Recursion elimination with variable parameters

R. S. Bird

Department of Computer Science, University of Reading, Whiteknights Park, Reading RG6 2AX

Standard methods of recursion elimination are not immediately applicable to recursive procedures which possess variable (reference, name) parameters. Two methods of overcoming this problem are described. One method involves a transformation which replaces such parameters with constant ones, while the second involves the creation of a new level of reference variables, a device only possible in a language such as ALGOL 68. As examples of the techniques, iterative versions of a procedure to build a balanced tree are derived in both ALGOL 68 and PASCAL.

(Received January 1978)

1. Introduction

Among the growing number of techniques for efficient recursion elimination (Knuth, 1974; Bird, 1977; Partsch and Pepper, 1976) perhaps the simplest is the one embodied in the following rule:

'if the last action of procedure p before it terminates is to call procedure q, simply go to the beginning of procedure q instead'. This rule, which is discussed more fully in Bird (1977) and Knuth (1974), works just as well when $q = p$ and so can be used to replace terminal recursive calls by simple loops. For instance, we can translate the recursive procedure

```
proc P; if T then A; P fi
```

into the equivalent iterative version

```
proc P; while T do A.
```

The rule can also be used with procedures which possess constant (or value) parameters, the only modification necessary being the injunction to assign such parameters their new values just before the jump is made. To take an example in ALGOL 60, consider

```
procedure P(x); integer x; value x;
  if T(x) then begin A(x); P(f(x)) end
```

This translates into the iterative procedure

```
procedure P(x); integer x; value x;
  while T(x) do
    begin A(x); x := f(x) end
```

(Actually, ALGOL 60 does not permit **while** statements of this form, but the idea is clear enough). The semantics of ALGOL 60 guarantee that on entry to the procedure body a new local variable x is automatically created, so the assignment $x := f(x)$ is permissible. In ALGOL 68 we have to create this local variable ourselves, but otherwise the method is just the same. Thus

```
proc P = (int x) void:
  if T(x) then A(x); P(f(x)) fi
```

becomes

```
proc P = (int x) void:
  begin int y := x;
    while T(y) do
      A(y); y := f(y) od
  end.
```

The trouble with the rule arises as soon as we consider other forms of parameter passing, and it is the purpose of the present paper to explore this problem a little further. Consider the ALGOL 60 procedure

```
procedure P(x); integer x;
  if x ≠ 0 then P(L[x])
    else x := 1
```

Here x is a parameter called by name and the effect of a call $P(L[1])$ is to set $L[y] = 1$ where y is the first integer of the form $L[\ldots L[1] \ldots]$ such that $L[y] = 0$. It doesn't take much thought to see that in this case we cannot simply replace the procedure body with the code

```
while x ≠ 0 do x := L[x];
x := 1
```

for then the call $P(L[1])$ would have an entirely different effect.

In ALGOL 60 the solution to the problem lies in a preliminary transformation which changes parameters called by name into parameters called by value. For the above example, we first change procedure P into a procedure Q designed with the intention that the call $Q(x)$ should be equivalent to the call $P(L[x])$. Q has the definition

```
procedure Q(x); integer x; value x;
  if L[x] ≠ 0 then Q(L[x])
    else L[x] := 1
```

and recursion elimination can be applied to Q in the standard way. As long as the only calls to P are calls of the form $P(L[\ldots])$ no further problem arises; if not, then some additional copying into and out of a particular element of L has to be performed. This is perhaps a rather artificial example but we shall see the same principle at work on a much more natural example in Section 2.

There is an alternative way of attacking the problem, but it is only possible in a language such as ALGOL 68 which permits arbitrary levels of reference variables to be defined. The following section explains the idea.

2. Recursion elimination in ALGOL 68

One feature of ALGOL 68 which turns out to be useful in recursion elimination is the fact that for any object of mode **thing** one can create an object of mode **ref thing.** The example of an ALGOL 68 program given in the last section suggests a way in which this fact can be used. There, in order to eliminate the recursion from a procedure with a parameter of mode **int,** we had to create an object y of mode **ref int.** The same idea can be extended to the next level. Consider again

```
proc P = (ref int x) void:
  if x ≠ 0 then P(L[x])
    else x := 1 fi
```

In ALGOL 68 we can translate P directly into an iterative procedure as follows:

```
proc P = (ref int x) void:
  begin ref int y := x;
    while y ≠ 0 do y := L[y] od;
    ref int (y) := 1
  end
```

Here, y is an object of mode **ref ref int.** During the execution of the loop y is repeatedly assigned the address of $L[y]$. On termination, the instruction **ref int** $(y) := 1$ forces a dereferencing of y to an object of mode **ref int** and assigns to it the value 1. In other words, the contents of the final address stored in y is set to 1, and this is just what is wanted.

The same method can be used with further levels of refs and to illustrate this we shall consider an example in which both recursive procedures and variable parameters arise naturally. The problem deals with the creation of a perfectly balanced binary tree. We can define a binary tree in ALGOL 68 as follows:

```
mode node = struct (int key, ref node left, right);
mode tree = ref node;
tree null = nil.
```

The problem is to read n integers from the input and build them into a balanced tree. One procedure does the job (see Wirth, 1976, from which the problem was taken) can be given as follows:

```
proc build = (int n, ref tree t) void:
 begin int x, nl, nr;
  if n = 0 then t := null else
   read(x); t := heap node := (x, null, null);
   nl := ndiv2; nr := n − nl − 1;
   build(nl, left of t);
   build(nr, right of t)
  fi
 end
```

The first problem to tackle in eliminating the recursion from *build* is that the recursive form of *build* is not one which can be solved by exclusive use of the rule mentioned in the introduction. In addition, we have to make use of a stack. To see how this works, consider first the schematic procedure

```
proc B(x);
 if t(x) then A(x); B(fx); B(gx)
        else C(x)
 fi
```

of the same form as *build*, except that for the moment we suppose that x is a value parameter. In the direct method of recursion elimination (Bird, 1977), items x on the stack S record obligations to carry out procedure calls $B(x)$. In its first form the solution is as follows:

```
proc B(x);
 begin stack S; |S| := 0; S ⇐ x;
  repeat x ⇐ S;
   if t(x) then A(x);
                S ⇐ gx;
                S ⇐ fx
   else C(x) fi
  until |S| = 0
 end
```

The notations $S \Leftarrow x$, $x \Leftarrow S$ are used as abstract representations of the operations of inserting x on top of the stack and removing the top item and assigning it to x, respectively; further $|S|$ denotes the length of the stack. An improvement can immediately be made to this solution by noticing that the record fx is placed on the stack only to be removed at the very next step. Thus we can change $S \Leftarrow fx$ to $x := fx$ and return control to the point just after the operation $x \Leftarrow S$ (this device is the counterpart to our original rule in more complicated recursions). So the second version of the solution is

```
proc B(x);
 begin stack S; |S| := 0; S ⇐ x;
  repeat x ⇐ S;
   while t(x)
    do A(x); S ⇐ gx; x := fx od;
    C(x)
  until |S| = 0
 end
```

Returning to the procedure *build*, we can now give the iterative solution

```
proc build = (int n, ref tree t) void:
 begin int m, nl, nr; ref tree p; stack S := empty;
  S ⇐ (n, t);
  repeat (m, p) ⇐ S;
   while m ≠ 0 do
    begin read (x);
     ref tree (p) :=
     heap node := (x, null, null);
     nl := mdiv2; nr := m − nl − 1;
     S ⇐ (nr, right of p);
     m := nl; p := left of p
    end;
   ref tree (p) := null
  until S is empty
 end
```

We can create a stack in ALGOL 68 with the structure definitions

```
mode cell = struct(int num, ref tree ash, ref cell next);
mode stack = ref cell;
stack empty = nil;
```

and expand the operation $S \Leftarrow (n, t)$ into

S := **heap cell** := (n, t, S)

and the operation $(m, p) \Leftarrow S$ into

m := *num* **of** S; p := *ash* **of** S; S := *next* **of** S.

When these substitutions are carried out we are left with an iterative ALGOL 68 version of *build* (apart from the fact that the **repeat** . . . **until** construct is not legal ALGOL 68). Notice, in particular, that the mode of p is **ref ref ref node.**

It is arguable whether or not the above solution is at all comprehensible taken by itself; it certainly has been derived in a hopefully comprehensible manner from an intuitively simple procedure. The introduction of refs for purposes of recursion elimination closely parallels the introduction of gotos for the same purpose, and gotos in Dijkstra's famous phrase 'are too much of an invitation to make a mess of one's program'. Nevertheless, the fact that the ref device can be used at all is a remarkable testament to the flexibility of ALGOL 68.

It is instructive to compare this method of elimination with the one which eliminates variable parameters first. In the next section we take up the problem of building a balanced tree again, but this time using the language PASCAL to express both the recursive and iterative versions.

3. Recursion elimination in PASCAL

The language PASCAL does not easily permit arbitrary levels of reference variables, so it is a reasonable vehicle in which to study the second method of recursion elimination. We could have chosen to stay with ALGOL 68, but the *build* example occurs in Wirth's (1976) excellent book on programming, together with the exhortation to the reader to use his ingenuity in writing a non-recursive equivalent. Wirth appends such a program without further comments to serve as a challenge for the reader to discover how and why it works. Since the primary object of this section is to systematically derive Wirth's iterative version, it is only natural that we should do so in PASCAL.

One way of defining a binary tree in PASCAL is given by

```
type tree = ↑node;
     node = record key: integer;
```

```
        left, right: tree
    end
```

The procedure for building a balanced tree is

```
procedure build(n: integer; var t: tree);
  var x, nl, nr: integer;
  begin if n = 0 then t := nil else
    begin read(x); new(t);
      t↑·key := x;
      nl := ndiv2; nr := n − nl − 1;
      build(nl, t↑·left);
      build(nr, t↑·right)
    end
  end
```

Although one can define variables of type pointer-to-tree in PASCAL, and so carry out recursion removal in the manner of the last section, we choose instead to eliminate the variable parameter *t* from *build* and do the recursion removal according to the first method. This is achieved by splitting *build* into two mutually recursive procedures *buildright* and *buildleft*, designed with the intention that

$$buildright(n, t) \equiv build(n, t\uparrow\cdot right)$$
$$\text{and } buildleft(n, t) \equiv build(n, t\uparrow\cdot left)$$

The definitions are very similar so we shall just give the definition of *buildright*:

```
procedure buildright(n: integer; t: tree);
  var q: tree;
    x, nl, nr: integer;
  begin if n = 0 then t↑·right := nil else
    begin read(x); new(q);
      t↑·right := q;
      q↑·key := x;
      nl := ndiv2; nr := n − nl − 1;
      buildleft(nl, q);
      buildright(nr, q)
    end
  end
```

These two procedures can be used in place of *build* provided we change every procedure call *build*(n, *root*1) into a call *buildright*(n, *root*2) where *root*2 is a new node with *root*2↑·*right* = *root*1. (Of course, we could equally well have chosen *buildleft* as the 'dominant' procedure).

Having eliminated the variable parameter we can now go on to the recursion elimination stage. This has a number of interesting features as we are dealing with two mutually recursive procedures. To see what is involved, consider first the following schematic procedures of the same form as *buildleft* and *buildright*:

```
proc R(x);
  if p(x) then A(x) else B(x); L(fx); R(gx) fi
proc L(x);
  if p(x) then C(x) else D(x); L(fx); R(gx) fi
```

Once again, to solve these procedures we have to invoke the use of a stack. In the present case, items on the stack take the form $(1, x)$ and $(2, x)$ and signify obligations to carry out the procedure calls $R(x)$ and $L(x)$ respectively. If we are interested in evaluating $R(x_0)$, the direct method of elimination yields the solution

```
S ⇐ (1, x0);
repeat (b, x) ⇐ S;
  if b = 1 then if p(x) then A(x)
                else B(x); S ⇐ (1, gx); S ⇐ (2, fx) fi
           else if p(x) then C(x)
                else D(x); S ⇐ (1, gx); S ⇐ (2, fx) fi
  fi
until |S| = 0.
```

We can improve this solution to read

```
S ⇐ (1, x0);
repeat (b, x) ⇐ S;
  if b = 1 then if p(x) then A(x)
                else B(x); S ⇐ (1, gx); x := fx; goto L fi
           else L: if p(x) then C(x)
                else D(x); S ⇐ (1, gx); x := fx; goto L fi
  fi
until |S| = 0,
```

since the record $(2, fx)$ is placed on the stack only to be removed at the next step. Observe now that all records have the form $(1, x)$, so there is no longer any need for the tag 1 to signify that it is $R(x)$ we wish to evaluate. The final version of the solution is

```
S ⇐ x0;
repeat x ⇐ S;
  if p(x) then A(x)
  else B(x); S ⇐ gx; x := fx;
    while p(x) do
      begin D(x); S ⇐ gx; x := fx end;
    C(x)
  fi
until |S| = 0.
```

This solution can now be used to solve the original *build* procedure with the substitutions

(n, t) for x_0, $p\uparrow\cdot right$:= **nil** for $A(x)$
(m, p) for x, $p\uparrow\cdot left$:= **nil** for $C(x)$
$m = 0$ for $p(x)$,
and *read*(x); *new*(q); $q\uparrow\cdot key := x$;
$p\uparrow\cdot right := q$;
$nl := m$**div**2; $nr := m - nl - 1$;

for $B(x)$. The code for $D(x)$ is just the same as for $B(x)$ except that $p\uparrow\cdot left := q$ replaces $p\uparrow\cdot right := q$. Finally,

$S \Leftarrow gx$ translates to $S \Leftarrow (nr, q)$
and $x := fx$ translates to $m := nl; p := q$

Once these substitutions are incorporated, we obtain the procedure

```
procedure buildright (n: integer, t: tree);
  var p, q: tree;
    x, m, nl, nr: integer;
    S : stack;
  begin S ⇐ (n, t);
    repeat (m, p) ⇐ S;
      if m = 0 then p↑·right := nil
      else begin read(x); new(q); q↑·key := x;
        p↑·right := q;
        nl := mdiv2; nr := m − nl − 1;
        S ⇐ (nr, q);
        p := q; m := nl;
        while m ≠ 0 do
          begin read(x); new(q); q↑·key := x;
            p↑·left := q;
            nl := mdiv2; nr := m − nl − 1;
            S ⇐ (nr, q);
            p := q; m := nl
          end;
        p↑·left := nil
      end
    until |S| = 0
  end
```

Though correct, the above program contains too much duplication of code for us to be happy with it as a final version. If the inner loop can be changed to a repeat loop, the resulting program will be shorter and more satisfying. We can move the assignment $p\uparrow\cdot right := q$ to the end of the loop by (i) changing all subsequent occurrences of p to a new variable r, thereby saving the value of p; and (ii) saving the value of q in a suitable manner. The second task is accomplished very neatly by having one extra node *link* to hold the value of q in its *left* field. By initialising r to *link* and setting $r\uparrow \cdot left$ to q we not only manage to completely duplicate the body of the while loop, thus reducing it to a repeat loop, but also preserve the initial value of q in $link\uparrow \cdot left$. The final program is therefore as follows:

```
procedure buildright(n: integer, t: tree);
 var p, q, r, link: tree;
  x, m, nl, nr: integer;
  S: stack;
 begin new(link); |S| := 0; S ⇐ (n, t);
  repeat (m, p) ⇐ S;
   if m = 0 then p↑·right := nil else
   begin r := link;
    repeat read(x); new(q); q↑·key := x;
     nl := mdiv2; nr := m − nl − 1;
     S ⇐ (nr, q);
     m := nl; r↑·left := q; r := q
    until m = 0;
    r↑·left := nil; p↑·right := link↑·left
   end
  until |S| = 0
 end
```

This procedure is essentially the one given by Wirth in (1976).

4. Results and conclusions

One may justifiably ask whether or not the energy spent on recursion elimination leads to significant gains in efficiency. To answer this question, the three versions of the balanced tree procedure were coded in ALGOL 68-R and run on the University of Reading's 1904S computer together with a timing program. The following table shows the time in seconds required to build a balanced tree of 250 nodes. In the table, *build* refers to the recursive version, *build*1 to the iterative version which uses the ref device and *build*2 to the iterative version given in Section 3. Both *build*1 and *build*2 were coded in two ways; in the first the stack S was represented by a structured linked list, while in the second two linear arrays were used. The results were:

build	*build*1	*build*2
0·072	0·081	0·088
	0·066	0·072

Clearly, the table tells a somewhat disappointing story; only the array version of *build*1 managed to beat the recursive procedure and then by only about 10%. The reason is that, although the balanced tree algorithm possesses a running time which is linear in the number of recursive calls, this is completely dominated by the time spent on manipulating the ALGOL 68 heap. Certainly it does not seem a good idea to involve the heap again by representing the stack as a linked list.

Nevertheless, the problem of recursion elimination is a useful vehicle in which to study and broaden our knowledge of program transformations and a practically useful one for the machine code and FORTRAN programmer who remains obliged to manufacture his or her own implementation of recursion.

Acknowledgements

The author would like to thank Dr J. Ogden for many stimulating discussions on the pros and cons of ALGOL 68 as well as making available his timing program. Thanks are also due to Dr J. Roberts for similar discussions on PASCAL.

References

BIRD, R. S. (1977). Notes on recursion elimination, *CACM*, Vol. 20 No. 6, pp. 434-439.
KNUTH, D. E. (1974). Structured programming with goto statements, *ACM Computing Surveys*, Vol. 6, pp. 261-302.
PARTSCH, H. and PEPPER, P. (1976). A family of rules for recursion removal, *Information Processing Letters*, Vol. 5 No. 6, pp. 174-177.
WIRTH, N. (1976). *Algorithms + Data Structures = Programs*, Prentice-Hall.

Book review

Computer-Aided Design of Digital Systems, by D. Lewin, 1977; 313 pages. (*Edward Arnold*, £15·00)

This book will be of interest mainly to students of computer science at a postgraduate level, and to those practising engineers who are already familiar with 'formal' logic design methods. The book surveys the current status of computer aids in the fields of logic network synthesis, logic simulation and logic testing. The subject of system specification, both by means of register transfer languages and by graph theoretic models is also discussed.

The main barrier to the more widespread acceptance of computer aided logic design is the lack of sufficiently powerful algorithms, especially methods applicable to circuits using MSI and LSI components. Most of the algorithms described in this book are orientated toward design using flipflops and discrete NAND and NOR gates; this limits its usefulness to designers of CAD systems in industry, who have to work with the current technology.

The longest chapter in this book (117 pages) covers the topic of logic network synthesis. A large number of algorithms (over 20) are described, for state reduction, state assignment and for implementation of the resulting switching functions in a particular 'logic family'. The algorithms described first appeared in a variety of journals, Ph.D. theses, etc; this book serves the useful purpose of collecting and comparing such a diversity of methods. The algorithms are described in Professor Lewin's usual lucid style and a large number of helpful worked examples are provided. Many of these algorithms suffer severe limitations on the size of problem which they can handle. Unfortunately, little numeric information is provided in this book to indicate to the reader the limitations of each technique.

The chapter on system specification contains an up-to-date survey of hardware description languages and also describes more recent developments, e.g. the use of Petrinets. The subjects of logic simulation for design verification and for test-program validation are treated in rather less detail; for example, the techniques of deductive fault simulation and the use of worst case timing are mentioned but not described in detail. The book concludes with a review of the subjects of logic circuit testing and testable logic design. The book provides a large number of references (nearly 300), and a useful subject and authors index.

D. BUMSTEAD (Poole)