Proof of a recursive program: Quicksort

M. Foley and C. A. R. Hoare

Department of Computer Science, Queen's University, Belfast

This paper gives the proof of a useful and non-trivial program, Quicksort (Hoare, 1961). First the general algorithm is described informally; next a rigorous but informal proof of correctness of the coded program is given; finally some formal methods are introduced. Conclusions are drawn on the possibility of enlisting mechanical aid in the proof process.

(Received January 1971)

1. Introduction

It has been suggested (Hoare, 1971a, 1969, 1971b) that the advancement of the art of proving programs may lead to a reduction of the nuisance of programming error in the development and use of computer programs. We attempt here to show how a realistic program which incorporates recursion may be proved.

We also attempt to illustrate how the proof of a program can take advantage of a previously published proof; in particular, the proof of any procedure which it calls. This gives grounds for hope that the labour of program proving may eventually be reduced in the same way as that of mathematical theorem proving, by building up on the work of others rather than starting from scratch on each occasion.

A third objective has been to illustrate a method of annotating a program by comments in such a way that a keen and experienced reader may verify the correctness of the program by inspection and study rather than by poring over tedious and often trivial proofs.

Fourthly, an attempt is made to illustrate the adequacy of the formal rules of inference described by Hoare (1971b) by applying them to Quicksort. Finally, it is suggested that the formalisation of proof methods is a possible basis on which a computer can be programmed to assist in the construction and verification of proofs.

2. Description of Quicksort

2.1. *Criterion of correctness*

The purpose of the program Quicksort is to sort the elements $A[m]$ to $A[n]$ of an array into ascending order, while leaving untouched those below $A[m]$ and above $A[n]$. The desired result of the program is described by two terms. The first states that the elements from $A[m]$ to $A[n]$ are in ascending order

$$\forall p, q (m \leqslant p \leqslant q \leqslant n \supset A[p] \leqslant A[q])$$

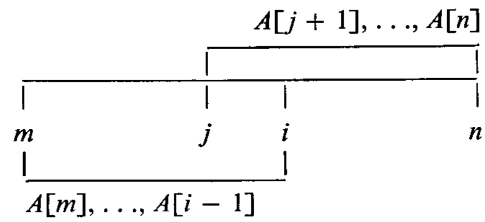
This may be abbreviated as Sorted (A, m, n). The second term states that the sorted array is equal to the original array for elements below m and above n; and between m and n it has the same elements as the original array but not necessarily in the same order. If A_0 is the initial value of the array, this may be expressed 'A is an $m - n$ permutation of A_0', or more briefly:

$$\text{Perm } (A, A_0, m, n).$$

Note that if $n < m$, Perm (A, A_0, m, n) is taken to assert that A is identical to A_0. Our proof will rely on a knowledge of the elementary properties of permutations; and there is therefore no need to define the concept of an $m - n$ permutation in greater detail.

2.2. *Outline of method*

Quicksort (A, m, n) works as follows. The elements between $A[m]$ and $A[n]$ are rearranged into two partitions such that those in the lower partition, $A[m], \ldots, A[i-1]$ are less than or equal to those in the upper partition, $A[j+1], \ldots, A[n]$, where $j < i$



The first rearrangement of elements between m and n is achieved by a call of a procedure Partition which has parameters A, i, j, m and n. Noting that elements between j and i are already in their correctly sorted positions it is fairly obvious that the entire array will be sorted if the elements between m and j and those between i and n are sorted. This is achieved by two recursive calls of Quicksort with parameters (m, j) and (i, n) respectively. Provided that a partition containing less than two elements is recognised as already sorted, and neither partition is ever as large as the original area to be sorted, this recursion will successfully terminate.

3. The procedure partition

3.1. *Description*

3.1.1. *Criterion of correctness*

The procedure Partition rearranges the elements $A[m]$ to $A[n]$ of an array into two parts, one of smaller and one of larger elements as described in Section 2. The criterion of correctness consists of two terms. The first states the necessary ordering relation

$$j < i \,\&\, \forall p, q\, (m \leqslant p < i \,\&\, j < q \leqslant n \supset A\,[p] \leqslant A[q])$$

This is abbreviated as Partd (A, i, j, m, n). The second is

$$\text{Perm } (A, A_0, m, n)$$

which states that the partitioned array must be an $m - n$ permutation of the original array A_0.

3.1.2. *Outline of method*

The division into smaller and larger elements is done by selecting an arbitrary element, say r, and placing elements smaller than it in the lower partition and elements larger than it in the upper partition. i is initially set to m and j to n. Then i is stepped up for as long as $A[i] < r$, since these elements belong to the

*Except possibly in certain 'critical regions'.

lower partition and may be left in position. When an $A[i]$ is encountered which is not less than r and hence out of place, the stepping up of i is interrupted. The value of j is then stepped down while $r < A[j]$, and this stepping down is interrupted when an $A[j]$ not greater than r is met. The current $A[i]$ and $A[j]$ are now both in the wrong partitions, which situation is corrected by exchanging them. If $i \leqslant j$, i is stepped up and j stepped down by one, and i search and j search is continued until the next out of place pair is found. If $j < i$ the lower and upper parts overlap and the partition is complete. It may be seen that these final values of i and j are not specified in advance but are set by the procedure.

3.2. *Informal proof*

The correctness of the body of the procedure Partition has been informally established by Hoare (1971a). However the version used in Quicksort is slightly different as it is written as a procedure and the arbitrary element r is taken as $A\left[\frac{m+n}{2}\right]$.

Also Perm (A, A_0, m, n) was not directly proved by Hoare (1971a). Hence some additions will be necessary to complete the proof.

An annotated version of Partition may be written as

```
comment Let A₀ be the initial value of an array A. The pro-
        cedure rearranges the elements between m and n so
        that
j < i & ∀p, q (m ≤ p < i & j < q ≤ n ⊃ A[p] ≤ A[q])
                                        & Perm (A, A₀, m, n)
        provided that m < n on entry.
Partition (A, i, j):(m, n) proc
begin new r, f;
    comment Perm (A, A₀, m, n)                    A—invariant
            & m ≤ f ≤ n                           f—invariant
            & m ≤ i & ∀p(m ≤ p < i ⊃ A[p] ≤ r)
                                                  i—invariant
            & j ≤ n & ∀q(j < q ≤ n ⊃ r ≤ A[q]):
                                                  j—invariant;
            f := (m + n)/2; r := A[f]; i := m; j := n;
            while i ≤ j do
            begin while A[i] < r do i := i + 1;
                  while r < A[j] do j := j − 1;
            comment A[j] ≤ r ≤ A[i];
            if i ≤ j then
            begin new w; w := A[i]; A[i] := A[j];
                      A[j] := w;
                  comment A[i] ≤ r ≤ A[j];
                  i := i + 1; j := j − 1;
                  end
            end
end
```

The annotations for a program include the criterion of correctness and propositions at certain points which are true each time control reaches that point. A proposition expressing the purpose of a variable is known as an invariant and is intended to remain true throughout the execution* of the program even when the value of the variable concerned is changed by assignment.

In the second **comment** invariants corresponding to the variables A, f, i and j are given. The proof that those for i and j are invariant over the main loop of Partition is given in lemmas 8, 9, 10 and 11 by Hoare (1971a) with m substituted for 1 and n for N. The f-invariant is unchanged by the loop since f is not reassigned within it. It remains to show the following three results:

(*a*) Perm (A, A_0, m, n) is invariant over the main loop.

The only assignments to array members are in the group in the conditional '**if** $i \leqslant j$ **then** . . .'. Hence only these assignments could cause Perm (A, A_0, m, n) to become untrue. If A' is the value of A after these assignments then, as given by Hoare (1971a)

$$A'[i] = A[j]$$
$$A'[j] = A[i]$$

and

$$\forall s\,(s \neq i \;\&\; s \neq j \supset A'[s] = A[s])$$

Since at this point $m \leqslant i \leqslant j \leqslant n$ it is obvious that A' is an m–n permutation of A and hence the truth of Perm (A, A_0, m, n) is preserved. (The composition of two permutations is a permutation.)

(*b*) The initial values of the variables satisfy the invariants i.e.

(i) $m \leqslant \frac{m+n}{2} \leqslant n$

(ii) $m \leqslant m \;\&\; \forall p(m \leqslant p < m \supset A[p] \leqslant r)$

(iii) $n \leqslant n \;\&\; \forall q(n < q \leqslant n \supset r \leqslant A[q])$

(i) follows from the precondition $m < n$. (ii) and (iii) are obviously true since the antecedents of the implications are false.

(*c*) The criterion of correctness is true after execution.

On exit from the main loop of Partition, $j < i$ and the invariants are still true. Hence it is required to prove that

$$j < i \;\&\; m \leqslant f \leqslant n \;\&\; m \leqslant i \;\&\; Vp(m \leqslant p < i \supset A[p] \leqslant r)$$
$$\&\; j \leqslant n \;\&\; \forall q(j < q \leqslant n \supset r \leqslant A[q]) \;\&\; \text{Perm}\,(A, A_0, m, n)$$
$$\supset j < i \;\&\; Vp, q(m \leqslant p < i \;\&\; j < q \leqslant n \supset A[p] \leqslant A[q])$$
$$\&\; \text{Perm}\,(A, A_0, m, n)$$

which is an obvious result, following from the transitivity of $\leqslant$: hence if $A = A_0$ and $m \leqslant n$ before a call of Partition $(A, i, j):(m, n)$, Partd (A, i, j, m, n) and Perm (A, A_0, m, n) will be true after execution.

4. Informal proof of Quicksort

An annotated version of Quicksort is as follows.

```
Comment Let A₀ be the initial value of an array A. Quicksort
        (A, m, n) rearranges the elements between m and n
        so that
          Sorted (A, m, n) & Perm (A, A₀, m, n)
Quicksort (A): (m, n) proc
if m < n then
  begin new i, j;
        comment m < n: and let A = A₀ here;
        call Partition (A, i, j):(m, n);
            comment let A = A₁ here:
                    Partd (A₁, i, j, m, n) &
                        Perm (A₁, A₀, m, n);
        call Quicksort (A): (m, j);
            comment let A = A₂ here:
                    Sorted (A₂, m, j) & Perm (A₂, A₁, m, j);
        call Quicksort (A): (i, n);
            comment Sorted (A, i, n) & Perm (A, A₂, i, n);
  end
```

The annotations following the recursive calls require some explanation. In order to prove a recursive program it is necessary to assume that the recursive calls work, and then prove that the program body works based on these assumptions. Such an assumption means that (a suitably modified version of) the criterion of correctness is true after each recursive call. Thus Sorted (A_2, m, j) & Perm (A_2, A_1, m, j) is assumed to be true after the first call of Quicksort and Sorted (A, i, n) & Perm (A, A_2, i, n) after the second.

The first requirement is to prove that the criterion of correctness is true when the body of the main conditional is not executed at all, i.e. when $n \leqslant m$. In this case, the truth of Perm (A, A_0, m, n) follows from $A = A_0$ by the definition of Perm. The truth of

$$\forall p, q(m \leqslant p \leqslant q \leqslant n \supset A[p] \leqslant A[q])$$

follows directly from the unsatisfiability of the antecedent.

Next we note that the variables A_1 and A_2 denote 'snapshots' of the specific values of A at certain points. These values are fixed at the time of the snapshot so that the propositions containing them remain true independently of subsequent actions of the program. Hence these propositions are still true at the end of the program, and it must be proved that their conjunction implies

(i) Sorted(A, m, n) and
(ii) Perm(A, A_0, m, n)

The proof of (ii) is given by

$$m \leqslant i \;\&\; j \leqslant n \;\&\; \text{Perm}\,(A_1, A_0, m, n) \;\&\; \text{Perm}\,(A_2, A_1, m, j) \;\&\; \text{Perm}\,(A, A_2, i, n) \supset \text{Perm}\,(A, A_0, m, n)$$

This follows from the fact that an $m - j$ or $i - n$ permutation is also an $m - n$ permutation for $m \leqslant i$ and $j \leqslant n$; and that the composition of three $m - n$ permutations is itself an $m - n$ permutation. To prove (i) it will first be established that (*a*) the partitioning of the array is not disturbed by the subsequent sorts and (*b*) the sorting between m and j is not disturbed by the second sort.

(*a*) $\text{Partd}\,(A_1, i, j, m, n) \;\&\; \text{Perm}\,(A_2, A_1, m, j) \;\&\; \text{Perm}\,(A, A_2, i, n) \supset \text{Partd}\,(A, i, j, m, n)$

Consider an element $A[k]$ for $k < j$ and an element $A[l]$ for $i < l$. We need to prove that $A[k] \leqslant A[l]$.

Since A is an $i - n$ permutation of A_2 and $k < j < i$, $A[k] = A_2[k]$; and there is an l', $(i \leqslant l' \leqslant n)$ such that $A[l] = A_2[l']$. Since A_2 is an $m - j$ permutation of A_1 and $j < i \leqslant l'$, $A_2[l'] = A_1[l']$; and there is a $k'(m \leqslant k' \leqslant j)$ such that $A_2[k] = A_1[k']$. Since A_1 is partitioned, and $m \leqslant k' \leqslant i < i$ and $j < i \leqslant l' \leqslant n$, it follows that:

$$A[k] = A_1[k'] \leqslant A_1[l'] = A[l]$$

(*b*) $j < i \;\&\; \text{Sorted}\,(A_2, m, j) \;\&\; \text{Perm}\,(A, A_2, i, n) \supset \text{Sorted}\,(A, m, j)$

A is an $i - n$ permutation of A_2 and thus is equal to A_2 for elements below i. Since $j < i$ A is equal to A_2 for elements between m and j. Thus Sorted (A, m, j) follows from sorted (A_2, m, j).

It now remains to prove that Partd (A, i, j, m, n) & Sorted (A, m, j) & Sorted $(A, i, n) \supset$ Sorted (A, m, n). The proof of this will be given in greater detail since all the terms may be expanded. On expansion the lemma becomes

(i) $j < i$
(ii) $\& \;\forall p, q(m \leqslant p < i \;\&\; j < q \leqslant n \supset A[p] \leqslant A[q])$
(iii) $\& \;\forall p, q(m \leqslant p \leqslant q \leqslant j \supset A[p] \leqslant [q])$
(iv) $\& \;\forall p, q(i \leqslant p \leqslant q \leqslant n \supset A[p] \leqslant A[q])$
$\supset \forall p, q(m \leqslant p \leqslant q \leqslant n \supset A[p] \leqslant A[q])$

A proof by cases is now given. The following is a simple but tedious theorem of ordering.

$$\begin{aligned} j < i \;\&\; m \leqslant p \leqslant q \leqslant n \supset\; & m \leqslant p < i \;\&\; j < q \leqslant n \\ & \vee\, m \leqslant p \leqslant q \leqslant j \\ & \vee\, i \leqslant p \leqslant q \leqslant n \end{aligned}$$

In each of the three cases of the consequent of this theorem, either (ii), (iii), or (iv) state that $A[p] \leqslant A[q]$.

The body of Quicksort has been proved correct based on the assumption that the recursive calls work. Hence Quicksort is correct and if $A = A_0$ initially, Sorted (A, m, n) & Perm (A, A_0, m, n) will be true after execution.

5. Formal proof

In formalising the proof given in the previous section, we are not interested in giving a formal proof of the lemmas of the previous section; that may be done (if desired) by the familiar apparatus of mathematical logic. However, it does seem worthwhile to formalise the relationship between the lemmas proven in the domain of mathematics, and the program itself, written to be executed on a computer. This will both help to reassure us that the formulation of the lemmas validly reflects the correctness of the program; it will also illustrate the adequacy of the proof techniques described by Hoare (1971b) for treating a realistic program. The formal proof is given in Appendix 1; the notations it uses are explained by Hoare (1969) and the full set of inference rules required is reproduced in Appendix 2.

As with the informal approach it is necessary to assume that the recursive calls are correct and then on this basis prove that the program body is correct; thus the theorem to be proved is used as a hypothesis in the proof of the program body. This hypothesis is line 3 of the proof of Quicksort. Using it the program body is proved correct (line 12), and the desired result (line 13), follows by the rule of recursion. The theorem of line 1 gives the result established informally for the procedure partition.

6. The lemma generator

Formal proofs such as those of Appendix 1 are tedious to write and check. Their only purpose is to expose the lemmas on which the proof of correctness depends, e.g. line 11 in the proof of Quicksort. Hence it would be useful to have a mechanical means of generating these lemmas from the text of the program. Such a mechanical procedure is possible provided that certain comment information is given in addition to the program text as in Section 3.2. This information includes the criterion of correctness of the program as a whole, and a sufficiently powerful invariant for each loop of the program. If the correctness of the program depends on an initial precondition (e.g. $m < n$ in the case of Partition) this must be given, and if the program contains a procedure call the theorem expressing the correctness of this procedure must also be supplied.

The lemma generator works as follows. For each command type there is a rule of inference permitting the mechanical construction of the 'weakest' proposition which is 'provably' true before execution of the command if a certain proposition is true after it. Consider assignment, for example. The rule for assignment is

$$R_e^x \{x := e\}\; R$$

If the proposition R is true after the assignment $x := e$ then R with e substituted for x must have been true before it.

The criterion of correctness specified by the programmer must be true at the end of a program. Using the relevant rule of inference a machine can construct a proposition true before execution of the last command. This proposition can then be 'moved back' through the penultimate command and so on for each command of the program in turn. Eventually the weakest proposition true before execution of the first command will be produced. Call it 'Precondition'. If 'Initial' is required to be true before execution one of the basic lemmas produced by the machine will be

$$\text{Initial} \supset \text{Precondition}$$

The machine will also produce lemmas for each loop of the program in accordance with the Rule of Iteration.

It is fairly obvious how the basic lemma (line 11) in the proof of Quicksort could be generated, given lines 1 and 13. The Rule of Adaptation is used to generate the weakest proposition true before the recursive and non-recursive procedure calls. Thus the lemma generator would begin with line 7, using this rule to move the criterion of correctness back through the second recursive call to give the proposition $L7$. This application of the rule uses line 6, which may be derived mechanically by substitution from line 3. The parameters of the recursive call are substituted in the hypothesis for the formal parameters of the program, and the variables to be existentially quantified are given different names if they clash with any variables in the proposition to be moved back through the recursive call. For example A_0 occurs on the right of line 7 and hence must be replaced by a different variable (i.e., A_2) in the substituted version of the hypothesis.

$L7$ would then be moved back through the first recursive call to give $L5$ and $L5$ moved through the call of Partition to give $L2$. $L2$ moved back through the conditional would give the 'Precondition', $L10$. Since for Quicksort 'Initial' is '$A = A_0$' the lemma produced would be (line 11):

$$A = A_0 \supset \textbf{if } m < n \textbf{ then } L2 \textbf{ else } \text{Sorted}(A, m, n) \; \& \; \text{Perm}(A, A_0, m, n)$$

N.B. Note that this 'Initial' does not impose necessary constraints on the initial values of the program variables. It is merely a 'snapshot' necessary to define the A_0 used in the criterion of correctness. Only one lemma would be generated for Quicksort since it contains no loops.

Written in full this lemma is

$$\begin{aligned}
&A = A_0 \supset \textbf{if } m < n \textbf{ then}\\
&\quad \exists A_0(A = A_0 \;\&\; m < n \;\&\; VA, i, j(\text{Partd}(A, i, j, m, n) \;\&\; \text{Perm}(A, A_0, m, n) \supset\\
&\quad \exists A_1(A = A_1 \;\&\; VA(\text{Sorted}(A, m, j) \;\&\; \text{Perm}(A, A_1, m, j) \supset\\
&\quad \exists A_2(A = A_2 \;\&\; VA(\text{Sorted}(A, i, n) \;\&\; \text{Perm}(A, A_2, i, n) \supset\\
&\qquad \text{Sorted}(A, m, n) \;\&\; \text{Perm}(A, A_0, m, n))))))))\\
&\quad \textbf{else } \text{Sorted}(A, m, n) \;\&\; \text{Perm}(A, A_0, m, n)
\end{aligned}$$

By eliminating quantifiers and performing other obvious simplifications this becomes

(a) $\neg m < n \supset \text{Sorted}(A, m, n) \;\&\; \text{Perm}(A, A_0, m, n)$

(b) $m < n \;\&\; \text{Partd}(A_1, i, j, m, n) \;\&\; \text{Perm}(A_1, A_0, m, n)$
$\;\&\; \text{Sorted}(A_2, m, j) \;\&\; \text{Perm}(A_2, A_1, m, j)$
$\;\&\; \text{Sorted}(A, i, n) \;\&\; \text{Perm}(A, A_2, i, n)$
$\;\supset \text{Sorted}(A, m, n) \;\&\; \text{Perm}(A, A_0, m, n)$

which is the same as the lemma derived informally from the annotated program.

7. Conclusion

The lemmas on which proof depends may be generated by machine as has been indicated. However this can be regarded as merely isolating the problem, as for complex programs the proof of the lemmas will be the major part of the proof of correctness. This suggests that mechanical aids in proving the lemmas should therefore also be sought.

The obvious first suggestion is an automatic theorem prover. King (1969) has successfully proved small programs using such an aid, but general purpose theorem provers are not powerful enough to handle complex lemmas. Another possibility is the use of proof checking rather than proof generation. The machine is supplied with an abbreviated proof of a lemma and then fills in the gaps and checks the complete proof. Such an approach has been implemented by Abrahams (1963) but again is so far satisfactory only for simple examples. Finally there is the possibility of constructing proofs by some form of man–machine co-operation. This seems a promising approach. Good (1970) suggests that the answer lies in combining simplification methods, special purpose automatic theorem provers (preferably decision procedures) and man–machine interaction. Burstall (1970) has used such interaction to guide a resolution-based theorem prover, but reports that the process was rather laborious.

Acknowledgement

This work was carried out with the aid of a grant from the Ministry of Education, Northern Ireland.

Appendix 2

$$R^x_e\{x := e\}\, R \qquad \text{Assignment}$$

$$\frac{P\{Q\}S \quad S \vdash R}{P\{Q\}R} \qquad \frac{P \vdash S \quad S\{Q\}R}{P\{Q\}R} \qquad \text{Consequence}$$

$$\frac{P\{Q_1\}S,\; S\{Q_2\}R}{P\{Q_1;\, Q_2\}R} \qquad \text{Composition}$$

$$\frac{P \supset \textbf{if } B \textbf{ then } P_1 \textbf{ else } R,\; P_1\{Q\}P}{P\{\textbf{while } B \textbf{ do } Q\}R} \qquad \text{Iteration}$$

$$\frac{P\{Q\}R}{\textbf{if } B \textbf{ then } P \textbf{ else } R\{\textbf{if } B \textbf{ then } Q\}R} \qquad \text{Condition}$$

$$\frac{p(\mathbf{x}):(\mathbf{v})\ \textbf{proc}\ Q \qquad P\{\textbf{call}\ p(\mathbf{x}):(\mathbf{v})\}R \;\vdash P\{Q\}R}{P\{\textbf{call}\ p(\mathbf{x}):(\mathbf{v})\}R} \qquad \text{Recursion}$$

$$\frac{P\{\textbf{call}\ p(\mathbf{x}):()\mathbf{v}\}R}{P\{\mathbf{c}^{k,x,}_{k',a,e}\ \textbf{call}\ p(\mathbf{a}):(\mathbf{e})\}R^{k,x,v}_{k'a,e,}} \qquad \text{Substitution}$$

$$\frac{P\{\textbf{call}\ p(\mathbf{a}):(\mathbf{e})\}R}{\exists k'(P \;\&\; \forall \mathbf{a}(R \supset S))\{\textbf{call}\ p(\mathbf{a}):(\mathbf{e})\}S} \qquad \text{Adaptation}$$

$$\frac{P\{Q^x_y\}R}{P\{\textbf{new}\ x;\, Q\}R} \quad \text{(where } y \text{ is not in } Q \text{ unless } y \text{ and } x \text{ are the same)} \qquad \text{Declaration}$$

Explanation

P, P_1, P_2, R, S	stand for	propositional formulae
Q, Q_1, Q_2	stand for	program statements
x, y	stand for	variable names (y not free in P or R)
e	stands for	an expression
B	stands for	a Boolean expression
p	stands for	a procedure name
$\mathbf{x}$	stands for	a list of non-local variables of Q which are subject to change in Q
$\mathbf{v}$	stands for	a list of other non-local variables of Q
$\mathbf{a}$	stands for	a list of distinct variables
$\mathbf{e}$	stands for	a list of expressions, not containing any of the variables $\mathbf{a}$
k	stands for	a list of variables not free in $\mathbf{x}$, $\mathbf{v}$
k'	stands for	a list of variables not free in $\mathbf{a}$, $\mathbf{e}$, S
$P\{Q\}R$	stands for	if P is true of the program variables before executing the first statement of the program Q, and if Q terminates, then R will be true of the program variables after execution of Q is complete
S^x_e	stands for	the result of replacing all free occurrences of x in S by e. If e is not free for x in S, a preliminary systematic change of bound variables is assumed to be made
$\dfrac{P, R}{S}$	stands for	a rule of inference which states that if P and R have been proved, then S may be deduced
$\dfrac{R \quad P_1 \vdash P_2}{S}$	stands for	a rule of inference which permits deduction of S if R and P_2 are proved; however, it also permits P_1 to be assumed as a hypothesis in the proof of P_2. The deduction of P_2 from P_1 is known as a subsidiary deduction.

APPENDIX 1

Line No.	Proof			Justification
1	$A = A_0 \,\&\, m < n$	{**call** Part $(A, i, j):(m, n)$}	Partd (A, i, j, m, n) & Perm (A, A_0, m, n)	Already Proved
2	$\exists A_0(A = A_0 \,\&\, m < n \,\&\, \forall,A\, i,j$ (Partd (A, i, j, m, n) & Perm $(A, A_0, m, n) \supset L5))$	{**call** Part $(A, i, j):(m, n)$}	$L5$	Adaptation (1)
3	$A = A_0$	{**call** Quicksort $(A):(m, n)$}	Sorted (A, m, n) & Perm (A, A_0, m, n)	Hypothesis
4	$A = A_1$	{**call** Quicksort $(A):(m, j)$}	Sorted (A, m, j) & Perm (A, A_1, m, n)	Substitution (3)
5	$\exists A_1(A = A_1 \,\&\, \forall A($Sorted$(A, m, j)$ & Perm $(A, A_1, m, j) \supset L7))$	{**call** Quicksort $(A):(m, j)$}	$L7$	Adaptation (4)
6	$A = A_2$	{**call** Quicksort $(A):(i, n)$}	Sorted (A, i, n) & Perm (A, A_2, i, n)	Substitution (3)
7	$\exists A_2(A = A_2 \,\&\, \forall A($Sorted (A, i, n) & Perm $(A, A_2, i, n) \supset$ Sorted (A, m, n) & Perm $(A, A_0, m, n)))$	{**call** Quicksort $(A):(i, n)$}	Sorted (A, m, n) & Perm (A, A_0, m, n)	Adaptation (6)
8	$L2$(**call** Part $(A, i, j):(m, n)$;	**call** Quicksort $(A):(m, j)$; **call** Quicksort $(A):(i, n)$}	Sorted (A, m, n) & Perm (A, A_0, m, n)	Composition (2, 5, 7)
9	$L2${**new** i, j; **call** Part $(A, i, j):(m, n)$;	**call** Quicksort $(A):(m, j)$; **call** Quicksort $(A):(i, n)$}	Sorted (A, m, n) & Perm (A, A_0, m, n)	Declaration (8)
10	**if** $m < n$ **then** $L2$ **else** Sorted (A, m, n) & Perm $(A, A_0, m, n)\{Q\}$		Sorted (A, m, n) & Perm (A, A_0, m, n)	Condition (9)
11		$A = A_0 \supset L10$		Lemma
12	$A = A_0$	$\{Q\}$	Sorted (A, m, n) & Perm (A, A_0, m, n)	Consequence (11, 10)
13	$A = A_0$	{**call** Quicksort $(A):(m, n)$}	Sorted (A, m, n) & Perm (A, A_0, m, n)	Recursion (3-12)

The justification column gives the rule of inference and previous line(s) used in the derivation Q is an abbreviation for the body of Quicksort. $L2$ stands for the proposition on the left hand side of line 2, $L5$ for that on the left of line 5 and so on.

References

ABRAHAMS, P. W. (1963). Machine Verification of Mathematical Proof, Ph.D. thesis, Massachusetts Institute of Technology.

BURSTALL, R. M. (1970). Formal Description of Program Structure and Semantics in First Order Logic, *Machine Intelligence*, Vol. 5, University of Edinburgh.

FLOYD, R. W. (1967). Assigning Meanings to Programs, *Mathematical Aspects of Computer Science* (ed. J. T. Schwartz), American Mathematical Society.

GOOD, D. C. (1970). Towards a Man–Machine system for Proving Program Correctness, Ph.D. thesis, University of Wisconsin.

HOARE, C. A. R. (1961). Algorithm 64, *CACM*, Vol. 4, No. 7, p. 321.

HOARE, C. A. R. (1969). An Axiomatic Approach to Computer Programming, *CACM*, Vol. 12, No. 10, pp. 576-580.

HOARE, C. A. R. (1971a). Proof of a Program: Find, *CACM*, Vol. 13, No. 12.

HOARE, C. A. R. (1971b). Procedures and Parameters: An Axiomatic Approach, *Symposium on the Semantics of Algorithmic Languages* (ed. E. Engeler), Springer Verlag. Lecture Notes in Mathematics 188, 1971.

KING, J. C. (1969). A Program Verifier, Ph.D. thesis, Carnegie-Mellon University.

LONDON, R. L. (1970). Certification of Treesort, *CACM*, Vol. 13, No. 6, pp· 371-373.