The Reve's Puzzle: An iterative solution produced by transformation

An iterative solution to the Reve's puzzle is produced by largely automatic program transformation from a recursive solution. The result compares favourably with published, manually produced iterative algorithms, both in terms of comprehensibility in its own right, and efficiency.

Received November 1990

1. Introduction

The Reve's Puzzle is an extension of the Towers of Hanoi problem by Dudeney\(^1\) (1907). The puzzle is posed in terms of 4 pegs (unaware at the time of Dudeney's book) which used the standard 3 peg solution to the Towers of Hanoi problem with 4 or 5 pegs, however, the precise rules are not given. The illustration shows 5 stools and a number of cheeses of diminishing size. The rules are identical to those for the Towers of Hanoi.

2. Prologue

Rohl and Gedeon\(^1\) presented a programming solution to the Towers of Hanoi problem with 4 pegs (unaware at the time of Dudeney's book) which used the standard 3 peg solution as a sub-routine as follows:

```pseudocode
procedure Hanoi4(n:ndiscs; p1, p2, p3, p4:pegs);
begin
  if n > 0 then
    begin
      Hanoi4(n - F(n), p1, p4, p3, p2);
      Hanoi3(F(n), p1, p2, p3);
      Hanoi4(n - F(n), p4, p2, p3, p1)
    end
end
```

that the clerical function of 'adjudication' corresponded precisely to 'freezing' all items of a record other than those contained in the standard evidence-header.

Consideration was given to providing REAL-TIME-PERIOD and REFERENCE-TIME-PERIOD as record-types in the physical database structure, to which every evidence record would be physically linked. Our conclusion was, however, that this would lead to unacceptable performance overheads arising from the additional database navigation whenever an evidence record was to be updated. So in the final system, evidence records are chained only to the owning PERSON record, and common routines have been developed that perform the necessary date-matching against candidate records.

The requirement to retain multiple conflicting versions of the 'same' data, as it was believed to be at different points in the past, cannot be unique to the DSS. The design solution described here is now well-proven and is published in the belief that it could be of value to a wider audience.

1. SHEARER

References

3. S. Jones and P. Mason, Handling the time dimension in a database. In S. M. Deen and P. Hammersley (see above).

Table 1. The first number indicates the value of n in calls to \textit{Hanoi4}, while the list of numbers shows the values of n in initial calls to \textit{Hanoi3}.

<table>
<thead>
<tr>
<th>n</th>
<th>Hanoi4(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1, 1</td>
</tr>
<tr>
<td>3</td>
<td>2, 1</td>
</tr>
<tr>
<td>4</td>
<td>2, 1, 1</td>
</tr>
<tr>
<td>5</td>
<td>3, 2, 1</td>
</tr>
<tr>
<td>6</td>
<td>3, 2, 1</td>
</tr>
<tr>
<td>7</td>
<td>3, 2, 1, 1</td>
</tr>
<tr>
<td>8</td>
<td>3, 2, 2, 1</td>
</tr>
<tr>
<td>9</td>
<td>3, 2, 2, 1</td>
</tr>
<tr>
<td>10</td>
<td>4, 3, 2, 1</td>
</tr>
</tbody>
</table>

We can also see that the number of times for which A is \(2^{\text{th}}\) is repeated is also the same as the depth within the recursive descent on \textit{Hanoi4} that the 'doubling' occurs.

3. The Reve's Puzzle

The above observations lead to the following procedure:

```pseudocode
procedure (Reve(n:integerses));
var position, discontinuity:natural; s1, s2, s3, s4:stools;
begin
  if position > 0 then
    begin
      Reve(position - 1, skip, s1, s4, s3, s2);
    end
  Hanoi(position - 1, s1, s2, s3)
```

Figure 3. Extended logical data model.

Table 3. The first number indicates the value of n in calls to \textit{Hanoi4}, while the list of numbers shows the values of n in initial calls to \textit{Hanoi3}.

<table>
<thead>
<tr>
<th>n</th>
<th>Hanoi4(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1, 1</td>
</tr>
<tr>
<td>3</td>
<td>2, 1</td>
</tr>
<tr>
<td>4</td>
<td>2, 1, 1</td>
</tr>
<tr>
<td>5</td>
<td>3, 2, 1</td>
</tr>
<tr>
<td>6</td>
<td>3, 2, 1</td>
</tr>
<tr>
<td>7</td>
<td>3, 2, 1, 1</td>
</tr>
<tr>
<td>8</td>
<td>3, 2, 2, 1</td>
</tr>
<tr>
<td>9</td>
<td>3, 2, 2, 1</td>
</tr>
<tr>
<td>10</td>
<td>4, 3, 2, 1</td>
</tr>
</tbody>
</table>

We can also see that the number of times for which A is \(2^{\text{th}}\) is repeated is also the same as the depth within the recursive descent on \textit{Hanoi4} that the 'doubling' occurs.

References

3. S. Jones and P. Mason, Handling the time dimension in a database. In S. M. Deen and P. Hammersley (see above).

Table 1. The first number indicates the value of n in calls to \textit{Hanoi4}, while the list of numbers shows the values of n in initial calls to \textit{Hanoi3}.
else Hanoi(position, s1, s2, s3);
R4(position = 1, skip, s4, s2, s3, s1)
end (R4 );
begin (Reve )
position := trunc(sqrt(1 + 8*n) – 1)/2;
discontinuity := n – position*(position + 1)
div 2;
R4(position + 1, discontinuity, 1, 2, 3, 4)
end (Reve );
Note that the procedure has also been recast as per Dudeney, using the name Reve, and in terms of cheeses and stools. We can therefore revert to using just Hanoi for the name of the standard 3 peg problem. A two-level solution has been used to localise initialisations. The variable position is now set once, unlike the function F which was re-calculated repeatedly.

Analysing the nature of the recursion, we can see that none of the parameters of R4 need to be stacked.

The variable position is modified by the subtraction of 1 at each call, this is invertible, the inverse being +1.

Similarly, the stool parameters are only modified in swaps which are their own inverse.

The variable skip does not require stacking, as its value is not altered throughout the traversal. Its use as a parameter is good programming practice from a structured viewpoint is not using a global variable, and maintaining a well controlled interface to the rest of the program. The cost of this better structure is a loss of efficiency, due to the allocation and assignment to a local copy at every call to the procedure. This cost is of course recoverable when we eliminate the recursion, which we will do by using the IMP program transformation system by Gedeon.2

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 Rohlf’s method of substitution, being a symbolic evaluation of the order of execution of the substantive statements of the recursive version.

In this case, the Reve procedure performs a complete tree of procedure calls, allowing a specialised recursive schema to be generated, similar to those of Partsch and Pepper.4 A complete tree of procedure calls 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 non-recursive result produced for the Reve procedure (by IMP) is shown below:

begin Decompose2(S, level);
if odd(level) then
swap(s1, s4);
swap(s2, s4);
if 1 + level > skip then
Hanoi(level, s1, s2, s3)
elself Hanoi(1 + level, s1, s2, s3);
if odd(level) then
swap(s4, s2)
end
end (Reve);

The for loop on S traverses the leaves, the procedure Decompose2 discovers how far up the tree the next substantive statement is, and which recursive call it is ‘in’. The parameter position controls the recursive descent and is largely subsumed by the schema variable level, which is extracted from S as needed.

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 Reve(n: « cheeses);
type natural = 0..maxint;
var position, discontinuity: natural;
s1, s2, s4: nstools;
S, level, nap: integer;

begin
position := trunc(sqrt(1 + 8*n) – 1)/2;
discontinuity := n – position*(position + 1)
div 2;
s1 := 1;
s2 := 2;
s4 := 4;
if odd(position + 1) then
swap(s4, s2);
for S := 1 to Power2(position + 1) – 1 do
begin
Decompose2(S, level);
if odd(level) then
swap(s1, s4);
swap(s2, s4);
if 1 + level > skip then
Hanoi(level, s1, s2, s3)
elself Hanoi(1 + level, s1, s2, s3);
swap(s4, s1);
if odd(level) then
swap(s4, s2)
end
end (Reve);

4. Conclusions
We have produced an iterative algorithm for the Reve’s puzzle which compares well with previous versions. The algorithm is certainly easier to understand, and has much lower space requirements than Lu,4 and also has an efficiency lead on that of the quite similar algorithm by Hinz2 which requires the use of exponentiation by powers other than 2, which can not therefore be implemented by efficient shift instructions on compilation.

Most importantly, we have produced the above program 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 Reve’s puzzle, properties which others have assumed.

This iterative algorithm for the Reve’s puzzle is similar to the well known iterative solution to the Towers of Hanoi problem, the stools are arranged in a circle, and the towers in alternate levels move in opposite directions. It is surprising from a purely intuitive viewpoint that the rotation is again of length 3 not 4. Also interesting is that stood 3 is always the spare stool when moving discs using 3 stools; in retrospect this is also clear from the original recursive version.

Acknowledgements
I would like to acknowledge the guidance and encouragement of Jeff Rohl.

GEDEON, T. D.
Department of Computer Science, Brunei University, Uxbridge, Middlesex, UB8 3PH.

References