The Cyclic Towers of Hanoi: An Iterative Solution Produced by Transformation

T. D. Gedeon

School of Computer Science and Engineering, The University of New South Wales, Sydney 2052, Australia
Email: tom@cse.unsw.edu.au

An iterative solution to the Cyclic Towers of Hanoi puzzle is produced by largely automatic program transformation from a recursive solution. The result compares favourably with the best published, manually produced iterative algorithm, both in terms of comprehensibility in its own right, and in efficiency.

Received November 11, 1993; revised April 19, 1996

1. INTRODUCTION

The Cyclic Towers of Hanoi puzzle is an extension by Atkinson [1] of the Towers of Hanoi problem. The three pegs are arranged in a circle. A restriction in addition to the normal rules is the requirement that discs can only be moved clockwise from peg 1 to peg 2, from peg 2 to peg 3, or from peg 3 to peg 1.

2. PROLOGUE

Gedeon [2] presented an iterative algorithm for the Reve’s Puzzle produced by transformation using schemata similar to those of Partsch and Pepper [3]. The program produced compares favourably with previous iterative versions. The algorithm is certainly easier to understand, and has much lower space requirements than Lu’s [4], and also has an efficiency lead on the algorithm by Hinz [5]. The transformation was produced largely automatically, with some cosmetic interactive manipulations of the program, using the IMP program transformation system [6, 7].

Briefly, the IMP program transformation system eliminates recursion by the automatic generation of recursive and non-recursive schema pairs. The recursive schema is derived from the recursive procedure by abstraction, while the non-recursive schema is generated from the recursive schema by using an extension of Rohr’s method [8] of substitution, being a symbolic evaluation of the order of execution of the substantive statements of the recursive version.

The Reve’s puzzle forms a complete tree of procedure calls, which allows us to consider the symbolic evaluation in terms of traversing the leaves along the bottom of the tree, from left to right rather than climbing up each branch and then back down at each step. This allows us to collect together the recursive ascent and descent simulating code. The Cyclic Towers of Hanoi puzzle does not form a complete tree of procedure calls, requiring an extension of this method.

3. CYCLIC TOWERS OF HANOI

Atkinson’s solution uses two mutually recursive procedures clock and anti. They are shown below, with minor modifications of the names and types of the variables, and to the writeln statements:

\[
\begin{align*}
\text{procedure clock}(n : \text{discs}; p1, p2, p3 : \text{peg}); & \quad \text{forward}; \\
\text{procedure anti}(n : \text{discs}; p1, p2, p3 : \text{peg}); & \\
\text{begin} \\
\text{if } n > 0 \text{ then} \\
\text{begin} \\
\text{anti}(n - 1, p1, p2, p3); \\
\text{writeln}(p1 : 1, \rightarrow, p3 : 1); \\
\text{clock}(n - 1, p2, p1, p3); \\
\text{writeln}(p3 : 1, \rightarrow, p2 : 1); \\
\text{anti}(n - 1, p1, p2, p3) \\
\text{end} \\
\text{end} \{ \text{anti} \}; \\
\text{procedure clock}; \\
\text{begin} \\
\text{if } n > 0 \text{ then} \\
\text{begin} \\
\text{anti}(n - 1, p1, p3, p2); \\
\text{writeln}(p1 : 1, \rightarrow, p2 : 1); \\
\text{anti}(n - 1, p3, p2, p1) \\
\text{end} \\
\text{end} \{ \text{clock} \};
\end{align*}
\]

The initial call is to \(\text{clock}(n, 1, 2, 3)\). We can see that this call will produce only two (indirectly) recursive calls, each of which have three descendants, but the middle calls (to \text{clock}) again only have two descendants. This is the pattern illustrated in Figure 1. Note that the towers of discs in the calls to \text{anti} move counterclockwise, though the individual discs always move clockwise as shown by the writeln statements. Atkinson’s solution is optimal though not obvious. Even experienced programmers will produce a solution based on the original Towers of Hanoi solution in which the towers also move clockwise for two
moves, to achieve the same effect for tower position as Atkinson’s anticlockwise tower move. Moving towers clockwise is inefficient because some discs end up moving four times clockwise rather than just once clockwise.

Recursive procedures with regular deviations from a complete tree of procedure calls as shown in Figure 1 occur naturally from the merging of a system of mutually recursive calls in which the calls in different procedures are different in number. The Cyclic Towers of Hanoi is one of the simplest examples of this phenomenon.

The two procedures used by Atkinson can be merged with the IMP program transformation system performing the clerical labour of verifying the correctness and performing the rewriting to produce a single recursive procedure as follows:

```
procedure Cycle(which : whichtype; n : discs; p1, p2, p3 : peg);
begin
  if n > 0 then
    begin
      Cycle(anti, n - 1, p1, p2, p3);
      writeln(p1 : 1, ' → ', p3 : 1);
      if which = anti then
        begin
          Cycle(clock, n - 1, p2, p3, p1);
          writeln(p3 : 1, ' → ', p2 : 1)
        end
      else
        rotate3(p1, p2, p3);
      Cycle(anti, n - 1, p1, p2, p3)
    end
end { Cycle };
```

An enumerated type which has been introduced with values derived from the names of the two original procedures. To effect the above change the initial call has now become Cycle(clock, n, 1, 3, 2) and a procedure rotate3 introduced. The textually first and last calls to Cycle correspond to the first and last calls in both clock and anti. The call to rotate3 is introduced to reorder the peg parameters of the second call to anti in clock to match the last call in anti.

Analysing the nature of the recursion, we can see that none of the parameters of Cycle need to be stacked upon eliminating the recursion:

- The variable n is modified by the subtraction of 1 at each call, this is invertible, the inverse being + 1.
- Similarly, the peg parameters are only modified in swaps which are their own inverse, or by the rotate3 which is also readily inverted.
- The parameter which, on the other hand, is used after a recursive call and the expression is clearly not invertible without storing the previous value. When eliminating recursion in IMP, this is done by creating a non-recursive schema by symbolic evaluation of the order of execution of the substantive statements of the recursive version. This has the effect that the iterative version produced can be viewed as a simulation of the recursive version. It is possible to extract the information as to which recursive call is being simulated at any moment. This information is not generally useful. In this example one of the actual parameters passed into a recursive call is a constant, the value of which being clock occurs uniquely in the second textual recursive call. With all other calls having the value anti, this value can be retrieved in this case without stacking the value of which. This is automatically discovered by IMP, as are the inverses for the other parameters.

The tree of procedure calls will comprise complete sub-trees and complete sub-trees with regular ‘holes’ whose location we can find, in the same manner. If there is a substantial statement which is executed instead of the missing recursive call, it must be invertible for this transformation to be correct (and allowed by IMP). The holes in the tree of procedure calls occur whenever which has the value clock, which occurs in the very first call and in the descendants of textually second recursive calls. The holes are handled by skipping over them, if we consider the symbolic evaluation in terms of traversing the leaves along the bottom of the tree, from left to right rather than climbing up each branch and then back down at each step.

4. RECURSIVE SCHEMA

The following recursive schema is automatically generated from the recursive procedure by a process of abstraction.

```
begin
  if expression(1) then
    begin
      call(1);
      statement(1);
```

if expression(2) then
   begin
      call(2);
      statement(2)
   end
else
   statement(3);
   call(3)
end;

Note that the schema has no separate procedure heading, being generated in the context of the specific recursive procedure.

5. ITERATIVE SCHEMA

The following non-recursive (iterative) schema is automatically generated to match the recursive schema. The schema is exactly as generated.

begin
   initial;
   S: = 1;
   final: = Power3to(n) - 1;
while S <= final do
   begin
      Decompose3(S, level, value, residue);
      if value = 2 then
         begin
            if level > 0 then
               statement(-3) { inverse of statement(3) };
               exit(2);
               statement(2);
               entry(3)
            end
         else
            begin
               exit(1);
               statement(1);
               if (residue mod 9 = 4) or (level = n - 1) then
                  begin
                     statement(3);
                     S: = S + Power3to(level);
                     entry(3)
                  end
               else
                  begin
                     entry(2)
                  end;
               S: = S + 1
         end
end;

The iterative schema is generated from the recursive schema in the context of the recursive procedure and the modifications of the parameters. The IMP system recognised that all of the functions applied to the parameters are invertible, allowing a symbolic evaluation traversing the leaves. In IMP these are called for-loop schemata [6]. Since the tree of procedure calls is not complete a while loop is actually used, with schema control statements included to skip the gaps.

Note also that the behaviour for value = 1 subsumes the behaviour for value = 3, since the parameter lists of both calls are identical, and in for-loop schemata the exit and entry schema devices affect only the parameters which do not control the recursion. In this case the peg parameters, which are not modified by calls 1 and 3.

6. ITERATIVE PROCEDURE

The non-recursive result produced by IMP for the Cycle procedure is shown below:

procedure Cycle(which : whichtype; n : discs; p1, p2, p3 : peg);
var S, level, value, residue, final : integer;
begin
   S: = 1;
   final: = Power3to(n) - 1;
while S <= final do
   begin
      Decompose3(S, level, value, residue);
      if value = 2 then
         begin
            if level > 0 then
               rotate3(p1, p3, p2);
               rotate3(p2, p1, p3);
               writeln(p3 : 1,' → ',p2 : 1)
         end
      else
         begin
            writeln(p1 : 1,' → ',p3 : 1);
            if (residue mod 9 = 4) or (level = n - 1) then
               begin
                  rotate3(p1, p2, p3); 
                  S: = S + Power3to(level)
               end
            else
               rotate3(p3, p1, p2)
         end;
      S: = S + 1
   end
end { Cycle };

The while loop on S traverses the leaves, the procedure Decompose3 discovers how far up the tree the next substantive statement is, and which recursive call it is ‘in’. The parameter n controls the recursive descent and is largely subsumed by the schema variable level, which is extracted from S as needed.

There are four calls to rotate3 in the above iterative procedure which have different origins in the recursive procedure. They are described by the iterative schema devices they correspond to, in order they appear textually in the iterative procedure above:

(i) is the inverse of the rotate3 in the text of the recursive version;
(ii) is the inverse of the shuffling of the peg parameters in the second recursive call;
(iii) is directly copied from the recursive version; and
(iv) implements the shuffling of the peg parameters in the second recursive call

The value of residue is used to find the identity of the parent recursive call, and the Power3to is used to determine the size of the gap at the leaves of the tree, depending on the height at which it was ‘pruned’.

This can be simplified further within IMP, with user guidance of the sequence and application of transformations, which are verified for correctness and applicability by the system.

We will now skip directly to the simplified version, which is now as follows:

```
procedure Cycle(which : whichtype; n : discs; p1, p2, p3 : peg);
var S, level, value, residue, final : integer;
begin
S := 1;
final := Power3to(n) - 1;
while S <= final do
begin
Decompose3(S, level, value, residue);
if value = 2 then
if level = 0 then
rotate3(p1, p3, p2)
else
if (residue mod 9 = 4) or (level = n - 1) then
S := S + Power3to(level);
writeln(p1 : 1, ' -> ', p2 : 1);
rotate3(p1, p3, p2);
S := S + 1
end
end
end { Cycle };
```

We can clearly see that the discs all move in the same direction: in the clockwise direction from p1 to p2, with the pegs rotating in a counterclockwise direction. The order of magnitude of the number of moves required for \( n \) discs is visibly < 3\(^n\). The number is of the order \( (1 + \sqrt{3})^n \) which is approximately 2.73\(^n\). The occasional use of the first textual rotate3 means that sometimes the same disc will be moved twice in a row. This is clearly the smallest disc (level = 0) as it moves out of the way of another disc, corresponding to the two successively executed writeln statements in the original procedure anti at the lowest level of the recursion (\( n = 1 \) corresponds to level = 0).

The best previous iterative algorithm available in the literature is due to Er [9]. Er’s algorithm is marginally slower than that developed above, and requires \( O(n) \) space versus \( O(\log n) \) space for the IMP version. The IMP algorithm is simpler and clearer, for example proving that the loop terminates is trivial as the control variable \( S \) increases by at least one every time through the loop.

7. CONCLUSIONS

We have produced an iterative algorithm for the Cyclic Towers of Hanoi puzzle which compares favourably with previous versions. The algorithm is certainly easier to understand than previous interactive versions, and has lower space requirements than [9]. Of course all of the iterative versions are still somewhat less comprehensible or amenable to complexity analysis than the recursive version, hence the interest in automatic transformation to iterative versions.

Most importantly, the above program has been produced essentially automatically, the interactive manipulations of the program are largely cosmetic, the bulk of the work had already been done by the IMP program transformation system.

Finally, in this process we have also shown by transformation some interesting properties of the Cyclic Towers of Hanoi puzzle.

REFERENCES