BIT 6 (1966), 310–316

PROOF OF ALGORITHMS BY GENERAL SNAPSHOTS

PETER NAUR

Abstract.

A constructive approach to the question of proofs of algorithms is to consider proofs that an object resulting from the execution of an algorithm possesses certain static characteristics. It is shown by an elementary example how this possibility may be used to prove the correctness of an algorithm written in ALGOL 60. The stepping stone of the approach is what is called General Snapshots, i.e. expressions of static conditions existing whenever the execution of the algorithm reaches particular points. General Snapshots are further shown to be useful for constructing algorithms.

Key words: Algorithm, proof, computer, programming.

Introduction.

It is a deplorable consequence of the lack of influence of mathematical thinking on the way in which computer programming is currently being pursued, that the regular use of systematic proof procedures, or even the realization that such proof procedures exist, is unknown to the large majority of programmers. Undoubtedly, this fact accounts for at least a large share of the unreliability and the attendant lack of over-all effectiveness of programs as they are used to-day.

Historically this state of affairs is easily explained. Large scale computer programming started so recently that all of its practitioners are, in fact, amateurs. At the same time the modern computers are so effective that they offer advantages in use even when their powers are largely wasted. The stress has been on always larger, and, allegedly, more powerful systems, in spite of the fact that the available programmer competence often is unable to cope with their complexities.

However, a reaction is bound to come. We cannot indefinitely continue to build on sand. When this is realized there will be an increased interest in the less glamorous, but more solid, basic principles. This will go in parallel with the introduction of these principles in the elementary school curricula. One subject which will then come up for attention is that of proving the correctness of algorithms. The purpose of the present article is to show in an elementary way that this subject not only exists, but is ripe to be used in practise. The illustrations are phrased in ALGOL 60, but the technique may be used with any programming language.

The meaning of proof.

The concept of proof originated in mathematics where we prove that a theorem, i.e. some statement about the properties of certain concepts, is logically contained in, or follows from, certain given axioms. A proof always involves two things, e.g. a theorem and a set of axioms, which in consequence of the proof are shown to be consistent.

In data processing we need proofs to relate the transformation defined by an algorithm to a description of the transformation in some other terms, usually a description of the static properties of the result of the transformation. The need for a proof arises because we want to convince ourselves that the algorithm we have written is correct. Ideally our way of proceeding is as follows. We first have the description of the desired result in terms of static properties. We then proceed to construct an algorithm for calculating that result, using examples and intuition to guide us. Having constructed the algorithm we want to prove that it does indeed produce a result having the desired properties.

As a very simple example, let the problem be to find the greatest of N given numbers. The numbers may be given as an **array** $A[1:N]$. The result, R, may be described statically as being the value of such a member of the array A, $A[r]$ say, that it holds for any i between 1 and N that $A[i] \leqq A[r]$. Note that this description is static. It describes the desired result as being greater than or equal to any other element of the array, not by way of a process showing how to find it. Note also that there is no guarantee implied that such a result exists at all. Proving in general that objects having certain properties exist is an important mathematical pursuit. One way of expressing such a proof is simply to provide an algorithm which finds the object. For our purposes this is often fully sufficient, so normally we do not have to worry about any other existence proof.

We now enter into the construction of an algorithm for finding the greatest number. We may be more or less clever in doing this. A good idea, which may or may not occur to us, is to run through the numbers once, keeping at all stages track of the greatest number found so far. We thus arrive at something like the following algorithm, which incidentally still can be made more effective:

PROGRAM 1

Greatest number

```
r := 1;
for i := 2 step 1 until N do
  if A[i] > A[r] then r := i;
R := A[r];
```

This looks pretty good to me. However, can I prove that it is correct? Many people, including many experienced programmers, will tend to dismiss the question, claiming that it is obvious that the solution is correct. I wish to oppose this view. Although the algorithm is not very complicated, it does contain some 7 operators and 12 operands. This is more than what can be grasped immediately by anyone, and I am sure that even the experienced programmer, in studying the algorithm, makes use of mental images and a certain decomposition of the process before he accepts it. I wish to claim that he goes through a proof, and I want this to be brought into the open, using a technique which can be used also in more complicated cases.

Snapshots.

Our proof problem is one of relating a static description of a result to a dynamic description of a way to obtain the result. Basically there are two ways of bringing the two descriptions closer together, either we may try to make the static description more dynamic, with a hope of getting to the given algorithm, or we may try to make the dynamic description more static. Of these the second is clearly preferable because we have far more experience in manipulating static descriptions, through practise in dealing with mathematical formulae. Therefore, if only we can derive some static description from the dynamic one, there is good hope that we may manipulate it so as to show that it is identical with the given static description.

	r	i	R	N	$A[1]$	$A[2]$	$A[3]$	$A[4]$
Initial				4	2	1	5	2
Following $r := 1$	1							
– **for** $i := 2$		2						
Second time in loop		3						
Following $r := i$	3							
Third time in loop		4						
Following $R := A[r]$			5					

We are thus led to the use of snapshots of the dynamic process for purposes of proof, because a snapshot is an instantaneous, and therefore static, picture of the development of the process. In its most primitive form a snapshot refers to one particular moment in the development of the process applied to one particular set of data. In a slightly more developed use we give a series of snapshots, referring to successive mo-

ments in the process, but still as applied to one particular set of data. To illustrate this technique, every detail of an example of the use of Program 1 is given above. Successive snapshots are given in successive lines, where for clarity a value given in a column holds unchanged in following lines unless another value is given.

This snapshot technique is quite useful as an aid to understanding a given algorithm. However, it is not a proof technique because it depends entirely on the choice of the data set. In order to achieve a proof we shall need more general snapshots.

General Snapshots.

By a General Snapshot I shall mean a snapshot of a dynamic process which is associated with one particular point in the actual program text, and which is valid every time that point is reached in the execution of the process.

From this definition it immediately follows that the values of variables given in a General Snapshot normally at best can be expressed as general, mathematical expressions or by equivalent formulations. I have to say "at best" because in many cases we can only give certain limits on the value, and I have to admit "equivalent formulations" because we do not always have suitable mathematical notation available.

In order to illustrate this notion, here is a version of the above algorithm expanded with General Snapshots at six different points:

PROGRAM 2

Greatest number, with snapshots

comment *General Snapshot* 1: $1 \leqq N$;
$r := 1$;
comment *General Snapshot* 2: $1 \leqq N$, $r = 1$;
for $i := 2$ **step** 1 **until** N **do**
  begin comment *General Snapshot* 3: $2 \leqq i \leqq N$, $1 \leqq r \leqq i-1$, $A[r]$ *is the greatest among the elements* $A[1], A[2], \ldots, A[i-1]$;
  if $A[i] > A[r]$ **then** $r := i$;
  comment *General Snapshot* 4: $2 \leqq i \leqq N$, $1 \leqq r \leqq i$, $A[r]$ *is the greatest among the elements* $A[1], A[2], \ldots, A[i]$;
  end;
comment *General Snapshot* 5: $1 \leqq r \leqq N$, $A[r]$ *is the greatest among the elements* $A[1], A[2], \ldots, A[N]$;
$R := A[r]$;
comment *General Snapshot* 6: R *is the greatest value of any element,* $A[1], A[2], \ldots, A[N]$;

In the first instance the reader may feel that these snapshots are as unsystematic as the original algorithm. The reason for this is that I have only given the final result, not the steps leading to it. The best way to see how they have arisen is to start with number 3. At this stage we are primarily interested in the values of i and r. The condition on i, $2 \leqq i \leqq N$, seems a reasonable first guess in view of the preceding for clause. The guess on r, $1 \leqq r \leqq i-1$, expresses that we have so far scanned the elements up to $A[i-1]$, and the explanation of $A[r]$ hopefully will seem a sensible claim. With snapshot 3 as our initial hypothesis we can now execute one step of the algorithm, to snapshot 4. The step is conditional, so we have to consider the two cases separately.

First, suppose that $A[i] > A[r]$ is true. Then clearly $A[i]$ is the greatest among the elements up to $A[i]$. Changing r to i as done in the assignment then makes it again true to say that $A[r]$ is the greatest. In this case the previous condition on r becomes false and has to be changed to $1 \leqq r \leqq i$.

Second, suppose that $A[i] > A[r]$ is false. Then $A[r]$ remains the greatest element so far found. The extended range for r is also valid. Altogether we have seen that if snapshot 3 is correct, then snapshot 4 follows.

Continuing now from snapshot 4 we must be quite clear on the meaning of the for clause governing the execution, from which it follows that the situations described in snapshot 4 must be divided into two cases: (1) $2 \leqq i \leqq N-1$, leading to stepping on of i and a repetition at snapshot 3, and (2) $i = N$, continuing the execution beyond the following **end**. In case (1) the stepping on of i leads to a situation which may be derived from snapshot 4, case (1), by replacing all occurrences of i by $i-1$. The point is that what is now denoted i is unity larger than that value of i for which the snapshot is valid. By this substitution we arrive at a situation in which we have $2 \leqq i-1 \leqq N-1$, or equivalently $3 \leqq i \leqq N$, and otherwise precisely the situation of snapshot 3. We have thus proved that the hypothetical snapshot 3 is consistent with the action taking place in the for loop. To complete the proof of the correctness of snapshot 3 we still have to check the first entry into the loop, i.e. the transition from snapshot 2 to 3. Again remembering the definition of the for clause, we only have to consider the subcases of snapshot 2 for which $2 \leqq N$. These subcases clearly lead to a situation consistent with snapshot 3 and moreover show that the cases $2 = i$ and $1 = r$ are realized, a fact which does not follow from the transition from snapshot 4.

The situation of snapshot 5 can be entered either from snapshot 2 in case $1 = N$, or from snapshot 4 in case $i = N$ with $2 \leqq N$. Both of these entries combine to form the situation described in snapshot 5, which is thereby proved. It should be carefully noted that snapshot 5 requires

$1 \leqq N$ to make sense. This condition is therefore necessary both in snapshots 2 and 1. Appearing before the complete algorithm it becomes a condition on the proper use of it. This is emphasized in the final transition from snapshot 5 to 6 where the variable $A[r]$ occurs. If the array is declared as **array** $A[1:N]$, this variable is defined only if $1 \leqq N$.

The proof of snapshot 6 establishes the correctness of the algorithm.

Exhaustive enumeration of entry and exit conditions.

One of the most common mistakes in programming is that special cases, needing special treatment in the program, are overlooked. The General Snapshots offer obvious help in this matter. As an illustration, consider the situation of General Snapshot 4. The situation is entered only by one route, and we have

Entry condition: $2 \leqq i \leqq N$

However, the effect of the for clause is to divide the further process into two branches:

Exist condition: (1) $2 \leqq i \leqq N-1$, (2) $i = N$

To check that no cases have been overlooked we have to see that the union of the exist conditions exhaust the entry condition.

More generally a situation may be entered by many different routes. General Snapshot 3 shows a situation having two entries. Similarly a situation may continue along many different routes. To check for the completeness we must in any case establish an *exhaustive enumeration* of the entry conditions and see that it is matched perfectly in the exit.

External parameters of a process.

When describing a General Snapshot it is important to realize that all variables have their current values, which are generally not the same from one snapshot to another. This may throw doubt on the real significance of a result derived by means of a proof. As an example, we may be concerned whether in General Snapshot 6 above the values of the elements $A[1], A[2], \ldots, A[N]$ are in fact the ones provided when the algorithm was entered. In order to remove such doubts we may add further information to the snapshots, relating the values of variables in different snapshots to one another. Thus we may add in each of the snapshots 2 to 6 in Program 3.2 the claim:

comment *The values of* N, $A[1], A[2], \ldots, A[N]$, *are the same as when General Snapshot* 1 *was last encountered*;

A statement of this kind must be proved along with all other claims made in a General Snapshot.

This technique particularly helps in describing parameters of subprocesses designed to be joined together to more complicated programs.

Construction of algorithms.

The principal difficulty in the construction of algorithms lies with the loops. A process written as part of a loop must so to speak bite its own tail properly. At the same time most of the execution time of normal programs is spent in a loop, often the innermost one of a nest of loops. When constructing algorithms we should therefore start by establishing the loop structure and then attack each nest of loops by starting with the innermost one. While doing this we must decide on the most convenient representation of the data processed.

When writing the actual statements of loops, General Snapshots may be used as aids to construction, as follows: Consider the method of algorithmic solution which you intend to use. Imagine that you are right in the middle of using this method, in other words that the algorithm is going, but that nothing has been completed yet. Now describe the situation, in terms of a General Snapshot. Introduce any variables you need to make this description specific. Write down the limits on your variables and other descriptions of their current values. When you have done this, write down the essential, central part of your process and convince yourself that the picture you have made is in fact adequate as a basis for doing the central process. Then write the proper preparation of the process. In particular, for every single variable used make sure that it is (1) declared, (2) initialized, (3) changed, and (4) used.

BIBLIOGRAPHY

The basic question of proof has so far been ignored in data processing to an incredible degree. A review of recent work is given in:

J. McCarthy, *Problems in the Theory of Computation*, Proc. IFIP Congress 65, Vol. 1, 1965, pp. 219–222.

Most of the work described there is heavily oriented towards the basic theoretical problems of computation. The General Snapshots of the present chapter are related to the state vectors of J. McCarthy:

J. McCarthy, *A Formal Description of a Subset of ALGOL*, Proc. of a Conference on Formal Language Description Languages, Vienna, 1964.

However, the present approach, which is directly applicable in proving and constructing practical programs, is believed to be new. Similar concepts have been developed independently by Robert W. Floyd (unpublished paper, communicated privately).

A/S REGNECENTRALEN, COPENHAGEN, DENMARK