Acta Informatica 1, 214–224 (1972)

Program Proving: Jumps and Functions

M. CLINT and C. A. R. HOARE

Received July 25, 1971

Summary. Proof methods adequate for a wide range of computer programs have been expounded in [1] and [2]. This paper develops a method suitable for programs containing functions, and a certain kind of jump. The method is illustrated by the proof of a useful and efficient program for table lookup by logarithmic search.

1. Introduction

There is a current well-justified view [3] that the use of jumps (**go to** statements) in programming is neither desirable nor necessary; and that the quality of a program tends to be inversely proportional to the number of jumps it contains. It is certainly true that the majority of programs are much more clearly expressed by use of regular structuring methods such as blocks, loops and procedure calls where necessary. These constructions reveal explicitly the underlying structure and normal flow of the algorithm at run time.

However, there are certain cases in which the program needs to break away from the normal flow, disrupting the regular hierarchical structure which has been built up. These cases are often those in which some abnormal condition has been detected which invalidates the assumptions on which the structure is based. Thus the only appropriate solution is immediately to break away from the current context by means of a jump. Since no return to this context in desired, the resulting breakdown of structure is deliberate and necessary.

If jumps are recognised as exceptional structure-breaking actions, then they fall into one of two classes, return-jumps and exit-jumps. A return-jump is one that occurs on recognition of the fact that the answer to the problem has already been found (or nearly found), and there is no need to unfold the nested loops, conditionals, procedures, and perhaps even recursions; all that is required (possibly after some adjustment) is to jump straight to the place where the answer is wanted. The exit jump is one that occurs on recognition of the fact that the problem posed is not soluble anyway; again there is no need to pass in an orderly fashion through the stacked **ends** of the enclosing loops, conditionals and procedures; what is required (possibly after some adjustment) is to jump to the end of the program; or at least jump to a context surrounding that in which the problem was posed, and which is prepared to take action in the event of its insolubility.

There is no doubt that the effect of return-jumps and exit-jumps can be obtained without using explicit jump instructions. However, this involves the introduction, assignment, and possibly quite frequent testing of Boolean markers;

and although the loss of efficiency may be slight, it is doubtful whether the program is any more perspicuous than one which contains an occasional explicit jump [4]. The objection to jumping is that it destroys the orderly structure of the algorithm; but in exceptional conditions, this expresses exactly the intention of programmer. The jump may be considered an ugly device, but it is thereby well suited to the occasional ugly situation in which it should be used.

If the jump is to be recognised as a useful programming tool, the problem arises of devising a notation for it that is highly convenient for its primary purpose of returning and exiting, but is sufficiently inconvenient to deter its use for expressing conditionals or loops, for which better notations are available. The solution was suggested by Landin [5], who regarded a label as a special kind of procedure. The "body" of the label is the sequence of statements which is executed immediately following that label—in ALGOL 60, the statements (roughly speaking) between the labelled statement and the end of the block to which the label is local. However, we prefer to "declare" the label (together with its body) at the head of the block, just like an ALGOL 60 procedure. Now the only difference between a procedure and a label is that on completion of a procedure body control returns to the place of call, but on completion of the label body, it always returns to the end of the block to which the label is local. A jump statement to a label is just like the call of a procedure, except that it is known that control will never come back to the place of the call.

However, the general proof methods described in this paper do not depend on the use of this non-standard notation for jumps; and Appendix III describes similar proof methods for more conventional jumps, provided that jumps *into* a compound or conditional statement from outside it are excluded.

Readers who have reached this point will probably be divided into two classes: those who ask "what's wrong with jumps anyway?" and those who are still unconvinced by the need for any kind of jump. To those in the first class there is little to say; to those in the second, it is possible to offer the consolation that programs which confine their jumping to genuine returns and exits do not present any great difficulty in the proof of their correctness.

Apart from jumps, this paper also gives proof rules for function declarations and calls. These rules apply only to the simplest type of a function—one which has no side-effects, and whose value does not depend on any global variables. The rule permits functions defined by procedures to feature in the normal way in mathematical and logical formulae, on the understanding that the functions are possibly only partial. The main interest in dealing with jumps and functions together is that although their proof rules are entirely independent, the proof methods turn out to apply satisfactorily even to functions which are exited by jumps.

2. Jumps

We introduce the following notation for the declaration of a label within a block

$$l \textbf{ label } Q_1; Q_2$$

where l is the identifier for the label, Q_1 is the body of the label, and Q_2 is the body of the block. As in the case of a procedure declaration, it is Q_2 that is executed on entry to the block; Q_1 is executed only if and when the label is jumped to.

As described above, a label is a procedure which shares its exit point with the block to which it is local. Thus the exit from this block may be made either after execution of Q_1 (the body of the label) or after execution of Q_2 (the body of the block). If R_1 is a condition which is always true on execution of Q_1, and R_2 is a condition which is always true after executing Q_2, then the condition which is true on exit from the block will be either one or the other, i.e., $R_1 \vee R_2$.

Now let P_1 be the precondition which must always be true before execution of the label body Q_1 if the desired result R_1 is to be true afterwards. If the block as a whole is to be correct, P_1 must be true before every jump to the label. Provided this is guaranteed, it does not matter what the programmer assumes to be true when control returns from the jump, since by definition of a jump such a return never takes place. It is even possible to assume that the uniformly false statement **false** is true after such a return. Using the notation of [1], this intuitive reasoning is embodied in the rule that

$$P_1\{\textbf{go to } l\}\textbf{false}$$

may be assumed as a hypothesis in the proof of the block Q_2 to which the label is local. Since **false** implies any proposition, the rule of consequence justifies replacement of **false** in this hypothesis by an arbitrary assertion.

In summary, proofs of programs containing jumps will use a rule of inference requiring a subsidiary deduction (based on hypothesis) as follows:

$$\frac{\begin{array}{c} P_1\{Q_1\}R_1 \\ P_1\{\textbf{go to } l\}\textbf{false} \vdash P\{Q_2\}R_2 \end{array}}{P\{l \textbf{ label } Q_1; Q_2\}R_1 \vee R_2}$$

A method of dealing axiomatically with more conventional methods of labelling is given in Appendix III.

3. Functions

A function is declared by the schema

$$f(\boldsymbol{x}) \textbf{ function } Q$$

where f is the function name, Q is the body of the function, and $\boldsymbol{x}$ is a list of formal parameters containing *all* of the free non-local variables appearing in Q (this stipulation can be relaxed under certain conditions not relevant to this paper). It is assumed that Q makes no assignment to any of its parameters so that there is no possibility of side effects. The function is invoked by writing $f(\boldsymbol{y})$ as part of an expression, where $\boldsymbol{y}$ is a list of expressions (or actual parameters) of the same length as $\boldsymbol{x}$. The effect of the invocation of a function is the execution of the body of the function with the actual parameters 'replacing' the formal parameters. As in ALGOL 60 and FORTRAN, the function name is used as a variable to store the result of the function to be returned to the place of the function call.

Suppose that some precondition P holds before commencing execution of Q, and that P is sufficient to establish the truth of a consequence R after execution of the function body. R will include the function name f, denoting the result of the function; and both P and R will include the formal parameters x of the function. It follows that the truth of P is sufficient to guarantee the truth of R with each occurrence of the function name f replaced by the function call $f(x)$. Furthermore, this is true of all possible values of x that satisfy P. Thus the rule for functions may be written as

$$\frac{P\{Q\}R}{\forall x\,(P \supset R^f_{f(x)})}.$$

4. The Null Statement

Normally, the body of a label includes actions which must be performed to make final adjustments before exiting from the associated block. However in some cases no further action is required at this point. This is expressed by writing a null statement as the body of the label. To make this more explicit, the null statement is denoted by the symbol

null;

thus a block with a null label would have the form

begin l **label null**; Q **end**.

The ALGOL equivalent of this form would be

begin Q; l: **end***.

In either case, exit from the block may be achieved by executing the statement **go to** l within Q.

The statement **null** is a dummy one and any condition P which holds before its execution is preserved after execution. Thus we obtain the obvious axiom schema:

$$P\{\textbf{null}\}P.$$

5. Description of Lookup

5.1. Criterion of Correctness

The function lookup is designed to perform a logarithmic search of a linear array A of length N. The elements in this array are assumed to be sorted in increasing order and no two elements have the same value. The purpose of the function is to discover which element of the array has a value equal to a specified number x. This number x must satisfy

$$A\,[1] \leqq x < A\,[N].$$

The result returned by the function is the array subscript value corresponding to the element which is equal to x, i.e. the value k such that $A\,[k] = x$. If such an

* The importance of this kind of exit jump is evidenced by the frequency in ALGOL 60 programs of labelled **ends** and by the incorporation in FORTRAN and PL/I of a special RETURN statement.

element does not exist then a branch to an error label is made. This causes an error flag to be set and a message to be printed.

Thus what we wish to prove is that

$$\forall A, N, x\,(1<N \;\&\; \textit{sorted}\,(A) \;\&\; A[1] \leqq x < A[N] \supset A[\textit{lookup}\,(A, N, x)] = x)$$

where sorted $(A) \equiv_{df} \forall i, j\,(1 \leqq i < j \leqq N \supset A[i] < A[j])$. However, it is obvious that lookup is a partial function, being undefined whenever

$$\neg\, \exists i (1 \leqq i \leqq N \;\&\; A[i] = x).$$

We therefore also wish to prove that a jump is made to the error label when (and only when) this condition holds. Thus this condition is chosen as the precondition for the label.

5.2. The Method of Lookup

Lookup operates on a sorted linear array A. None of the elements of the array are reassigned during execution of lookup so that A remains sorted throughout. The method used by lookup is a logarithmic scan. It consists in defining a sequence of progressively shorter nested segments of A, each of which is such that the values of its terminal elements provide bounds for the sought number x. The upper element of these pairs has value strictly greater than x, and the lower has value less than or equal to x. The endpoints of the segments are recorded by the variables m and n, where $m < n$. Thus the following relationship is invariant, and holds throughout the algorithm:

$$A[m] \leqq x < A[n].$$

The first segment in the sequence coincides with the full array A so that initially x must satisfy

$$A[1] \leqq x < A[N].$$

The length of the segment of interest is repeatedly reduced as follows. The value of the element of the array with subscript $(m+n) \div 2$ (this will be termed the central element) is compared with x. If this value is equal to x *lookup* is assigned the value of $(m+n) \div 2$ and exit from the function occurs immediately. If x has a greater value than the central element then the search may be confined to those elements of the array which have subscripts higher than this element and still lower than n. The lower endpoint of the target segment is therefore advanced by assigning the value of $(m+n) \div 2$ to m. If x is less than the value of the central element then the search may be confined to those elements of the array which have subscripts greater or equal to m and less than $(m+n) \div 2$. Accordingly the upper endpoint of the target segment is reset by assigning the value of $(m+n) \div 2$ to n. Having fixed the endpoints of the new subsegment the process is repeated.

This procedure is continued if necessary until the values of m and n are consecutive integers. When this point has been reached then either x is equal to $A[m]$ or it does not appear at all in the array A. In the first case *lookup* is assigned the value of m and exit from the function occurs. In the second case a jump is made to an error label E.

5.3. The Annotated Program

comment $\neg\exists i(1\leqq i<N \,\&\, A[i]=x)$; E **label** {*print* ('*number not there*'); *error* := **true**} **comment** *error* & *last* (*print*) = '*number not there*';
lookup (A, N, x) **function comment** $1<N$ & *sorted* (A) & $A[1]\leqq x<A[N]$;
begin comment $A[lookup]=x$;
out **label null**; **new** m, n;
$m:=1$; $n:=N$; **comment** $m<n$ & $A[m]\leqq x<A[n]$ & *sorted* (A) *is invariant*;
while $m+1<n$ **do**
begin new i;
$i:=(m+n)\div 2$;
if $x<A[i]$ **then** $n:=i$
else if $A[i]<x$ **then** $m:=i$
else {$lookup:=i$; **goto** *out*}
end;
if $A[m]\neq x$ **then goto** E
else $lookup:=m$; **comment** $A[lookup]=x$;
end;

5.4. Statement of Proof Requirements

The proof of the program given in § 5.3 falls into two parts, one part being a proof of the correctness of the label E and the other a proof of the correctness of the function lookup. For the label we require to prove

$$\neg\exists i\,(1\leqq i\leqq N \,\&\, A[i]\neq x)\{body_E\}error \,\&\, last(print)='number\ not\ there' \,\&\neg\exists i(1\leqq i\leqq N \,\&\, A[i]=x).$$

This is taken to be reasonably obvious. For the function we require to prove

$$1<N \,\&\, sorted(A) \,\&\, A[1]\leqq x<A[N]\{body_{lookup}\}\ A[lookup]=x$$

on the assumption that

$$\neg\exists i\,(1\leqq i<N \,\&\, A[i]=x)\{\textbf{go to } E\}\ \textbf{false}.$$

The proof of the function depends on the lemmas

Lemma 1. $m+1<n \supset$ **if** $x<A[i]$ **then** $m<i$ & *sorted* (A) & $A[m]\leqq x<A[i]$
else if $A[i]<x$ **then** $i<n$ & *sorted* (A) & $A[i]\leqq x<A[n]$
else $A[i]=x$
where $i=(m+n)\div 2$.

Lemma 2. $\neg m+1<n \supset$ **if** $A[m]\neq x$ **then** $\neg\exists i(1\leqq i\leqq N \,\&\, A[i]=x)$
else $A[m]=x$

given that *sorted* (A) & $A[m]\leqq x<A[n]$ & $m<n$.

5.5. *Proof of the Lemmas*

Proof of Lemma 1. The proof may be divided into three cases

Case 1, given that $x < A[(m+n) \div 2]$ it follows from the relation $A[m] \leqq x < A[n]$ that

$$A[m] \leqq x < A[(m+n) \div 2];$$

Case 2, given that $x > A[(m+n) \div 2]$ it follows from the same relation that

$$A[(m+n) \div 2] \leqq x < A[n];$$

Case 3, $\neg (x < A[(m+n) \div 2]) \,\&\, \neg (x > A[(m+n) \div 2]) \supset x = A[(m+n) \div 2]$.

Proof of Lemma 2. Obviously

$$m < n \,\&\, \neg m+1 < n \supset m = n-1.$$

If $A[m] \neq x$ then $A[n-1] < x < A[n]$. Since A is sorted, x is greater than all elements below the $(n-1)$th and less than all elements above the nth. It is therefore unequal to all elements of A. Otherwise $A[m] = x$, and the lemma is proved.

6. Formal Proof of Lookup

The purpose of this section is twofold; firstly to formalise the intuitive reasoning which convinces us that the lemmas proved in the previous section are those on which the correctness of the program depends; and secondly, to provide some grounds for belief that the formal rules of inference introduced in this paper are adequate to their purpose. The proof is given in full in Appendix I.

The complete set of axioms and inference rules used in the proof are given in Appendix II.

7. Conclusion

The precondition for the error label E in the proof above was selected in order to ensure that the function lookup fails to evaluate only in the case in which there is no answer. If a weaker precondition had been set (for example **true**) the proof of correctness of lookup would have been trivial but correspondingly useless. Thus the stated precondition for the jump must really be regarded as part of the criterion of correctness of lookup rather than as part of the correctness criterion of the label. This awkwardness shows that the $P\{Q\}R$ notation for expressing correctness is not wholly adequate and that some aspects of the correctness can be stated only by making assertions at some intermediate stage of the program i.e., just before the jumps.

The technique proposed here for declaring labels which deal with abnormal exits from functions might be developed to provide simple and efficient means of achieving the combined effect of conditions, ON-units and condition prefixes in PL/I. All that is necessary is to regard a condition as a standard label name (e.g., fpoverflow). If a label is declared with this name, and the corresponding error is detected in the block to which the label is local, a jump is made to the body of the label, and then to the end of the block. The proof methods associated with this facility are similar to those displayed in the case of lookup.

Of course, ON-conditions in the full PL/I language are more powerful than this, since they permit a return to the point at which the error was detected. However, the possibility that this was not the point at which the error actually occurred, together with some ugly problems of implementation and efficiency, suggest that the proposed treatment of error conditions as exit jumps may be a more successful language design decision, both for high quality implementation, and for user comprehension.

There may be some advantages to be gained by following a suggestion of Landin, permitting labels to have parameters in the same way as procedures. This would be certainly desirable for the suggested approach to traps and ON-conditions.

This work was carried out with the aid of a grant from the Science Research Council.

Appendix I

This Appendix contains the formal proof of lookup. The convention L_i and R_i (where i is an integer) is used to denote the propositions to the left-hand side and right-hand side of line i respectively. Q_i represents the fragment of the program text under consideration at line i.

Line No.	Formal proof of lookup	Justification
1.	$1<N$ & *sorted* (A) & $A[1]\leqq x<A[N]\{m:=1;\ n:=N\}m<n$ & *sorted* (A) & $A[m]\leqq x<A[n]$	Assignment and Composition
2.	$m+1<n$ & $R_1 \supset$ **if** $x<A[(m+n)\div 2]$ **then** $m<(m+n)\div 2$ & *sorted* (A) & $A[m]\leqq x<A[(m+n)\div 2]$ **else if** $A[(m+n)\div 2]<x$ **then** $(m+n)\div 2<n$ & *sorted* (A) & $A[(m+n)\div 2]\leqq x<A[n]$ **else** $A[(m+n)\div 2]=x$	Lemma 1
3.	$R_2\{i:=(m+n)\div 2\}$ **if** $x<A[i]$ **then** $A[m]\leqq x<A[i]$ & $m<i$ & *sorted* (A) **else if** $A[i]<x$ **then** $i<n$ & *sorted* (A) & $A[i]\leqq x<A[n]$ **else** $A[i]=x$	Assignment
4.	$m<i$ & *sorted* (A) & $A[m]\leqq x<A[i]\{n:=i\}R_1$	Assignment
5.	$i<n$ & *sorted* (A) & $A[i]\leqq x<A[n]\{m:=i\}R_1$	Assignment
6.	$A[i]=x\{lookup:=i\}A[lookup]=x$	Assignment
7.	$R_6\{$**go to** out$\}$ **false** $[\supset R_1]$	Hypothesis
8.	$R_3\{$**if** $x<A[i]$ **then** Q_4 **else if** $A[i]<x$ **then** Q_5 **else** $\{Q_6;Q_7\}\}R_1$	Composition (6, 7) & Alternation (4, 5)
9.	$R_2\{Q_3;Q_8\}R_1$	Composition (3, 8)
10.	$R_2\{$**new** $i;Q_9\}R_1$	Declaration (9)
11.	$L_2\{Q_{10}\}R_1$	Consequence (2, 10)

12. $R_1\{\textbf{while } m+1<n \textbf{ do } Q_{10}\}\neg\, m+1<n \;\&\; R_1$ Iteration (2, 11)

13. $R_{12} \supset \textbf{if } A[m] \neq x \textbf{ then } \neg\, \exists i(1 \leqq i \leqq N \;\&\; A[i]=x) \textbf{ else } A[m]=x$ Lemma 2

14. $\neg\, \exists i(1 \leqq i \leqq N \;\&\; A[i]=x)\{\textbf{go to } E\}\ \textbf{false} \quad [\supset A[\mathit{lookup}]=x]$ Hypothesis

15. $A[m]=x\{\mathit{lookup}:=m\}A[\mathit{lookup}]=x$ Assignment

16. $R_{13}\{\textbf{if } A[m] \neq x \textbf{ then } Q_{14} \textbf{ else } Q_{15}\}R_{15}$ Alternation (14, 15)

17. $R_{12}\{Q_{16}\}R_{15}$ Consequence (13, 16)

18. $L_1\{Q_1; Q_{12}; Q_{16}\}R_{15}$ Composition (5, 12, 17)

19. $L_1\{\textbf{new } m, n; Q_{18}\}R_{15}$ Declaration (18)

20. $R_{15}\{\textbf{null}\}R_{15}$ Vacuity

21. $L_1\{\text{out } \textbf{label null}; Q_{19}\}R_{15} \vee R_{15} \quad [\equiv R_{15}]$ Interruption (19, 20)

22. $L_{14}\{\mathit{print}(\text{'number not there'}); \mathit{error}:=\textbf{true}\}\ \mathit{error} \;\&\; \mathit{last}(\mathit{print}) = \text{'number not there'}$ Obvious

23. $\forall A, N, x\,(L_1 \supset A[\mathit{lookup}(A, N, x)]=x)$ Functionality (21)
Suppose that the definition of the function lookup is included in a program text Q and that a condition R holds on execution of Q. Then

24. $L_1\{E \textbf{ label } Q_{22}; Q\}R \vee R_{22}$ Interruption

---

Appendix II

P, P_1, P_2, R, R_1, S	stand for propositional formulae
Q, Q_1, Q_2	stand for program statements
x, y	stand for variable names (with y not free in P or R)
e	stands for an expression
B	stands for a Boolean expression
f	stands for a function name
$\boldsymbol{x}$	stands for a list of all free (non-local) variables of Q.

Formal rules

D_0 $\qquad R^x_e\{x:=e\}R$ Assignment

D_1 $\qquad \dfrac{P\{Q\}S \quad S \vdash R}{P\{Q\}R} \qquad \dfrac{P \vdash S \quad S\{Q\}R}{P\{Q\}R}$ Consequence

D_2 $\qquad \dfrac{P\{Q_1\}S \quad S\{Q_2\}R}{P\{Q_1; Q_2\}R}$ Composition

$$\frac{P\{Q\}S}{S\vdash \textbf{if } B \textbf{ then } P \textbf{ else } R}{S\{\textbf{while } B \textbf{ do } Q\}R} \qquad D_3 \quad \text{Iteration}$$

$$\frac{P_1\{Q_1\}R \qquad P_2\{Q_2\}R}{\textbf{if } B \textbf{ then } P_1 \textbf{ else } P_2\{\textbf{if } B \textbf{ then } Q_1 \textbf{ else } Q_2\}R} \qquad D_4 \quad \text{Alternation}$$

$$\frac{P\{Q^x_y\}R}{P\{\textbf{new } x; Q\}R} \qquad D_8 \quad \text{Declaration}$$

(where y is not in Q unless y and x are the same)

$$\frac{P_1\{Q_1\}R_1 \qquad P_1\{\textbf{go to } l\}\textbf{false} \vdash P\{Q_2\}R}{P\{l \textbf{ label } Q_1; Q_2\}R \vee R_1} \qquad D_9 \quad \text{Interruption}$$

$$\frac{f(x) \textbf{ function } Q \qquad P\{Q\}R}{\forall x(P \supset R^f_{f(x)})} \qquad D_{10} \quad \text{Functionality}$$

$$R\{\textbf{null}\}R \qquad \text{Vacuity}$$

Appendix III

This appendix gives an axiomatic treatment of a more conventional method of setting labels, by prefixing them to the statement to which they refer, e.g.:

$$Q_1; l: Q_2$$

where Q_1 and Q_2 are simple or compound statements constituting the compound tail of the block to which label l is local. For simplicity, we will assume that l is the only local label of the block. Now if S is the desired precondition of each jump to l, S must also be true on termination of Q_1. Since jumps to l may occur in either Q_1 or Q_2, we shall require to use the relevant hypothesis in both halves. In other respects the rule given below is very similar to that for an ordinary compound statement

$$\frac{S\{\textbf{go to } l\}\textbf{false} \vdash P\{Q_1\}S \qquad S\{\textbf{go to } l\}\textbf{false} \vdash S\{Q_2\}R}{P\{Q_1; l: Q_2\}R}$$

This rule becomes very much more complicated when there is more than one label in the block; though the extra complexity would be avoided if *backward* jumps were disallowed. But even worse complications follow if the programmer is permitted to jump *into* a structure, such as a conditional statement. These complications were first discovered in [5], and it is gratifying to note that some recent programming languages have disallowed such jumps.

References

1. Hoare, C. A. R.: An axiomatic basis for computer programming. Comm. ACM **12**, No. 10, 576–580 (October 1969).
2. — Procedures and parameters; an axiomatic approach, Symposium on the Semantics of Algorithmic Languages (ed. E. Engeler). Berlin-Heidelberg-New York: Springer 1971.
3. Dijkstra, E. W.: Go to statement considered harmful. Letter to the editor. Comm. ACM **11**, No. 3, 147–148 (March 1968).
4. Knuth, D. E., Floyd, R. W.: Notes on avoiding "go to" statements. Technical Report No. CS 148, Computer Science Dept., Stanford, Jan. 1970.
5. Landin, P. J.: A correspondence between ALGOL 60 and Church's lambda notation, parts I and II. Comm. ACM **8**, Nos. 2 and 3, 89–101, Feb., 158–165, Mar. (1965).

Dr. M. Clint
Department of Mathematics
New University of Ulster
Coleraine, Co. Londonderry
Northern Ireland

Prof. C. A. R. Hoare
Department of Computer Science
The Queen's University of Belfast
Belfast BT 7 1 NN
Northern Ireland