約を実施すれば 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 $\mathring{m}, \mathring{n}$ のように ˚（ドット）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)
`x`（ただし`x`は定数・変数）	`\ _c -> _c x`
val `x =` *m* **in** *n*	`\ _c ->` $\mathring{m}$ `(\ x ->` $\mathring{n}$ `_c)`
f a	`\ _c ->` $\mathring{f}$ `(\ _g ->` `  `$\mathring{a}$ `(\ _x ->` `  _g _x _c))`
`\ x ->` *m*	`\ _c -> _c (\ x ->` $\mathring{m}$`)`
if *b* **then** *t* **else** *e*	`\ _c ->` $\mathring{b}$ `(\ _b ->` `  **if** _b **then** `$\mathring{t}$ `_c` `  **else** `$\mathring{e}$ `_c)`
begin *s;* *t;* *u* **end**	`\ _c ->` $\mathring{s}$ `(\ _ ->` `  `$\mathring{t}$ `(\ _ ->` `  `$\mathring{u}$ `_c))`

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。（関数呼出しがネストしないので、ターゲット言語の評価戦略が遅延評価か先行評価かは関係ない。）

ソース (JavaScript)	ターゲット (JavaScript)
`x`（ただし`x`は定数・変数）	**function** `(_c) { return _c(x); }`
var `x =` *m;* *n*	**function** `(_c) {` `  **return** $\mathring{m}$`(`**function** `(x) {` ` **return** $\mathring{n}$`(_c);` `  });` `}`
f(*a*)	**function** `(_c) {` `  **return** $\mathring{f}$`(`**function** `(_g) {` ` **return** $\mathring{a}$`(`**function** `(_x) {`

	`return _g(_x)(_c);` `});` `});` `}`
`function (x) { m }`	`function (_c) {` `  return _c(function (x) {` `  return m̊;` `  });` `}`
`if (b) { t } else { e }`	`function (_c) {` `  return b̊(function (_b) {` `  if (_b) {` `  return t̊(_c);` `  } else {` `  return e̊(_c);` `  }` `  });` `}`
`s;` `t;` `return u;`	`function (_c) {` `  return s̊(function (_) {` `  return t̊(function (_) {` `  return ů(_c);` `  });` `  });` `}`

7.2 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:

$$0! = 1$$
$$n! = n \times (n-1)! \quad (n > 0)$$

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。
ファイル Fact.js

```
1 function factCPS(n, c) {
2   if (n == 0) return c(1);
3   else
4     return factCPS(n - 1,
5                    function (r) { return c(n * r); });
6 }
```

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。
ファイル Fact.js

```
function aux(n, c) {
  return function(r) {
    return c(n * r);
  };
}

function factCPS(n, c) {
  while (n > 0) {
    c = aux(n, c);  n--; // 注†
  }
  return c(1);
}
```

†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 `n, c` の値も変わってしまうからである。`aux` 関数を介すると `n, c` の値がコピーされるため安全である。

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。

```
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) * … * 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。
ファイル Fact.js

```
function factCPS(n, m) {
  while (n > 0) {
    m *= n; n--;
  }
  return m;
}
```

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、繰り返しに書き換え、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。

7.3 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ーの入力を待って、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ーな（つまり、どんな場合にも適用可能な）手段である（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。）。

トリッキーな例として JavaScript のハノイの塔のプログラム:
ファイル Hanoi0.js

```
function move(n, a, b) {      // 非 CPS 版
  document.form.textarea.value
    += ("move " + n + " from " + a + " to " + b);
  return 0; // 形をそろえるため 0 を return する
}

function hanoi(n, a, b, c) { // 非 CPS 版
  if (n > 0) {
    hanoi(n - 1, a, c, b);
    move(n, a, b);
    hanoi(n - 1, c, b, a);
  }
  return 0;
}
```

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、

```
<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。
ファイル Hanoi.js

```
function moveCPS(n, a, b, k) {      // 暫定版（説明用）
  document.form.textarea.value
    += ("move " + n + " from " + a + " to " + b +
"\n");
  return k(0);
}

function hanoiCPS(n, a, b, c, k) { // 最終版
  if (n > 0) {
    return hanoiCPS(n - 1, a, c, b, function(ignore) {
      return moveCPS(n, a, b, function(ignore) {
        return hanoiCPS(n - 1, c, b, a, k);
```

```
      });
    });
  } else {
    return k(0);
  }
}
```

しかし、ここで、

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

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。

```
function moveCPS(n, a, b, k) {      // 最終版
  document.form.textarea.value
    += ("move " + n + " from " + a + " to " + b +
"\n");
  return k;      // k(0) 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らしい。）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。

```
function doEnd(n) {                 // 最終版
  document.form.textarea.value += "end\n"; // 最後の処理
  return doEnd;
}

// 最初のエントリーポイント
var restart = function(ignore) {
  return hanoiCPS(5, 'a', 'b', 'c', doEnd);
};

function exec() {                   // 最終版
  restart = restart(0);
}
```

この exec は restart(0) の実行結果を新しい restart 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。
ファイル Fib0.js

```
function showArgument(m) {      // 非 CPS 版
  document.form.textarea.value += ("argument = " + m);
  return 0;
}

function showResult(m, r) {     // 非 CPS 版
  document.form.textarea.value
    += ("result for argument: " + m + " = " + r);
  return 0;
}

function fib(m) {               // 非 CPS 版
  showArgument(m);
  var r;
  if (m < 2) {
    r = 1;
  } else {
    r = fib(m - 1) + fib(m - 2);
  }
  showResult(m, r);
  return r;
}

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。ここで `fib` は戻り値を持つので、接続は値を受け取る必要がある。

ファイル Fib.js

```
function showArgumentCPS(m, k) {     // 暫定版
  document.form.textarea.value
    += ("argument = " + m + "\n");
  return k(0);
}

function showResultCPS(m, r, k) {    // 暫定版
  document.form.textarea.value
    += ("result for argument: " + m + " = " + r +
"\n");
  return k(0);
}

function fibCPS(m, k) {              // 最終版
  return showArgumentCPS(m, function(ignore) {
    function tmp(r) {
      return showResultCPS(m, r, function(ingore)   {
        return k(r);
      });
    }
    if (m < 2) {
      return tmp(1);
    } else {
      return fibCPS(m - 1, function(r1) {
        return fibCPS(m - 2, function(r2) {
```

```
          return tmp (r1 + r2);
        });
      });
    }
  });
}

function doEnd(n) {                    // 暫定版
  document.form.textarea.value
    += "final result is " + n + " end\n";
}

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。

```
function showArgumentCPS(m, k) {     // 最終版
  document.form.textarea.value
    += ("argument = " + m + "\n");
  return k; // k(0) ではない
}

function showResultCPS(m, r, k) {    // 最終版
  document.form.textarea.value
    += ("result for argument: " + m + " = " + r +
"\n");
  return k; // k(0) ではない
}

/* fibCPS は変更なし */

function doEnd(n) {                    // 最終版
  document.form.textarea.value
    += "final result is " + n + " end\n";
  return function(ignore) { return doEnd(n); };
}

var restart = function(ignore) {
  return fibCPS(5, doEnd);
};
function exec() { restart = restart(0); }
```

これで、1 行表示するたびに停止する。

問 7.3.1 上のやり方にならって（CPS を使って）、次の関数を「ボタンを押したら一つの線分を表示する」というバージョンに書き換えよ。
ファイル Sierpinski.js

```
var ctx;
```

```
var x = 256, y = 256, dx = 8, dy = 0, h = 0;

function forward() {
  ctx.strokeStyle = "hsla(" + h + ", 100%, 50%, 0.8)";
  h++;
  ctx.beginPath(); ctx.moveTo(x, y);
  ctx.lineTo(x += dx, y += dy);
  ctx.closePath(); ctx.stroke();
  return 0;
}

function turnLeft() {
  var tmp = dx; dx = dy; dy = -tmp;
  return 0;
}

function turnRight() {
  var tmp = dx; dx = -dy; dy = tmp;
  return 0;
}

function sierpinski(n) {
  zig(n); zig(n);
  return 0;
}

function zig(n) {
  if (n <= 1) {
    turnLeft(); forward(); turnLeft(); forward();
  } else {
    zig(n / 2); zag(n / 2); zig(n / 2); zag(n / 2);
  }
  return 0;
}

function zag(n) {
  if (n <= 1) {
    turnRight(); forward(); turnRight(); forward();
    turnLeft(); forward();
  } else {
    zag(n / 2); zag(n / 2); zig(n / 2); zag(n / 2);
  }
  return 0;
}

function exec() {
  var canvas = document.getElementById('canvas');
  ctx = canvas.getContext("2d");
  sierpinski(16);
}
```

ヒント: 関数 `forward`, `sierpinski`, `zig`, `zag` を CPS に変換する必要がある。関数 `turnLeft`, `turnRight` については、（この問題では） CPS にする必要はない。

問 7.3.2 関数 `sierpinski` をジェネレーター関数を使って、「ボタンを押したら一つの線分を表示する」というバージョンに書き換えよ。

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` の各パラメーター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（このように `while` 文と `switch` ～ `case` 文を用いて、 `goto` 文を模倣する書き方のことは switch-in-a-loop construct というらしい。）。

```
// move は非 CPS 版と同じなので省略

var stack = new Array();
stack.push(new Array(5, 'a', 'b', 'c', 0));

function hanoiStack(n, a, b, c, pc) { // 明示スタック版
  while (n>0) {
    switch (pc) {
    case 0:
      stack.push(new Array(n, a, b, c, 1));
      var tmp = c; c = b; b = tmp; n--;
      continue;
    case 1:
      stack.push(new Array(n - 1, c, b, a, 0));
      move(n, a, b);
      return 0;
    }
  }
  return exec();
}

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

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、別のコンピューターで起動することさえ可能になる。
