SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 11, 745–751 (1981)

Recursion in Small Storage

LESLIE M. GOLDSCHLAGER
Department of Computer Science, University of Queensland, St. Lucia, Queensland 4067, Australia

SUMMARY

An implementation method for procedure calls is discussed which can lead to significant storage space reductions for recursive algorithms.

KEY WORDS Recursion Procedure-call implementation Space efficiency

INTRODUCTION

Recursion is a valuable tool in structured programming, as there exist many cases in which recursion is the natural way of thinking about an algorithm. In such cases, it is likely that the recursive expression of the algorithm will be easier to understand, to prove correct and to analyse than an iterative equivalent.

However, the use of recursion is often frowned upon for various reasons. Wirth[1] cites as one reason, the use by educators of inappropriate examples, coupled with the fact that the deeply entrenched Fortran language disallows recursion. Rohl[2] discusses time efficiency, concluding that time penalties for the use of recursion need not be large, and, indeed, with some machine language instruction sets, the implementation of a recursive procedure may execute faster than the corresponding loop construct and explicit stack handling.

In this paper, we concentrate upon the *space efficiency* of recursive procedures. The problem here is that the standard implementation of procedures uses an extra block of memory (stack frame) to hold links, parameters and local variables. This is not a penalty for certain recursive solutions which, if implemented iteratively, would necessarily require the same amount of memory, explicitly declared and handled.

There are other recursive algorithms, however, which under the standard implementation use more memory than is necessary. The 'inappropriate examples' cited by Wirth are uniformly in this category. For example, a linear recursive algorithm which makes c recursive calls would normally use $O(c)$ storage (read 'order c', meaning proportional to c). This can be reduced[3] to $O(\sqrt{c})$ with only a constant factor increase in the computation time. Gurari and Ibarra[4] have shown that the storage can be further reduced to $O(\log c)$, as can any algorithm which makes c recursive calls, but in general the technique given can result in an unacceptable execution time.

In the next section we discuss an implementation method for procedure calls which gives the same execution time as the standard implementation, but which reduces the storage space used to the Gurari–Ibarra bound (or better) for very many common types of recursive procedures. Examples are then presented of recursive algorithms whose extra storage would be reduced to almost zero by the non-standard implementation.

0038–0644/81/070745–07$01.00

Received 19 November 1979

The following section on unbalanced recursion is concerned with cases where the storage space reduces to the Gurari–Ibarra bound. Finally, we show that the non-standard implementation can also have a significant impact on the storage space used by mutually recursive procedures.

IMPLEMENTATION

It is well known[1, 5] that if the last action performed by a procedure is to call another procedure, then the procedure call can be replaced by a direct jump. Knuth and Wirth conclude that therefore the procedure call *should* be replaced by a direct jump, both for time efficiency and, more importantly in the context of this paper, to save a stack frame. But if the recursive algorithm is the natural expression of the problem solution, then it is a pity to obscure this natural expression by transforming the recursive algorithm into an equivalent iterative form.

Alternatively, it is possible for a compiler to detect the condition discussed above in certain special cases (e.g. the BLISS/11 compiler[6]). The disadvantage of this approach is that not all cases can be automatically detected, and a fairly good understanding of when a particular compiler performs this optimization is required in order to analyse the storage used by the program.

The approach that will be adopted in this paper is that the programming language should be able to indicate when the optimization is required. This can be achieved by using a return statement,[7, 8] whose semantics are that the expression following **return** should be evaluated and then control returned to the calling program (returning the expression value in the case of a function). This would appear to be a good programming practice in that the programmer's intention is made clear by the written code.

The implementation of procedure calls by a compiler is now as follows. In an ordinary case, a procedure call creates a new stack frame containing parameters and local variables, the return address and the static and dynamic links required by a block structured language in the usual way. But when the procedure call is preceded by **return**, the current stack frame is re-used. The return address and dynamic link are left unchanged, while the static link, parameters and local variables are set as in the ordinary case. For a value parameter, the value is placed on the stack, and for a reference parameter, the address of the object referred to is placed on the stack. This latter rule is important in case the actual parameter is itself a formal reference parameter of the current procedure.

Two new compile-time errors can arise from this use of the return statement, and they are easy to detect. The first occurs if the procedure call following **return** is declared locally to the current procedure. The second occurs if the procedure call following **return** contains a reference parameter which is either a local variable or a value parameter of the current procedure. In both cases, the return statement makes no sense, and the compiler should generate an error message.

LINEAR RECURSION

In this section we show that the non-standard recursion implementation discussed above allows some recursive algorithms to be expressed in the natural way and yet use no

stack space whatsoever (or rather just a single stack frame). Numerous examples occur in Lisp,[9] some of which appear in the section on Mutual Recursion.

A common elementary example of recursion is Euclid's algorithm:

```
function gcd(p, q);
(* returns the greatest common divisor of p and q, assuming p ≥ q > 0 *)
if p mod q = 0 then return q
               else return gcd(q,p mod q);
```

This would be one of Wirth's 'inappropriate examples' because the standard implementation would result in the use of unnecessary stack space. But the non-standard implementation results in the use and re-use of only a single stack frame to remember the current values of p and q, and thus it is not only a good procedure from the structured programming viewpoint, but also a space (and time) efficient solution.

The following binary search procedure is another example of a recursive procedure which becomes space efficient using only a single stack frame with the non-standard implementation.

```
function search (tree, value);
(* returns a pointer to that node of the ordered binary tree which contains value*)
if tree = nil then return nil
elseif value = tree ↑.datum then return tree
elseif value <tree↑.datum then return search(tree↑.left, value)
else return search(tree↑.right, value);
```

The non-standard implementation can also handle parameters which are passed by reference, as in the following procedure which produces a copy of a list.

```
procedure copy(list, var newlist);
(* returns a copy of list in newlist *)
if list = nil then newlist : = nil
else new(newlist);
     newlist↑.datum : = list↑.datum;
     return copy(list↑.next, newlist↑.next);
```

A little thought shows that we have managed to copy a list using a straightforward algorithm and two temporary locations, *list* and the address of *newlist*. We have not used stack space proportional to the length of the list.

A final example of linear recursion is a tree traversal algorithm which works on doubly linked trees. Of course the back links to the parent nodes use storage space, and therefore this space saving traversal routine is only practical if the back links are required anyway.

```
procedure traverse(node, origin);
(* We have just arrived at node from origin. Traverse the remainder of the tree in the
   order left subtree, right subtree and finally parent of node *)
if node < > nil then
  if origin = node↑.parent then (* preorder *)
          if node↑.left < > nil then return traverse (node↑.left,node);
  if origin < > node↑.right then (* inorder *)
          if node↑.right < > nil then return traverse (node↑.right,node);
  (* postorder *) return traverse(node↑.parent, node);
```

This procedure again uses no extra stack space. If it has arrived at a node coming from its parent, it tries to go left. If there is no left subtree, or it has just come from the left, it tries to go right. If there is no right subtree, or it has just come from the right, the procedure returns to the parent. If it is desired to process each node exactly once in either a preorder, inorder or postorder traversal, the statement *process(node)* should be inserted where shown. This traversal procedure also has some theoretical interest, as discussed in the Appendix.

UNBALANCED RECURSION

This section is concerned with procedures which execute two or more recursive calls per invocation. If the total number of recursive calls is c, the execution can be viewed as a recursion tree with c nodes. The standard procedure call implementation would result in a stack size equal to the depth of the tree, which is O($\log c$), the Gurari–Ibarra bound, if the tree is nicely balanced, but could be as bad as O(c) in an unbalanced tree. It turns out that the non-standard implementation will, for many problems, result in an O($\log c$) stack size, no matter how badly unbalanced the recursion tree is.

```
procedure quicksort (i,j);
(* sorts A[i]...A[j] using only log(j−i) stack frames *)
if i <> j then
  partition A into A[i]...A[m] and A[m+1]...A[j];
  if m<(i+j)/2 then quicksort(i,m);
                    return quicksort(m+1,j);
               else quicksort(m+1,j);
                    return quicksort(i,m);
```

Notice that this is Hoare's[10] trick of always sorting the smaller section first, so that any subproblem which is stacked can be at most half the size of the previous problem, resulting in an O($\log n$) stack space in the worst case for sorting n numbers. The conclusion reached by Knuth[5] (see also Wirth[1]) is to transform at least the second recursive call to an iterative form in order to save stack space. But we can now see that the structured recursive version can be retained, with the non-standard implementation saving stack space for us.

A similar space saving occurs when scanning for a syntactic structure in a parse tree, defined as follows:

```
type nodeptr = ↑node;
  node = record
                 syntstruc : string;   (* name of syntactic structure *)
                 start : integer;      (* segment of input string which *)
                 finish : integer;     (* matches this syntactic structure *)
                 branches : integer;   (* number of children of this node *)
                 child : array [1 .. branches] of nodeptr
         end;
```

The order in which we search the tree is irrelevant, except that care must always be taken to search the largest subtree last.

```
function scan (parsetree: nodeptr, structure: string): nodeptr;
(* returns a pointer to that node of parsetree which
   represents the given structure *)
if parsetree = nil then return nil
else with parsetree↑ do
if structure = syntstruc then return parsetree
else find largest subtree maxsubtree;
   for i: =1 to branches do
      if i <> maxsubtree then t: = scan(child[i], structure);
                              if t <> nil then return t;
   return scan(child[maxsubtree], structure);
```

By scanning the smaller subtrees first, all subtrees which are placed on the stack cannot be larger than half the parsetree. Only the largest subtree can be greater than half, but that recursive call re-uses the current stack frame. Therefore scan uses at most $O(\log n)$ stack frames, where n is the length of the input string, compared to $O(n)$ in the worst case using the standard implementation.

MUTUAL RECURSION

So far, we have seen how the non-standard implementation can sometimes save a significant amount of storage space for recursive procedures. We have not yet used the full power of the implementation, since it applies quite generally to any procedure call, and not just a call to the currently invoked procedure. For example, consider procedures which check if there are an odd or even number of elements in the list:

```
function even(list);
(* returns TRUE iff list has an even number of elements *)
if list = nil then return true
          else return odd(list↑.next);
function odd(list);
(* returns TRUE iff list has an odd number of elements *)
if list = nil then return false
                else return even(list↑.next);
```

Again, the non-standard implementation will allow these procedures to execute using no stack space. A larger example is the Lisp[9] evalquote interpreter:

```
function evalquote (fn, args);
   return apply(fn, args, nil);

function apply(fn, args, a);
(* applies the function fn to its actual parameters args,
   using the variables and their values currently on the
   association list a *)
```

```
case fn of
      CAR : return caar(args);
      CDR : return cdar(args);
     CONS : return cons(car(args), cadr(args));
     ATOM : return atom(car(args));
       EQ : return eq(car(args), cadr(args));
   others : if atom(fn) then return apply(assoc(fn,a),args,a)
            else case car(fn) of
   LAMBDA : return eval(caddr(fn),pairlis(cadr(fn),args,a));
    LABEL : return apply(caddr(fn),args,cons(cons(cadr(fn),caddr(fn)),a));
function eval(form, a);
(* evaluates form using the association list a *)
if atom(form) then return assoc(form, a)
elseif car(form) = QUOTE then return cadr(form)
elseif car(form) = COND then return evcon(cdr(form), a)
elseif car(form) = RETURN then return eval(cdr(form), cdr(a))
elseif car(form) = RETURN2 then return eval(cdr(form), cddr(a))
else return apply(car(form), evlis(cdr(form),a), a);

function assoc(item, a);
(* returns the value of item in the association list a *)
if item = caar(a) then return cdar(a)
                 else return assoc(item, cdr(a));

function pairlis (items, values, a);
(* forms a 1 - 1 association between the items and values, placing
   them onto the association list a *)
if items = nil then return a
else return pairlis (cdr(items), cdr(values), cons(cons(car(items),car(values)),a));

function evcon(c,a);
(* evaluates the conditional expression c *)
if eval(caar(c),a) then return eval(cadar(c),a)
                   else return evcon(cdr(c),a);

function evlis(args,a);
(* evaluates the list of arguments, returning a corresponding list of values *)
if args = nil then return nil
              else return cons(eval(car(args),a),evlis(cdr(args),a));
```

It is interesting to note the large number of procedure calls which will not require any extra stack space, because the non-standard procedure call implementation will simply re-use the current stack frame. Assuming that the primitive functions such as *car* and *cdr* are compiled in-line, the only procedure calls which will use an extra stack frame are the call on *pairlis* in *apply*, *evlis* in *eval*, the first call on *eval* in *evcon*, and the calls on *eval* and *evlis* in *evlis*.

CONCLUSION

In cases where the natural expression of an algorithm is recursive, then considerations of good programming practice indicate that the recursive algorithm should be used. We have shown that objections to recursion based on wasteful use of stack storage are often

not valid, as a proper (and easy) implementation of procedure calls can reduce the space used to the minimum required by an iterative solution.

ACKNOWLEDGEMENT

Many thanks to Dan Johnston for thought-provoking discussions on recursion.

APPENDIX

A variant of the tree traversal routine given in the paper can be used to provide an alternative proof that Boolean formulae can be evaluated in O(*log n*) space.[11] The following procedure is called with *evaluate* (*formula*, **nil**, **none**).

```
procedure evaluate(node, origin, currentval);
(* evaluates the boolean formula node, having come from origin
   whose value is currentval *)
if node = nil then return currentval;
if node↑.left = nil then return evaluate(node↑.parent, node, node↑.datum);
if origin = node↑.parent then return evaluate(node↑.left, node, none);
if node↑.operator = NOT then return evaluate(node↑.parent, node, not
   currentval);
if origin = node↑.left and ((currentval and node↑.operator = AND)
                        or (not currentval and node↑.operator = OR))
                    then return evaluate(node↑.right, node, none);
return evaluate(node↑.parent, node, currentval);
```

Since each recursive call is preceded by **return**, evaluate uses only one stack frame which contains a few pointers to the Boolean formula. If the Boolean formula has length *n*, then the procedure uses only O(*log n*) working storage. Furthermore, if the formula is stored on an input tape in some standard encoding rather than as an explicit tree in the computer's working storage space, then the constructs *node*↑*.left*, *node*↑*.right and node*↑*.parent* can be computed by a simple counting algorithm using only O(*log n*) space).

REFERENCES

1. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, N.J., 1976.
2. J. S. Rohl, 'Why recursion?', *Proc. Symp. on Language Design and Programming Methodology*, Sydney, 1979. pp.71–83.
3. S. Swamy and J. Savage, 'Space-time trade-offs for linear recursion', *Conf. Record Sixth Annual ACM Symp. on Principles of Programming languages*, 1979. 135–142.
4. E. M. Gurari and O. H. Ibarra, 'On the space complexity of recursive algorithms', *Information Processing Letters*, **8**, 267–271 (1979).
5. D. E. Knuth, 'Structured programming with go to statements', *Computing Surveys*, **6**, 261–301 (1974).
6. W. Wulf *et al.*, *The Design of an Optimizing Compiler*, American Elsevier, New York, 1975.
7. B. E. Carpenter, R. W. Doran and K. Hopper, 'Non-recursive recursion', *Software—Practice and Experience*, 7, 263–269 (1977).
8. R. Haskell, 'Efficient implementation of a class of recursively defined functions', *Computer J.*, **18**, 23–29 (1975).
9. J. McCarthy *et al.*, *Lisp 1·5 Programmer's Manual*, MIT Press, 1962.
10. C. A. R. Hoare, 'Quicksort', *Computer J.*, **5**, 10–15 (1962).
11. N. Lynch, 'Log space recognition and translation of parenthesis languages', *JACM*, **24**, 583–590 (1977).