Title	On a Class of Recursive Procedures and Equivalent Iterative Ones (情報科学の数学的基礎理論と応用)
Author(s)	HIKITA, TERUO
Citation	数理解析研究所講究録 (1979), 353: 107-115
Issue Date	1979-04
URL	http://hdl.handle.net/2433/104422
Right	
Type	Departmental Bulletin Paper
Textversion	publisher

数理解析研究所講究録
353巻 1979年 107-115

On a class of recursive procedures and equivalent iterative ones

Teruo Hikita

Department of Mathematics

Tokyo Metropolitan University

1. Introduction

Recursion removal (or recursion elimination) has been paid much attention for both theoretical and practical reasons. Several theoretical results on the translation of recursion schemes into equivalent flow chart schemes or other similar structures have clarified much of the nature of the problem, e.g., [1], [2], [5] and [6].

Recently H. Partsch and P. Pepper [3] concretely gave translations for a special class of recursion schemes to equivalent non-recursive schemes. The recursion scheme considered there has the following form, which was motivated by the well-known recursive solution for the "Towers of Hanoi" problem:

```
proc F = (int i) : if i > 0 then F(i-1); A(i); F(i-1) fi
```
[1,2]

1 All programs and program schemes in this paper are written in an Algol 68 -like notation.

2 proc F = (int i) void : ... would conform more to Algol 68.

They also gave several generalizations for the translation.

In this paper we investigate two other directions of generalizations for their recursion removals. First, in Section 2, the above scheme is extended to the case of mutual recursion. Recursive procedures for plotting space-filling curves such as the Hilbert curves and Sierpinski curves [7], [8] exactly fit into this class of recursion.

Secondly, Section 3 treats the case that the number of recursive calls in a procedure dynamically depends upon the current values of parameters given to this procedure. It turns out that the recursion removal given by J. S. Rohl [4] can be regarded as a special case of this type, and thus a unified view is obtained for these classes of recursion removal.

2. Mutual Partsch-Pepper recursion

Let us first consider the following scheme allowing mutual recursion of two procedures:

```
proc F = (int i) : if i > 0 then F(i-1); A(i); G(i-1) fi;
proc G = (int i) : if i > 0 then G(i-1); B(i); F(i-1) fi
```

with the initial call F(N). We assume that A and B stand for some actions non-local to F and G, using (but not changing) the value of i.

In this scheme the recursive procedure calls are used only as a control mechanism for generating a sequence of calls A(h) and B(h) where $1 \leq h \leq N$. More precisely, the call F(N) produces a ternary tree of which each node corresponds to a call of either F(h), G(h), A(h) or B(h), and the $2^N - 1$

leaves of the tree corresponding to A(h) and B(h) make up the resulted sequence for the call F(N). For example, the call F(3) is equivalent to the following:

```
begin  A(1);  A(2);  B(1);  A(3);  B(1);  B(2);  A(1)  end
```

Thus, the translation of this scheme to an iterative one is reduced to the problem of deciding what is the c-th call of such resulted sequence for a counter variable c ranging over 1 to $2^N - 1$. Computation of the value of h from c is exactly the same as the non-mutual case, which is given in [3]. The choice between two procedures A and B can be performed by simulating a certain finite-state automaton with its state set $\{1, 2\}$; these states correspond to the procedures F and G, respectively, and they indicate the current status of control, that is, in which procedure control is held.

An equivalent iterative scheme can be given as follows:

```
proc F = (int n) :
(for c from 1 to 2^n - 1 do
     int d, r, h, U;  r := c;  U := 1;
     for s from n by -1 to 1 do
          d := r ÷ 2^(s-1);  r := r mod 2^(s-1);
          if r = 0 then h := s;  goto execute
          elif d = 0 then  U := (if U = 1 then 1 else 2 fi)
                           else  U := (if U = 1 then 2 else 1 fi) fi od;
execute:  if U = 1 then A(h) else B(h) fi od)
```

We now proceed to a general form of mutual recursion of m procedures.

The recursion scheme to be considered is as follows:

$\underline{\text{proc}}\ F_1 = (\underline{\text{int}}\ i) :\ \underline{\text{if}}\ i > 0\ \underline{\text{then}}$

$A_{1,0}(i);\ F_{q_{1,1}}(i-1);\ A_{1,1}(i);\ F_{q_{1,2}}(i-1);\ \ldots\ ;$

$A_{1,k-1}(i);\ F_{q_{1,k}}(i-1);\ A_{1,k}(i)\ \underline{\text{else}}\ B_1\ \underline{\text{fi}};$

.....

$\underline{\text{proc}}\ F_m = (\underline{\text{int}}\ i) :\ \underline{\text{if}}\ i > 0\ \underline{\text{then}}$

$A_{m,0}(i);\ F_{q_{m,1}}(i-1);\ A_{m,1}(i);\ F_{q_{m,2}}(i-1);\ \ldots\ ;$

$A_{m,k-1}(i);\ F_{q_{m,k}}(i-1);\ A_{m,k}(i)\ \underline{\text{else}}\ B_m\ \underline{\text{fi}}$

with the initial call $F_1(N)$. Each of $F_{q_{1,1}}, \ldots, F_{q_{m,k}}$ is arbitrary chosen from $F_1, \ldots, F_m$, and is fixed; that is, each of $q_{1,1}, \ldots, q_{m,k}$ is equal to some integer among 1, ... , m. $A_{1,0}, \ldots, A_{m,k}, B_1, \ldots, B_m$ stand for some actions non-local to $F_1, \ldots, F_m$, without changing the value of i.

Note that the values of parameters given to recursive calls are all equal, and also that the numbers of these recursive calls in the procedures are all equal to k. The recursive programs in [7, Chap. 3, Sec. 3] for plotting some members of a sequence of polygonal arcs which converges to a space-filling curve, such as the Hilbert curves and the Sierpinski curves (see also [8]), exactly fit into this class of recursion.

It would be rather tedious to directly generalize the previous translation to this general case of mutual recursion of m procedures. A clever idea is to introduce an array U, which keeps information on the current status of control for each level of recursion corresponding to the value of i. The status is indicated by the values from 1 to m, each of which

represents that control is within the procedure F_1, ... , F_{m-1} or F_m, respectively. It is also more convenient to express the counter variable c as a k-ary counter, in order to easily obtain information on the status of control within a procedure for each level of recursion.

The program may be viewed as a kind of tree traversal or backtracking. An equivalent iterative scheme for the general mutual recursion is as follows:

```
proc F = (int n) :  ([1 : n] int c;  int t;  [0 : n] int U;
for s from 1 to n do c[s] := 0 od;  t := n + 1;  U[n] := 1;
do   for s from t - 1 by -1 to 1 do A_{U[s],0}(s);  U[s-1] := q_{U[s],1} od;
     B_{U[0]};
     for s from 1 to n do if c[s] = k - 1 then c[s] := 0
                          else c[s] +:= 1;  t := s;  goto up fi od;
     t := n + 1;
up:  for s from 1 to t - 1 do A_{U[s],k}(s) od;
     if t = n + 1 then goto fin fi;
     A_{U[t],c[t]}(t);  U[t-1] := q_{U[t], c[t]+1} od;
fin:  )
```

The array U works as a stack for the control of recursive calls. The resulted simplicity is owing to the uniformness of both the values of parameters given to recursive calls and the numbers of recursive calls in the procedures.

3. Rohl's recursion removal

In this section we treat the second direction of generalization for the recursion removal of Partsch and Pepper. This is the case that the number of recursive calls in a procedure is not fixed but dynamically depends upon the current values of parameters given to this procedure.

The recursion scheme to be considered is

```
proc F = (int i, xp) : (int x; x := xp;
if i > 0 then
    repeat A(i, x); F(i-1, a(i, x)); B(i, x) until P(i, x) [3]
else D(x) fi)
```

with the initial call F(N, X). A, B and D stand for some actions non-local to F, and we assume that they may possibly change the value of x but not that of i. P and a stand for some predicate and function, respectively, both computed on i and x without side-effects on their values.

In this case the total number of calls of A, B and D cannot be immediately predicted by the initial value of the parameter i, so that the counter in the previous section cannot be utilized. However, the idea of introducing an array V for retaining values of the parameter x for each level of recursion can be used in this case, and it can also be used in order to control the traverse on the "tree" generated by the call F(N, X).

An equivalent non-recursive scheme is as follows:

3 `while (A(i, x); F(i-1, a(i, x)); B(i, x); not P(i, x)) do skip od` would conform more to Algol 68.

```
proc F = (int n, x) :  (int t;  [0 : n] int V;  t := n + 1;  V[n] := x;
do   for s from t - 1 by -1 to 1 do A(s, V[s]);  V[s-1] := a(s, V[s]) od;
     D(V[0]);  t := 0;
     repeat  t := t + 1;  if t = n + 1 then goto fin fi;
             B(t, V[t])
     until   not P(t, V[t]);
     A(t, V[t]);  V[t-1] := a(t, V[t]) od;
fin:   )
```

J. S. Rohl [4] considered a certain recursive procedure which produces in ascending order all the combinations of the first N natural numbers taken r at a time, and he translated it to several equivalent iterative ones. Fixing the value N, the main part of his recursive program (originally written in an Algol 60 -like language) may be written in the following form:

```
[0 : r] int c;  c[0] := 0;
proc choose = (int k) :
for d from c[k-1] + 1 to N - r + k do
     c[k] := d;  if k ≠ r then choose(k+1) else output(c) fi od
```

with the initial call choose(1).

By safely exchanging the order of the for clause and the if clause, and introducing the second parameter dp, it can be shown that this procedure is equivalent to the following:

```
[1 : r] int c;
proc choosel = (int k, dp) :  (int d;  d := dp;
if k < r then
     repeat  d := d + 1;  c[k] := d;  choosel(k+1, d)
     until   d = N - r + k
else repeat  d := d + 1;  c[k] := d;  output(c)
     until   d = N - r + k fi)
```

with the initial call choosel(1, 0). Now it is immediately seen that this procedure belongs to the recursion scheme considered in this section. It is also seen that the translations of the recursive procedure choose into iterative ones in [4] may be regarded as a special case of ours.

References

[1] J. Engelfriet: Simple program schemes and formal languages, Lecture Notes in Computer Science, vol. 20, Springer, 1974.

[2] S. J. Garland and D. C. Luckham: Program schemes, recursion schemes, and formal languages, JCSS 7(1973), 119-160.

[3] H. Partsch and P. Pepper: A family of rules for recursion removal, Information Processing Letters 5 (1976), 174-177.

[4] J. S. Rohl: Converting a class of recursive procedures into non-recursive ones, Software - Practice and Experience 7 (1977), 231-238.

[5] H. R. Strong, Jr.: Translating recursion equations into flow charts, JCSS 5 (1971), 254-285.

[6] S. A. Walker and H. R. Strong: Characterizations of flowchartable recursions, JCSS 7 (1973), 404-447.

[7] N. Wirth: Algorithms + Data Structures = Programs, Prentice-Hall, 1976.

[8] Computer Recreations by Aleph Null: Space-filling curves, or how to waste time with a plotter, Software - Practice and Experience 1 (1971), 403-410.