

# Mathematical science of computation and structured programming

## Cultures of Programming — Part 4

**Tomas Petricek**, Charles University, Prague

✉ [tomas@tomasp.net](mailto:tomas@tomasp.net)

🌐 <https://tomasp.net>

🦋 [@tomasp.net](#)



# *Mathematical culture*

*Proving program properties*

# Mathematical science of computation

Programs as formal entities  
Algorithms as objects to study  
Formal mathematical reasoning

## Algol paradigm

### 4.2. ASSIGNMENT STATEMENTS

#### 4.2.1. Syntax

(left part) ::= (variable) := /(procedure identifier) :=  
(left part list) ::= (left part)|(left part list)(left part)  
(assignment statement) ::= (left part list)(arithmetic expression)|  
(left part list)(Boolean expression)

#### 4.2.2. Examples

$s := p[0] := n := n+1+s$   
 $n := n+1$   
 $A := B/C - v - q \times S$   
 $S[v, k+2] := 3 - \arctan(s \times \text{zeta})$   
 $V := Q > Y \wedge Z$

#### 4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

# Algol paradigm

Mathematical Science (McCarthy, 1962)

- = Two procedures (functions) are equivalent
- ◻ Procedure terminates for all input data
- ⌘ Translation correctly translates procedures
- ⚙ Translation increases efficiency

# McCarthy's factorial proof

$$g(n, s) = \text{if } n = 0 \text{ then } s \text{ else } g(n - 1, n \times s)$$

$$n! \times s = n! \times s \quad (1)$$

$$n! \times s = \text{if } n = 0 \text{ then } n! \times s \text{ else } n! \times s \quad (2)$$

$$n! \times s = \text{if } n = 0 \text{ then } 1 \times s \text{ else } ((n - 1)! \times n) \times s \quad (3)$$

$$n! \times s = \text{if } n = 0 \text{ then } s \text{ else } (n - 1)! \times (n \times s) \quad (4)$$

$$g(n, s) = \text{if } n = 0 \text{ then } s \text{ else } g(n - 1, n \times s) \quad (5)$$

# Algol paradigm

Proving programs correct

- ⚙ **Peter Naur (1966)**

Proofs of algorithms by general snapshots

- ⚙ **Robert Floyd (1967)**

Assigning meaning to programs

- ⚙ **Tony Hoare (1969)**

An axiomatic basis for computer programming

# Floyd (1967)

Cites McCarthy and  
uses a subset of Algol

But uses flowcharts!

Common visual  
representation of  
programs at the time

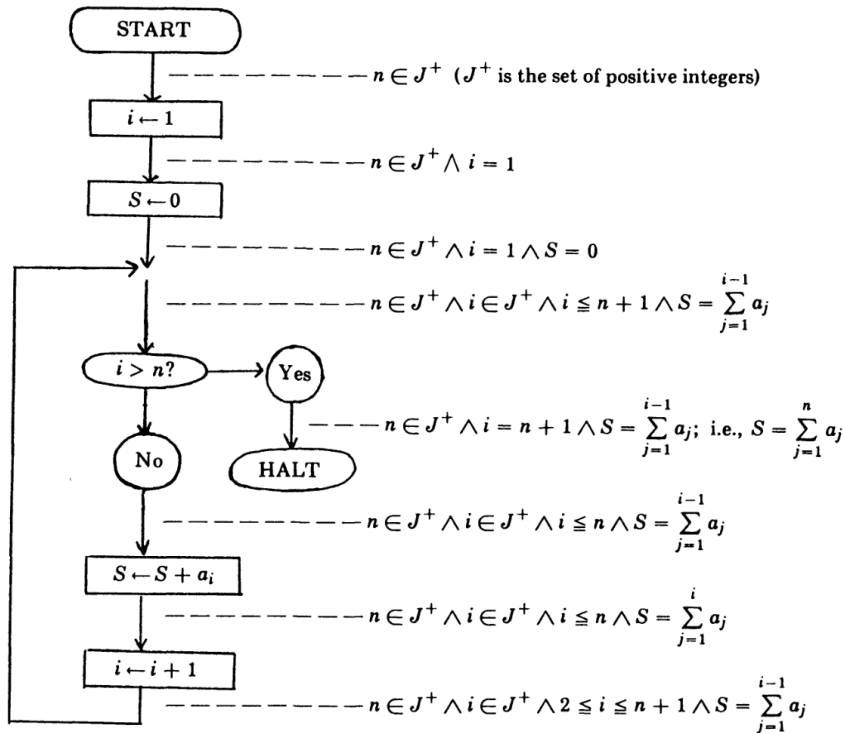


Fig. 1. Flowchart of program to compute  $S = \sum_{j=1}^n a_j$  ( $n \geq 0$ ).

```
s = 1; i = 1;

while i < n do

    i = i + 1;
    s = s * i;

end
```

# Hoare (1969)

Annotate programs using  
triples  $\{P\} c \{Q\}$  where:

- $P$  pre-condition holds before
- $c$  command to run
- $Q$  post-condition holds afterwards

Using modern notation for  
(now called) Hoare-triples

```

{ }
s = 1; i = 1;
{ s = i! }
while i < n do
    { s = i! }
    i = i + 1;
    s = s * i;
    { s' = i'! }
end
{ s = i! ∧ i = n }

```

# Hoare (1969)

Reasoning about programs  
using Hoare triples

## Composition rule!

If  $\{P\} c_1 \{Q\}$   
and  $\{Q\} c_2 \{R\}$   
then  $\{P\} c_1; c_2 \{R\}$

# Proofs of programs in practice

## Proof of a Program: FIND

C. A. R. HOARE  
Queen's University,\* Belfast, Ireland

A proof is given of the correctness of the algorithm "Find." First, an informal description is given of the purpose of the program and the method used. A systematic technique is described for constructing the program proof during the process of coding it, in such a way as to prevent the intrusion of logical errors. The proof of termination is treated as a separate exercise. Finally, some conclusions relating to general programming methodology are drawn.

KEY WORDS AND PHRASES: proofs of programs, programming methodology, program documentation, program correctness, theory of programming  
CR CATEGORIES: 4.0, 4.22, 5.21, 5.23, 5.24

### 1. Introduction

In a number of papers [1, 2, 3] the desirability of proving the correctness of programs has been suggested and this has been illustrated by proofs of simple example programs. In this paper the construction of the proof of a useful, efficient, and nontrivial program, using a method based on invariants, is shown. It is suggested that if a proof is constructed as part of the coding process for an algorithm, it is hardly more laborious than the traditional practice of program testing.

### 2. The Program "Find"

The purpose of the program Find [4] is to find that element of an array  $A[1:N]$  whose value is  $f$ th in order of magnitude; and to rearrange the array in such a way that this element is placed in  $A[f]$ ; and furthermore, all elements with subscripts lower than  $f$  have lesser values, and all elements with subscripts greater than  $f$  have greater values. Thus on completion of the program, the following relationship will hold:

$A[1], A[2], \dots, A[f-1] \leq A[f] \leq A[f+1], \dots, A[N]$

This relation is abbreviated as Found.

One method of achieving the desired effect would be to

sort the whole array. If the array is small, this would be a good method; but if the array is large, the time taken to sort it will also be large. The Find program is designed to take advantage of the weaker requirements to save much of the time which would be involved in a full sort.

The usefulness of the Find program arises from its application to the problem of finding the median or other quantiles of a set of observations stored in a computer array. For example, if  $N$  is odd and  $f$  is set to  $(N+1)/2$ , the effect of the Find program will be to place an observation with value equal to the median in  $A[f]$ . Similarly the first quartile may be found by setting  $f$  to  $(N+1)/4$ , and so on.

The method used is based on the principle that the desired effect of Find is to move lower valued elements of the array to one end—the "left-hand" end—and higher valued elements of the array to the other end—the "right-hand" end. (See Table I(a)). This suggests that the array be scanned, starting at the left-hand end and moving rightward. Any element encountered which is small will remain where it is, but any element which is large should be moved up to the right-hand end of the array, in exchange for a small one. In order to find such a small element, a separate scan is made, starting at the right-hand end and moving leftward. In this scan, any large element encountered remains where it is; the first small element encountered is moved down to the left-hand end in exchange for the large element already encountered in the rightward scan. Then both scans can be resumed until the next exchange is necessary. The process is repeated until the scans meet somewhere in the middle of the array. It is then known that all elements to the left of this meeting point will be small, and all elements to the right will be large. When this condition holds, we will say that the array is *split* at the given point into two parts (see Table I(b)).

The reasoning of the previous paragraph assumes that there is some means of distinguishing small elements from large ones. Since we are interested only in their comparative values, it is sufficient to select the value of some arbitrary element before either of the scans starts; any element with lower value than the selected element is counted as small, and any element with higher value is counted as large. The fact that this discriminating value is arbitrary means that the place where the two scans will meet is also arbitrary; but it does not affect the fact that the array will be split at the meeting point, wherever that may be.

Now consider the question on which side of the split the  $f$ th element in order of value is to be found. If the split is to the right of  $A[f]$ , then the desired element must of necessity be to the left of the split, and all elements to the right of the split will be greater than it. In this case, all elements to the right of the split can be ignored in any future processing, since they are already in their proper

\* Department of Computer Science

## Operating Systems

B. RANDALL, Editor

### Prevention of System Deadlocks

A. N. HABERMANN  
Carnegie-Mellon University,\* Pittsburgh, Pennsylvania

A well-known problem in the design of operating systems is the selection of a resource allocation policy that will prevent deadlock. Deadlock is the situation in which resources have been allocated to various tasks in such a way that none of the tasks can continue. The various published solutions have been somewhat restrictive: either they do not handle the problem in sufficient generality or they suggest policies which will on occasion refuse a request which could have been safely granted. Algorithms are presented which examine a request in the light of the current allocation of resources and determine whether or not the granting of the request will introduce the possibility of a deadlock. Proofs given in the appendixes show that the conditions imposed by the algorithms are both necessary and sufficient to prevent deadlock. The algorithms have been successfully used in the THE system.

KEY WORDS AND PHRASES: multiprogramming, time-sharing, scheduling, resource allocation  
CR CATEGORIES: 3.72, 4.32, 6.20

### Introduction

Consider a set of sequential processes which share system resources in such a way that (1) while a resource  $A$  is allocated to a process  $P_1$  no other process  $P_2$  can seize  $A$ , and (2) an allocated resource is not released until it has fulfilled its task.

Dijkstra shows in [2] and [6] how the use of system resources can be made exclusive by means of the primitives  $P$ -operation and  $V$ -operation. Dennis and Van Horn [3] use the macro instructions  $LOCK$  and  $UNLK$ , specially designed for this purpose. These techniques have in common that a process  $P_1$ , performing a  $P$ -operation (e.g.  $LOCK$  instruction) will get the resource requested if it is free and, if not, process  $P_1$  is held up until a matching  $V$ -operation (e.g.  $UNLK$  instruction) is performed. The techniques also have in common the fact that they do not imply any rules for resource allocation. This is still a matter of system design.

If the total request for resources of a certain type can

\* Computation Center

exceed the amount available, some strategy is needed to order the assignment of resources in time. Care must be taken that the strategy applied cannot cause a deadlock: i.e. the situation that cooperating processes prevent their mutual progress even though no single one requests more resources than are available.

Suppose, for instance, each of two processes  $P_1$  and  $P_2$  needs the resources  $A$  and  $B$  at the same time, i.e. when either  $A$  or  $B$  has been allocated to it.  $P_1$  will certainly not release the allocated resource before the other one has also been allocated to it (likewise  $P_2$ ). Suppose the strategy is such that allocation will follow on request if the resource is free. Then the situation may arise that  $P_1$  has seized resource  $A$  and  $P_2$  resource  $B$ . From then on,  $P_1$  and  $P_2$  will wait for each other indefinitely, since  $P_1$  certainly does not release  $A$  until it has seized  $B$ , i.e. not until  $P_2$  has released  $B$ , while  $P_2$  will not release  $B$  until it has seized  $A$ .

One way out of the difficulty is to permit a set of processes to run concurrently only if together they do not claim more resources than are available [4]. However, we will see that such a strategy is unnecessarily restrictive.

In this paper algorithms are described by means of which a deadlock can be prevented, and yet as many resource requests are granted as can be allowed, while neither are resources reserved on account of a claim, nor is resource request restricted to a few processes. The algorithms decide only whether or not granting a request can produce a deadlock, so that assignment rules (according to priority rules, for instance) can be implemented freely.

In the author's thesis [5] different sorts of resource requests are discussed, in particular, the case of temporary lease, i.e. when a process asks for a resource and can promise its release unconditionally, as well as the case of a single directed request, i.e. when a process can continue independently of resource activity. Though [5] also contains the theorems and proofs, albeit in a slightly different form, the aim of the present paper is to show the usefulness and applicability of the algorithms which prevent a deadlock.

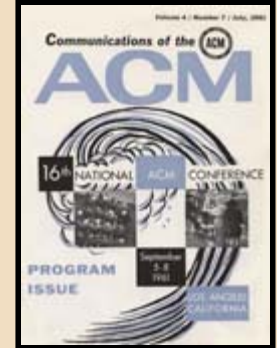
### Another Form of Deadlock

When resource activity itself may imply resource request, another form of deadlock can occur. Let, for instance, a disk device be controlled by a sequential process  $D$  that runs synchronously with the device interrupts.  $D$  is to be considered a resource, to which tasks can be

# Proofs of programs in practice

## Prevention of deadlocks (1969)

- Abstract algorithm written as pseudocode
- Model using states and transitions
- Not actual source code!



## Proof of FIND (1971)

- Part of Hoare's QuickSort algorithm
- Find  $f^{th}$  element in an array and rearrange
- $A[1], \dots, A[f - 1] \leq A[f] \leq A[f + 1], \dots, A[N]$
- ... it may be possible to enlist the aid of a computer