IBM

IBM United Kingdom
Laboratories Limited

CJ
FILE

Formal Development of Programs

C.B. Jones

June 1973

215-8061-0

Technical Report TR.12.117

Unrestricted

Formal Development of Programs

C. B. Jones
Programming Technology Group
Product Test Laboratory
IBM United Kingdom Laboratories Limited
Hursley
Winchester

12 April 1973

ABSTRACT

An approach to the construction of correct programs is proposed. The idea of developing an algorithm through stages of abstractness is backed by showing how proofs can be used to justify each stage of development. The gains of using formal notation are emphasised and illustrated by a number of examples.

CONTENTS

1. INTRODUCTION

The possibility of there being an error in a program, which is written and tested in the normal way, must be known to computer professional and newspaper reader alike. It has got to the stage where the problem of measuring the "reliability" - defined as the chance of (not) encountering an error - is receiving much attention. The use of the term reliability tends to hide the fundamental failure. Programs are not systems which decay or fail "if the wind is in the wrong direction". Their outcome is deducible. If a program fails on some input data today it will also fail on the same data tomorrow (ignoring the problems of asynchronous interaction, which makes the situation worse not better).

It can be argued that part of the reason for this state of affairs is the success of computing. Early success is one of the reasons that the tasks which programmers have been invited to tackle have expanded in size and, perhaps more important, in degree of dependence on other systems far faster than the methods governing program construction. It is the aim of this paper to make a contribution to these methods.

The author rules out any chance of achieving drastic improvements in "reliability" by working on testing techniques. The arguments for the limitations of testing are eloquently given in refs 9 and 11. A few moments consideration of the number of test cases required to "fully test" all paths of a really large program should soon bring the reader to the point of seeking an alternative to testing. The most promising alternatives would appear to be those based on the comment about the deducibility of the outcome of a program. The idea is to reason about all possible cases in the way mathematicians do when presenting proofs of theorems.

In order to prove that a program contains no errors, it is clearly necessary to have a precise statement of what its (correct) results should be. It will be argued below that only by using some (formal) notation is it possible to make this "specification" precise enough. Given a formal specification, much work (based largely on ref 8) has been done on proving that a program fulfils its specification. Such a proof establishes that for all permissible inputs, the output of the program will be in some specified relation to the input.

An attempt to write a program and then construct a proof of its correctness has the obvious shortcoming that such a proof will only be possible if the program is correct. The problem of obtaining a correct program is still unsolved. One could ask

whether attempting to construct a proof is an efficient way of removing "bugs" from a program. This is not so for three reasons. Firstly, the level of detail required to construct the proof of a finished program is too low. Secondly, there is the difficulty of choosing between two possible reasons for failing to prove a result: either the falsity of the result or a lack of ingenuity. Lastly there is a danger that any analysis made to help write the original program will not be effectively used in the proof construction.

Consideration of the problems caused by the combination of size and level of detail suggests that the way to handle the problem is to break it up in a suitable way. In order to avoid the details it is necessary to use abstract notions of the parts into which the problem is broken. The simplest example of such abstraction is to state only the required specification of the component and to delay development of its program to a later stage. This is discussed in section 5. The more subtle abstraction of the data on which a component works is discussed in section 7.

A proof of correctness of a program can now be constructed by proving each stage of development correct. Such a proof will, of course, require specifications of what can be assumed of sub-components. One is again forced to have a notation for recording such specifications. If used throughout the design of a program a significant bonus will be the recorded design history. Considering the proof the requirement for notation will be not only a question of preciseness but also of conciseness. Notice that whilst recording the proof is a new task, an argument for correctness should already be in the programmers mind. Having broken the development into levels it should be possible to convey this argument clearly by using ideas of the program.

This whole process - breaking up the development into stages or levels of abstraction, formally specifying and proving - is referred to as "Formal Development". It has much in common with "Structured Programming" (ref 9), "Stepwise Refinement" (ref 10) and ref 7.

Any comparison of the effort of Formally Developing programs with the more conventional methods of construction is difficult. It is clear that the time taken to Formally Develop a program is likely to be longer than the time to write code without documenting or justifying the stages of development. But the production of such code is unlikely to be the total investment: "testing" and "maintenance" now represent very large portions of the cost of a project. Only a large experiment in which careful records are kept would enable a suitable evaluation to be made.

Certainly, extrapolation from small examples is very misleading if for no other reason than that it is possible to handle the complexity of a small example without the aid of formal methods.

The discussion has already mentioned notation several times. As well as the symbols introduced in section 1.1, it is necessary to present a general notation for talking about parts of programs and their properties. The choice of notation is partly a matter of taste, but the use of concepts which express naturally ideas of programming will obviously simplify the development process. The author's early attempts at "Formal Development" (ref 2) used mathematical functions and their compositions because this was a system in which formal arguments were familiar. This not only presented a barrier to some programmers but also created the task of accurately translating such a collection of functions into more efficient programming constructs. The system proposed in sections 2-4 below is the author's attempt to provide the basis of a system closer to programming concepts.

The system to which that proposed below is closest is that of Hoare (ref 6). The differences are the result of two difficulties with that system.

The first difficulty encountered with using Hoare's axioms was that termination is not treated: ref 7 in fact does not discuss termination until the complete algorithm has been developed. It is the view of the current author that termination should be proven at each stage of development of the algorithm.

A second difficulty resulted from the fact that the domains of both the pre and post conditions is a single state. Thus to require that an operation does not change the value of a variable requires the use of a free variable, e.g.:

$$x=x_0\{\,OP\,\}x=x_0$$

The system given below has post conditions of state pairs thus reducing the use of free variables.

1.1 Notation

> "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race." (Quote from "An Introduction to Mathematics" by Alfred North Whitehead)

The use of certain widely used mathematical and logical symbols is a very convenient shorthand: no use of involved theorems of logic is made below. The author has been made aware of the negative effect that the use of "strange symbols" can have on experienced programmers who might with their aid be able to document concisely very useful statements about programs. It is difficult to understand why a programmer, who nigh often have invested several weeks in learning a new machine, is not prepared to spare a few days with a text book such as ref 14. However, intuition based on the following "readings" of the symbols should suffice for all but the most detailed sections of the current paper.

Some use is made of sets (i.e. unordered collections):

Φ	the empty set
$\{a,b,\ldots,e\}$	the set containing the listed elements

In particular:-

$\{T,F\}$	the set containing the two possible logical values True and False
$\{x \mid p(x)\}$	the set containing all elements which have the property p
$x \in A$	is x a member of the set A?
card(A)	the number of elements in set A
$A \subseteq B$	(Subset) are all elements which are in A also in B?
$A \cup B$	the union of sets A and B contains exactly those elements which occur in A or B.
$\beta(A)$	the set of all subsets of A (i.e. $\beta(A) = \{X \mid X \subseteq A\}$

The concept of a function should be familiar. The set out of which its arguments are chosen and the set which contains all of its possible results is defined by writing:-

$$F : A \rightarrow R$$

or if, say, two arguments one from each of sets A1 and A2 are required:-

$$F : A1 \times A2 \rightarrow R$$

The majority of the special symbols used below are those of logic. Consider a property, say, p which divides a set of objects into those having the property and those not having it. p is called a predicate giving the values True and False respectively:

$$p : O \rightarrow \{T,F\}$$

The readings of the symbols used to define/combine predicates are:-

$p \wedge q$	p and q
$p \vee q$	p or q
$\neg p$	not p
$p \supset q$	p implies q ($p \supset q$ is the same as $\neg p \vee q$)
$p \equiv q$	p is equivalent to q
$(\exists x)(p(x))$	there exists at least one element with the property p.

1.2 Plan of the Paper

Sections 2 and 3 develop abstractions of what might be thought of as store and instructions. The "operations" of section 3 are expressed as relations on the "states" of section 2. Section 4 presents a system in which it is possible to reason about properties of operations. Section 5 describes the ideas of formal development which are related to a fixed state and Section 6 uses the results so far to give a non-trivial example. Section 7 considers how abstract forms of algorithms can be presented by using abstract states and Section 8 returns to the earlier example to illustrate this idea. Only enough results to tackle

the examples are given in the body of this paper but section 9 indicates how the system sketched here might be extended to cover more of real programming languages.

Lines of a section numbered with Arabic Numerals are not referred to from outside, whereas lines numbered with Roman Numerals might be referred to from other sections. References are of the form 7.2(xii) but the section number is omitted for references within the referencing section.

2. STATES

The concept of storage, which contains values which can be changed, is central to programming. This section offers "states" as an abstraction of storage concepts of particular languages. The concept of states will be used in the next section to present an abstraction of the statements of programming languages.

A state can be thought of as a way of obtaining (current) values of particular names. With a high level programming language this mapping is from identifiers to values which may themselves be structured (e.g. arrays); with a machine the mapping is from integers to elementary values.

It is necessary below to group together states which yield values for the same names and, furthermore, yield values of the same types. Notice such a class is not constrained to yield the same values. Using n for names and p for predicates which define classes of values, such a class of states is defined:

$$\Sigma = (\langle n_1 : p_1 \rangle, \langle n_2 : p_2 \rangle, \ldots, \langle n_n : p_n \rangle)$$

For any state of this class, say:-

$\sigma^1 \in \Sigma$

The value of the ith name will belong to the class specified by the ith predicate.

The obvious notation for the value associated with a name, say n, is a state σ^1 might be n^1. This, indeed, is the notation used in most of the sequel. However, since the concepts introduced here are to underlay what follows a more formal notation is used as the basis and the above notation employed as an abbreviation. The state functions to be used are those of ref 1. The contents function:

$$c : NAME \times STATE \rightarrow VAL$$

Can be used as follows:-

$c(n, \sigma^1)$ abbreviated to n^1

In order to give a formal treatment of the property of a state containing different values at different times, a function:

$$a : NAME \times VAL \times \Sigma \rightarrow \Sigma$$

is used which yields a new state, of the identical structure to its third argument: the values differing only for the given name and there yielding the value given as a second argument.

With the intuitive readings of "contents" and "assign" the following properties should be acceptable (notice the concept of state has not allowed two names to refer to the same state element):

$$c(n, a(n, v, \sigma)) = v$$
$$n_1 \neq n_2 \supset c(n_1, a(n_2, v, \sigma)) = c(n_1, \sigma)$$

These two properties are used without reference in the sequel.

The contents abbreviation will be further extended to expressions like:-

n^1+m^1 for $c(n, \sigma^1) + c(m, \sigma^1)$

[For a given form of expressions, it would be possible to define a function, say Ψ, which would give:-

$\Psi("n^1+m^1", \sigma^1) = n(\sigma^1)+m(\sigma^1)$]

Consideration of large examples (e.g. ref 2) would require use of states with more complex structure. The notion of ULD objects (ref 3) would be one way of tackling this extension. The current simple store is adequate for the examples considered in the current paper even in section 7 which shows how the use of abstract states can aid Formal Development.

3. OPERATIONS

This section introduces the notion of Operations, which was partly prompted by ref 4. One way of viewing the statements of programming languages is to divide them into three groups as follows:

a) Statements which change the values stored in the state but leave its structure unchanged (e.g. assignment statements)

b) Statements, or constructs, which change the structure of the state (e.g. allocation or block structure).

c) Methods of combining units, possibly statements, to form other units (e.g. the conditional construct).

Perhaps the most commonly used statements come from a) and it is their property of changing the contents of the state which is treated in section 3.1 (and again in section 4.1) and which characterises an "operation". However, the main discussion will be of ways of combining operations. The reason for this emphasis is twofold. On the one hand, although the system given here does not attempt to be definitive, the degree of similarity in this area across languages makes it more widely applicable. On the other hand, section 5 will show that these combination rules can be used as ways of decomposing a Formal Development.

The categories a) and b) are discussed again in section 9.

3.1 Operations as relations on states

Given a class of states, say Σ, and members thereof σ^1, σ^2 etc, operations are considered as relations on states:

$$OP \subset \Sigma \times \Sigma$$

(The majority of this paper ignores the possibility of non-determinism but see section 3.5 and section 4.5). Rather than discuss (deterministic) operations as functions:

$$OP : \Sigma \rightarrow \Sigma$$

emphasis will be put on their preservation of the state structure by writing:-

$$OP :: \Sigma$$

The assertion that σ^1 and σ^2 are related by the operation of OP

is written:-

$$\sigma^1[OP]\sigma^2$$

The intuitive idea of what the relation means is that OP will transform a state σ^1 into a state σ^2.

The reader may find it useful to think of an interpretation of the relation via a "machine":-

$$I : OP \times \Sigma \rightarrow \Sigma$$

such that:-

$$\sigma^1[OP]\sigma^2 \equiv \sigma^2 = I(OP, \sigma^1)$$

In order to be a useful model of program statements, it is necessary to admit the possibility that an operation may be partial (i.e. will produce no output for some input values):-

$$(\exists\sigma^1)(\sim(\exists\sigma^2)(\sigma^1[OP]\sigma^2))$$

3.2 Combining Operations

This section defines the interpretation given to the three simplest ways of combining program statements.

3.2.1 Sequencing

Suppose two given operations work on the same state:-

i) $OP1, OP2 :: \Sigma$

Then their combination so as to run one after another in sequence will work on the same class of states:-

ii) $(OP1;OP2) :: \Sigma$

and some particular state will be transformed by the sequence to exactly that state which would be obtained by first applying one operation, then the other:-

iii) $\sigma^1[OP1;OP2]\sigma^3 \equiv (\exists\sigma^2)(\sigma^1[OP1]\sigma^2 \wedge \sigma^2[OP2]\sigma^3)$

3.2.2 Conditional

Suppose, as well as:-

i) $OP1, OP2 :: \Sigma$

A predicate expression p working on, but not changing Σ is given, then the conditional combination will also work on Σ:-

ii) (if p then OP1 else OP2)::Σ

and the state transformation will be either that of OP1 or of OP2 depending on whether $p(\sigma^1)$ (strictly $\Psi(p,\sigma^1)$, of section 2) is true:-

iii) $p(\sigma^1) \supset (\sigma^1[\text{if p then OP1 else OP2}]\sigma^2 \equiv \sigma^1[\text{OP1}]\sigma^2)$

iv) $\sim p(\sigma^1) \supset (\sigma^1[\text{if p then OP1 else OP2}]\sigma^2 \equiv \sigma^1[\text{OP2}]\sigma^2)$

3.2.3 Repetition

Suppose again:-

i) OP::Σ

and p is as in 3.2.2, then the repetition combination will also work on states of Σ:-

ii) (while p do OP)::Σ

If $p(\sigma^1)$ is false then the construct makes no change to the state:-

iii) $\sim p(\sigma^1) \supset (\sigma^1[\text{while p do OP}]\sigma^1)$

Otherwise the combination transforms the state using OP1 and reapplies the whole combination:-

iv) $p(\sigma^1) \supset (\sigma^1[\text{while p do OP}]\sigma^3 \equiv (\exists\sigma^2)(\sigma^1[\text{OP}]\sigma^2 \wedge \sigma^2[\text{while p do OP}]\sigma^3))$

Notice that the while property is recursive and does not give an immediate way of proving properties about while loops: this requires knowledge about (inductive) properties of the states.

(By using, for example, restricted identity relations for the tests, it would be possible to present a more complete theory in terms of relations: this is not done since it is the system of relations between conditions (see section 4) which is of interest.)

3.3 Example

In order to illustrate the use of the above combinations, a program:-

```
OP :: S where S = (<x:is-int>,
                   <y:is-int>,
                   <res:is-int>)

OP = (res:=0; while x≥y do (x:=x-y;res:=res+1))
```

which performs the integer division of x by y using successive subtraction is proven to perform:-

1. $x=5 \wedge y=3[\supset P]x=2 \wedge y=3 \wedge res=1$

proof:

Although no information is given about assignment most machines should give:-

2. $x=5 \wedge y=3[res:=0]x=5 \wedge y=3 \wedge res=0$

So, in order to prove 1, we must show (cf 3.2.1(iii)):-

3. $x=5 \wedge y=3 \wedge res=0[\underline{while}\ x \geq y\ \underline{do}\ (x:=x-y;res:=res+1)]\ x=2 \wedge y=3 \wedge res=1$

now since $3 \geq 2$, 3.2.3(iv) will be used and since it should again be true that:-

4. $x=5 \wedge y=3 \wedge res=0[x:=x-y;res:=res+1]x=2 \wedge y=3 \wedge res=1$

in order to prove 3, we must show (cf 3.2.3(iv)):-

5. $x=2 \wedge y=3 \wedge res=1[\underline{while}\ x \geq y\ \underline{do}\ (x:=x-y;res:=res+1)]\ x=2 \wedge y=3 \wedge res=1$

which, since $\sim(1 \geq 2)$, is immediate from 3.2.3(iii).

Thus the assertion 1 was true.

The effect of the above "proof" has been no more than that of running a test case on I of section 3.1: all that has been done is to establish, using the formal system of section 3.2, what result arises from one particular value of the state. However, it is this "formal system" on which the system of section 4, which permits proofs about classes of values, is built.

3.4 (Extended) Operations

There are features in high-level programming languages which require an extension of the notion of (restricted) operations given in section 3.1. This section presents the required generalisation to which subsequent unqualified occurences of "operations" refer. As well as the ability to refer to and change a state, some programming language statements also permit the use of an explicit argument list (e.g. procedure call). Furthermore there are constructs which also deliver extra results (e.g. function reference). Given a state Σ a domain A and a range Ω, the type of an operation might be:

$$OP : \Sigma \times A \rightarrow \Sigma \times \Omega$$

again to emphasise the immutability of the structure of the state, this is written:-

$$OP :: \Sigma : A \rightarrow \Omega$$

Unlike the position with states, where uniformity of notation is guaranteed by omission, it is not clear how the arguments/results should be shown in the attempted abstraction of programming languages. Various notations are used below all of which should be clear. Writing:-

$$\sigma^1, \alpha[OP]\sigma^2, \omega$$

the, informally, described constraints on p in section 3.2.2 can now be written:-

$$p :: \Sigma : \Phi \rightarrow \{T, F\}$$

$$\sigma^1[p]\sigma^1, T \supset (\sigma^1[\underline{\text{if}}\ p\ \underline{\text{then}}\ OP1\ \underline{\text{else}}\ OP2]\sigma^2 \equiv \sigma^1[OP1]\sigma^2) \text{ etc.}$$

The subject of extensions is returned to in section 9.

3.5 Problems of non-deterministic Operations

The obvious justification for viewing operations as relations rather than functions is to cover non-deterministic operations. In spite of this the body of the current paper ignores non-determinism for two reasons. Firstly, it would tend to cloud the "normal" case as indicated below. Secondly, it is of real practical value only when parallelism is treated (see ref 5 for a discussion of these problems). Only this section and section 4.5 discuss non-determinism.

Consider first the deterministic interpretation I given in section 3.1:-

$$I: OP \times \Sigma \rightarrow \Sigma$$
$$\sigma^1[OP]\sigma^2 \equiv \sigma^2 = I(OP, \sigma^1)$$

then:-

$$\sim(\exists\sigma^2)(\sigma^1[OP]\sigma^2)$$

says that the (partial) operation OP is not defined for σ^1.

If I is non-deterministic it is tempting to define I* as the machine which yields the set of all possible results from I:-

$$I^* : OP \times \Sigma \rightarrow \beta(\Sigma)$$

and:-

$$\sigma^1[OP]\sigma^2 \equiv \sigma^2 \in I^*(OP,\sigma^1)$$

Now the problem is that for partial non-deterministic operations:-

$$(\exists\sigma^2)(\sigma^1[OP]\sigma^2)$$

no longer indicates that the machine will always terminate for input state σ^1. A machine which yields different results on different runs with the same starting state is likely to be more usable than a machine which delivers a result on one run but loops indefinitely on a different run with the same input state. Characterising the former class, informally for the moment, as "useful" non-determinism it is interesting that this is the most natural view of an operation as a relation on input states.

In order to approach the problem more formally define:-

$$\Sigma' = \Sigma \cup \{\underline{u}\}$$

where $\underline{u}$ is an undefined symbol denoting non-termination. Then let:-

$$I' : OP \times \Sigma \rightarrow \beta(\Sigma')$$

be a machine like I* but which also creates $\underline{u}$ in the result set if I may fail to terminate for σ^1. Thus:-

$$\sigma^1[OP]\underline{u}$$

rather than omission from the relation, denotes non-termination. "Useful" non-determinism might then be defined by:-

$$\sigma^1[OP]\sigma^2 \supset \sim\sigma^1[OP]\underline{u}$$

This would be very convenient but our "useful" class is not now closed under the combinations given in section 3.2. Consider:-

$$\sigma^1[OP1;OP2]\sigma^3 \equiv (\exists\sigma^2)(\sigma^1[OP1]\sigma^2 \wedge \sigma^2[OP2]\sigma^3)$$

and suppose that OP1 and OP2 are both "useful" non-deterministic operations, say:-

$$\sigma^1[OP1]\sigma^2 \wedge \sigma^1[OP1]\sigma^4$$
$$\sigma^2[OP2]\sigma^3 \wedge \sigma^4[OP2]\underline{u}$$

(OP1;OP2) is not "useful" because if OP1 chooses to create σ^4, OP2 will fail to terminate! It is possible to close this combination by e.g.:-

$$\sigma^1[\,OP1;OP2\,]\sigma^3 \equiv ((\forall\sigma^2)(\sigma^1[\,OP1\,]\sigma^2 \supset \tilde{\sigma}^2[\,OP2\,]\underline{u}) \wedge (\exists\sigma^2)(\sigma^1[\,OP1\,]\sigma^2 \wedge \sigma^2[\,OP2\,]\sigma^3))$$

but the approach discussed in section 4 gives an alternative way of avoiding the problem which is reviewed in section 4.5.

4. PROPERTIES OF OPERATIONS

Section 3 has considered operations as relations. How are the relations to be defined? Actual operations may work on huge numbers of different input states so it is not practical to enumerate the state pairs. The answer is to define a predicate over state pairs which yields true if, and only if, the pair is meant to belong to the relation. The predicate is, in fact, the formal expression of the required input/output specification of the operation. Its definition can be built up, using the connectives of section 1.1. from understood symbols (e.g. factorial in section 4.3). Again most of the discussion, except section 4.5, assumes deterministic operations but there may be more than one operation which satisfies the predicate. In other words, the predicate may not determine which deterministic operation is required.

4.1 Notation

Suppose some operation is required:-

$$OP :: \Sigma$$

The simplest specification of its required properties would use a predicate:-

$$post : \Sigma \times \Sigma \to \{T, F\}$$
$$\sigma^1[OP]\sigma^2 \supset post(\sigma^1, \sigma^2)$$

However, this is only a constraint on OP for those input states for which it produces a result. This is not sufficient and another predicate:-

$$pre : \Sigma \to \{T, F\}$$

is used to specify the (minimum) domain of input states for which OP should produce a result as follows:-

$$pre(\sigma^1) \supset (\exists\sigma^2)(\sigma^1[OP]\sigma^2)$$

The correctness criteria can now be given as:-

$$pre(\sigma^1) \wedge \sigma^1[OP]\sigma^2 \supset post(\sigma^1, \sigma^2)$$

The above conditions are used so often that an abbreviation is adopted:-

pre <OP> post

is written only if:-

$pre(\sigma^1) \wedge \sigma^1[OP]\sigma^2 \supset post(\sigma^1, \sigma^2)$

and:-

$pre(\sigma^1) \supset (\exists\sigma^2)(\sigma^1[OP]\sigma^2)$

Readers who are familiar with ref 6 will readily see that:-

$P\{OP\}Q$

is written only if:-

$P(\sigma^1) \wedge \sigma^1[OP]\sigma^2 \supset Q(\sigma^2)$

Thus the difference in the notations is that termination is separated in Hoare's work (cf. ref 7) and that his (post) conditions are of single states. It will become clear below that the conditions on state pairs are less easy to manipulate. The argument for their use is that it is the input/output relation of an operation which is of interest, not just some property of the output state. The attempt to get by with such properties has caused many of the proven algorithms since ref 8 to have variables which record the initial values.

4.2 Combining Operations via Their Conditions

This section gives requirements for deducing results about combinations of operations where the base operations are known only via conditions (section 5 will take the alternative view that given required conditions for an operation, sub operations can be specified to fulfil the task). It will become clear that this way of proving results is the key to avoiding the "test-case" proofs of section 3.3. Many different sets of requirements could be designed and those given below are neither the shortest nor claimed to be the strongest. The requirements given below are chosen only to facilitate the example developed in section 6.

4.2.1 Compound Statement

The simplest combination of two operations is to perform one then the other (cf section 3.2.1). Given:-

i) OP1::Σ and pre1<OP1>post1
ii) OP2::Σ and pre2<OP2>post2

then:-

(OP1;OP2) :: Σ

and, providing it can be shown that, firstly, both operations are used only over their domains:-

iii) $pre(\sigma^1) \supset pre1(\sigma^1)$
iv) $pre1(\sigma^1) \wedge post1(\sigma^1,\sigma^2) \supset pre2(\sigma^2)$

Secondly, the input/output relations combine in the correct way:-

v) $pre1(\sigma^1) \wedge post1(\sigma^1,\sigma^2) \wedge post2(\sigma^2,\sigma^3) \supset post(\sigma^1,\sigma^3)$

then:-

vi) pre<OP1;OP2>post

The proof of this result is now given. For many readers the knowledge that a proof exists may be sufficient information.

Assume:
1. $pre(\sigma_1)$

then from iii:-
2. $pre1(\sigma_1)$ 1

from i:-
3. $pre1(\sigma^1) \supset (\exists\sigma^2)(\sigma^1[OP1]\sigma^2)$ letting σ_2 be such:-
4. $\sigma_1[OP1]\sigma_2$ 2,3

and from i:-
5. $pre1(\sigma^1) \wedge \sigma^1[OP1]\sigma^2 \supset post1(\sigma^1,\sigma^2)$
6. $post1(\sigma_1,\sigma_2)$ 2,4,5

then from iv:-
7. $pre2(\sigma_2)$ 2,6

from ii:-
8. $pre2(\sigma^1) \supset (\exists\sigma^2)(\sigma^1[OP2]\sigma^2)$ letting σ_3 be such:-
9. $\sigma_2[OP2]\sigma_3$ 7

so from 3.2.1(iii):-
10. $\sigma_1[OP1;OP2]\sigma_3$ 4,9

and from ii:-
11. $pre2(\sigma^1) \wedge \sigma^1[OP2]\sigma^2 \supset post2(\sigma^1,\sigma^2)$
12. $post2(\sigma_2,\sigma_3)$ 7,9

then from v:-
13. $post(\sigma_1,\sigma_3)$ 2,6,12

so:-
14. $pre(\sigma_1) \supset (\exists\sigma_3)(\sigma_1[OP1;OP2]\sigma_3)$ 1,10
15. $pre(\sigma_1) \wedge \sigma_1[OP1;OP2]\sigma_3 \supset post(\sigma_1,\sigma_3)$ 1,10,13

thus:-
16. pre<OP1;OP2>post

The extension of the requirements i - vi to longer compound statements should be clear.

4.2.2 Repetition Statement

Interesting programs can only be constructed using some form of repetition construct. The simplest form is the while loop. This is so frequently used after an initialisation operation that the requirements given below are for the combination. Given:

i) INIT::Σ and T<INIT>post-INIT
i.e. INIT is total in that it will work on any state of type Σ
ii) BODY::Σ pre-BODY<BODY>post-BODY
iii) p::Σ:Φ → {T,F}

then:-
(INIT;<u>while</u> p <u>do</u> BODY)::Σ

and, providing that with:-
iv) pre-LOOP:Σ → {T,F}
which limits the valid domain of loop, and
v) post-LOOP:Σ x Σ → {T,F}
which gives the input/output predicate for the loop

it can be shown that:-
vi) $pre(\sigma^1) \wedge post\text{-}INIT(\sigma^1,\sigma^2) \supset pre\text{-}LOOP(\sigma^2) \wedge post\text{-}LOOP(\sigma^2,\sigma^2)$
vii) $pre\text{-}LOOP(\sigma^1) \wedge p(\sigma^1) \supset pre\text{-}BODY(\sigma^1)$
viii) $pre\text{-}BODY(\sigma^1) \wedge post\text{-}BODY(\sigma^1,\sigma^2) \supset pre\text{-}LOOP(\sigma^2)$
ix) $post\text{-}LOOP(\sigma^1,\sigma^2) \wedge post\text{-}BODY(\sigma^2,\sigma^3) \supset post\text{-}LOOP(\sigma^1,\sigma^3)$
x) $post\text{-}INIT(\sigma^1,\sigma^2) \wedge post\text{-}LOOP(\sigma^2,\sigma^3) \wedge pre\text{-}LOOP(\sigma^3) \wedge \sim p(\sigma^3) \supset post(\sigma^1,\sigma^3)$

and further, that with:-

xi) term:$\Sigma \rightarrow \{0,1,\ldots\}$

it can be shown that:-
xii) pre-LOOP$(\sigma^1) \supset$ term$(\sigma^1) \geq 0$
xiii) term$(\sigma^1)=0 \equiv \sim$p(σ^1)
xiv) post-BODY$(\sigma^1,\sigma^2) \supset$ term$(\sigma^2) <$ term(σ^1)

then:-
pre<INIT;<u>while</u> p <u>do</u> BODY>post

Again a proof of this result is given.

The first part of the proof establishes:-

1. pre-LOOP$(\sigma^1) \wedge$ post-LOOP$(\sigma^0,\sigma^1) \supset (\exists\sigma^3)(\sigma^1$[<u>while</u> p <u>do</u> BODY$]\sigma^3)$
2. pre-LOOP$(\sigma^1) \wedge$ post-LOOP$(\sigma^0,\sigma^1) \wedge \sigma^1$[<u>while</u> p <u>do</u> BODY$]\sigma^3 \supset$ pre-LOOP$(\sigma^3) \wedge$ post-LOOP$(\sigma^0,\sigma^3) \wedge \sim$p$(\sigma^3)$

the proof is by induction on term$(\sigma^1) \geq 0$, see xii. Basis assume:-
3. term$(\sigma^1) = 0$

from xiii:- 3
4. $\sim$p(σ^1)

from 3.2.3(iii):- 4
5. σ^1[<u>while</u> p <u>do</u> BODY$]\sigma^1$

thus:- 5
6. $(\exists\sigma^3)(\sigma^1$[<u>while</u> p <u>do</u> BODY$]\sigma^3)$

and since $\sigma^3 = \sigma^1$ the hypotheses give:- 4
7. pre-LOOP$(\sigma^3) \wedge$ post-LOOP$(\sigma^0,\sigma^3) \wedge \sim$p$(\sigma^3)$

now assume the result proven for term$(\sigma^1) < n$, prove for term$(\sigma^1) = n > 0$, from xiii:-
8. p(σ^1)

from vii:- 8
9. pre-BODY(σ^1)

from ii:-
10. pre-BODY$(\sigma^1) \supset (\exists\sigma^2)(\sigma^1$[BODY$]\sigma^2)$ 9,10
11. σ^1[BODY$]\sigma^2$

and from ii:-
12. pre-BODY$(\sigma^1) \wedge \sigma^1$[BODY$]\sigma^2 \supset$ post-BODY(σ^1,σ^2) 9,11,12
13. post-BODY(σ^1,σ^2)

then from viii:-
14. pre-LOOP(σ^2) 9,13

and from ix:-
15. post-LOOP(σ^0,σ^2) 13

and from xiv:-
16. term(σ^2) < term(σ^1) 13

so by induction hypotheses:-
17. ($\exists\sigma^3$)(σ^2[<u>while</u> p <u>do</u> BODY]σ^3) 14,15

further:-
18. pre-LOOP(σ^3) ∧ post-LOOP(σ^0,σ^3) ∧ ~p(σ^3)

so from 3.2.3(iv):-
19. σ^1[<u>while</u> p <u>do</u> BODY]σ^3 8,11,17

and 18 gives the required properties.

Now the above result can be used as follows, assume:-
20. pre(σ_1)

from i:-
21. T ⊃ ($\exists\sigma^2$)(σ^1[INIT]σ^2) letting σ_2 be such:-
22. σ_1[INIT]σ_2 21

and from i:-
23. T ∧ σ^1[INIT]σ^2 ⊃ post-INIT(σ^1,σ^2)
24. post-INIT(σ_1,σ_2) 22,23

and from vi:-
25. pre-LOOP(σ_2) ∧ post-LOOP(σ_2,σ_2) 20,24

the above results are now used:-
26. ($\exists\sigma^3$)(σ_2[<u>while</u> p <u>do</u> BODY]σ^3) letting such be σ_3 1,25
27. pre-LOOP(σ_3) ∧ post-LOOP(σ_2,σ_3) ∧ ~p(σ_3) 2,25,26

from 3.2.1(iii):-
28. σ_1[INIT;<u>while</u> p <u>do</u> BODY]σ_3 22,26
29. post(σ^1,σ^3) 24,27,28,

which concludes the proof.

4.3 The Example

It is customary to supply at about this point in papers on program correctness a proof of the correctness of an algorithm for factorial. The custom is followed for the purposes of illustrating the use of the properties of section 4.2. A more interesting example is included after the ideas of formal development are covered in section 5.

Given states:

1. $S = (\langle n:\text{is-int}\rangle, \langle fn:\text{is-int}\rangle)$ $\qquad s, s_1, s_2, \ldots \in S$

it is required to prove:-

2. pre<fn:=1; **while** n≠0 **do** (fn:=fn.n;n:=n-1)>post

where:-

3. $pre(s_1) = n_1 \geq 0$ $\qquad$ i.e. $c(n, s_1) \geq 0$
4. $post(s_1, s_2) = fn_2 = n_1!$

accepting:-

5. I<fn:=1>post-INIT

where:-

6. $post\text{-}INIT(s_1, s_2) = n_2 = n_1 \wedge fn_2 = 1$

and:-

7. pre-BODY<fn:=fn.n;n:=n-1>post-BODY

where:-

8. $pre\text{-}BODY(s_1) = n_1 > 0$
9. $post\text{-}BODY(s_1, s_2) = fn_2 = fn_1 . n_1 \wedge n_2 = n_1 - 1$

using:-

10. $pre\text{-}LOOP(s_1) = n_1 \geq 0$
11. $post\text{-}LOOP(s_1, s_2) = fn_2 . n_2! = fn_1 . n_1!$
12. $term(s_1) = n_1$

results vi, vii, viii of section 4.2.2 follow immediately. Consider ix, expanding the hypotheses gives:-

13. $fn_2 . n_2! = fn_1 . n_1!$ $\qquad$ 11
14. $fn_3 = fn_2 . n_2$ $\qquad$ 9
15. $n_3 = n_2 - 1$ $\qquad$ 9

Now:-

16. $fn_3 . n_3! = fn_2 . n_2 . (n_2 - 1)!$ $\qquad$ 14,15

and the obvious property of factorial gives:-

17. $fn_3.n_3! = fn_2.n_2!$ 16
18. $= fn_1.n_1!$ 17,13

Thus:-

19. post-LOOP(s_1,s_3) 18,11

Consider x, expanding the hypotheses gives:-

20. $n_2 = n_1 \wedge fn_2 = 1$ 6
21. $fn_3.n_3! = fn_2.n_2!$ 11
22. $n_3 = 0$

Using another property of factorial and simplifying gives:-

23. $fn_3 = n_1!$ 20,21,22

Thus:-

24. post(s_1,s_3) 23.

Results xii, xiii and xiv follow immediately.

Notice that no temporary variable is used in the state, a formal proof using the system of ref 6 is more difficult. The reader might choose at this point to attempt an exercise. Using the example of section 3.3, given:

```
S = (<x:is-int>,
     <y:is-int>,
     <res:is-int>)
```

Prove:-

pre<res:=0;<u>while</u> x≥y <u>do</u>(x:=x-y;res:=res+1)>post

where:-

pre(s_1) = $x_1 \geq 0 \wedge y_1 > 0$
post(s_1,s_2) = $y_1.res_2+x_2 = x_1 \wedge 0 \leq x_2 < y_1$

4.4 Properties of Extended Operations

In order to present arguments about the operations presented in section 3.4, i.e.:-

OP::Σ : A → Ω

two predicates:-

pre:Σ x A → {T,F}
post:Σ x A x Σ x Ω → {T,F}

are used, so that:-

pre<OP>post

is written only if:-
$pre(\sigma^1,\alpha) \supset (\exists\sigma^2,\omega)(\sigma^1,\alpha[OP]\sigma^2,\omega)$
$pre(\sigma^1,\alpha) \wedge \sigma^1,\alpha[OP]\sigma^2,\omega \supset post(\sigma^1,\alpha,\sigma^2,\omega)$

The extension of the combination rules to cover extended operations is straightforward.

4.5 Non-Deterministic Operations Again

As foreseen in section 3.5, it is now possible to cover the problem of non-deterministic operations via their properties. Suppose, using the undefined symbol of section 3.5:-

pre<OP>post

is written only if:-
$pre(\sigma^1) \supset \sim\sigma^1[OP]\underline{u}$
$pre(\sigma^1) \wedge \sigma^1[OP]\sigma^2 \supset post(\sigma^1,\sigma^2)$

Then it will be seen that the above class is closed under combination rules of section 4.2.

5. FORMAL DEVELOPMENT

This section describes the method of formal development leaving aside until section 7 the very important issue of how the data can be evolved along with the algorithm. Section 6 gives an example of formal development on constant data.

The background is to accept the notion that the art of programming is to go from an implicit specification of what the task is, to a method for how to achieve that task. An example of "what" from mathematics might be:

> algorithm s such that given a positive argument x
> it delivers a result y such that $y^2 = x$

The "how" is some appropriate square root algorithm. In most current programming languages the definition of "how" will be a program which invokes ordered repetitions of operations. It is usually more difficult to write a program which evokes the sequence of operations than to state the required result.

The first step is to obtain the implicit specification. Clearly it must be unambiguous and non-contradictory. It should also be "complete" in the sense that any proferred algorithm which fulfils the stated criteria should not be rejected because it fails to fulfil a property which was not stated. As far as is practical it should also avoid being over-specific: it should not rule out acceptable algorithms. For all practical purposes this implies that it must use a notation which avoids the dangers of looseness that can be hidden by using natural language. "Specifications" are a subject in themselves which cannot be dwelt on in the current paper but it is worth pointing out the work in specifying programming languages (ref 3). The work on constructing such formal specifications is by no means a waste: the greater understanding and the early resolution of inconsistencies or incompleteness is a considerable reward.

Providing the given "problem" is non-trivial (i.e. not in the repertoire of instructions of the machine), the next step is to decide on some operations into which the job can be decomposed. That is, think of building blocks which if one had them, could be put together with one of the known ways of combining operations to achieve the required result. Notice that these operations are not necessarily primitive operations of the target machine, they will in general be characterised by assumptions about their behaviour. (The fact that these assumptions are also recorded formally is the point missing in the normal process of breaking a task up into sub-routines.) The claim that the proposed way of combining the blocks fulfils the specification now becomes the

statement of a theorem. Under the documented assumptions the justification of this theorem should then be given. It is worth repeating that the idea for why the construction is correct should already be in the programmers mind: he is only being asked to document rather than discard the argument.

The aim should be to stay as abstract as possible. This of course means that the algorithm using assumed (abstract) operations is not yet capable of running on a machine. Having proven the (abstract) algorithm correct so far, it is now possible to completely ignore the work of this stage. Now one takes the list of operations about which only assumptions on what they do are known and uses these assumptions as specifications for further development. Since any stage can introduce more than one assumed operation the development appears like a tree. Ultimately each branch will terminate when the assumptions about an operation can be met by primitives of the target language. There is a strong reason to suspect that, given a way of rigorously documenting sub-tasks, programmers will be less inclined to "rush into code".

This brief summary begs many questions, some of which will become clear on considering the example in section 6. Some are considered here.

One naturally wonders how large a step of development is appropriate. The danger of too large a step is that it will both confuse the details and increase the chance of backtracking. The actual step size will obviously be a matter of individual style. The only rule would appear to be that independent decisions should be separated.

In spite of taking care it is inevitable that erroneous steps will sometimes be made and there is no alternative but to trace an error back to the stage which introduced it and repeat the development from that point. Faced with several sub-problems, it should be clear that the one to be pursued first is that which is most likely to force a backtrack.

The careful choice of building blocks will not only yield more insight into the chosen algorithm but will also give a clear framework for considering alternatives.

This process of decomposition can also be applied to other requirements than input/output relations. It would for instance be possible to split resources of core, men, time etc to cover the tree of development.

Another interesting effect of the development is its possible affect on language features. Suppose a development has reached the point where an operation is required which simply tests whether any element of given array has a given property. This is not a primitive of the commonly used languages and another step might create a do loop type construct. If there is no reason to search in any particular order, it is not only tedious to do this, it also makes the job of an optimising compiler or parallel machine unnecessarily difficult. If one found constructs of this sort (cf the discussion of assignment in ref 2) occuring frequently one might be able to influence language design from the problem end rather than the machine end.

The author has frequently been met with the objection, by this point in a presentation, "If we knew our specs and they didn't change, the job is anyway easy!" Apart from some reservation the reaction to this is that a documented development of the proposed type would give a good framework in which to consider the effect of changes. If generality and abstraction have been used fully then incorporation of changes should not be disproportionately difficult (cf section 8 of ref 2).

5.1 Example

In order to see these points on a trivial example, consider the "exercise" of section 4.3 as a development problem.

Find an:-
DIV::S

Such that:-
pre<DIV>post

Where:-
S = (<x:is-int>,
<y:is-int>,
<res:is-int>)
$pre(s_1) = x_1 \geq 0 \wedge y_1 > 0$
$post(s_1, s_2) = y_1 . res_2 + x_2 = x_1 \wedge 0 \leq x_2 < y_1$

A "stage of development" might recognise the requirement to decompose this problem into two tasks, INIT and LOOP, as follows:-

INIT::S
I<INIT>post-INIT
$post\text{-}INIT(s_1, s_2) = x_2 = x_1 \wedge y_2 = y_1 \wedge res_2 = 0$

LOOP::S

pre-LOOP<LOOP>post-LOOP
pre-LOOP(s_1) = pre(s_1)
post-LOOP(s_1,s_2) = $y_1.res_2 + x_2 = x_1 + y_1.res_1 \wedge 0 \leq x_2 < y_1$

Using the rules of section 4.2.1 it is possible to prove:-
pre<INIT;LOOP>post

Notice that the view taken of the ways of combining programs (e.g. see section 4.2) is of tools to decompose a problem. Also that even in this trivial example it was possible to make the two sub-tasks more independent by not giving the second element all of the information about the context in which it is to be used (i.e. res = 0).

6. EXAMPLE OF FORMAL DEVELOPMENT

This section provides an example of the formal development process described in section 5. The problem chosen is that of reference 7. As well as permitting a comparison with that paper, the example is about as long as is practical to present in a paper. Any comparison of length should be made with care because the current proofs include arguments for termination and preservation of elements which are treated seperately in ref 7. The specifications, and the whole development, are given in terms of arrays. The ideas of evolving data types are illustrated, using the same example, in section 8.

6.1 Specifications

Consider a given set of values:-

i) U

and a total ordering relation on pairs of these values:-

ii) $\leq.: U \times U \rightarrow \{T,F\}$

The normal properties of such relations hold:-

iii) $v_1 \leq. v_2 \vee v_2 \leq. v_1$

iv) $v_1 \leq. v_2 \wedge v_2 \leq. v_3 \supset v_1 \leq. v_3$

The relation is marked with a dot to distinguish it from the simple relation on integers ($\leq.$ might for example be a comparison between data structures governed only by the values of one "key" field). In particular, it is not assumed that:

$$v_1 \leq. v_2 \wedge v_2 \leq. v_1 \supset v_1 = v_2$$

The required program will operate on arrays, which can be considered to be (partial) mappings from integers to elements of U:-

v) $\underline{A} : I \rightarrow U$

The set of elements in some segment of an array will be given by:-

vi) $A(m:n) = \{A[x] \mid m \leq x \leq n\}$

Proofs of the relations implied by this are not all given below (e.g. $m \leq n-1$).

The overall state of the program will be:-

vii) S = (<A:is-A>,
<f:is-int>,
<N:is-int>)

Where is-int characterises the set of positive integers:-
$\{1,2,\ldots\}$

Thus:-
viii) FIND::S

The required properties of FIND:-
ix) pre<FIND>post

are that given a state where N contains a (non-zero) integer giving the number of elements in A, and f contains an integer value between 1 and N, i.e.:-
x) $pre(s) = N \geq 1 \wedge 1 \leq x \leq N \supset A[x] \in U \wedge 1 \leq f \leq N$

(Given such states) the program should create output states in which the array can be considered to be split about the fth element:-
xi) $post(S^1,S^2) = ordered\text{-}split(A^1(1:N),A^2(1:f-1),A^2(f:f),A^2(f:N))$

Such that the set of values in the array element of the state is unchanged by the execution and that the three sections created by the split are ordered i.e.:-
xii) $ordered\text{-}split(V,V^1,V^2,V^3) = gd\text{-}split(V,V^1,V^2,V^3) \wedge ordered(V^1,V^2,V^3)$

The formal expression of the intuitive idea of a "good-split" is that it must contain the same elements and the sets must be pairwise disjoint.
xiii) $gd\text{-}split(V,V^1,V^2,V^3) = V^1 \cup V^2 \cup V^3 = V \wedge pdis(V^1,V^2,V^3)$

xiv) $pdis(V(1),V(2),\ldots,V(n)) = i \neq j \supset \sim (v \in V(i) \wedge v \in V(j))$

Whilst ordered on sets is defined in terms of the given element-wise relation by:-
xv) $ordered(V(1),V(2),\ldots,V(n)) = i<j \wedge v(i) \in V(i) \wedge v(j) \in V(j) \supset v(i) \leq v(j)$

The above predicates were intentionally given suggestive names and the reader will be invited to accept a number of lemmas in the course of the development which fit with his intuitive ideas. Notice, however, that these predicates are defined and in cases of doubt lemmas about them can be proven.

The specification of a programming task will normally contain other constraints than the required input/output relation. Of those listed in section 5 only store usage will be considered here. It is assumed that the requirement is for a program which uses roughly N storage cells. Thus a program with a few local scalars is acceptable whilst one requiring a temporary array the same size as A is not.

(The utility of a program which takes an array of values and organises them so that the fth element is in the fth position and all those below (above) are less than (greater than) with respect to the given ordering relation is shown in reference 13 where recursive use of the routine on non-unit element parts provides a sort algorithm.)

6.2 First Stage of Development

The storage requirement to work within the given array might suggest an algorithm which collects low elements at one end of the array and high at the other, marking the limits of these two collections with scalars. Thus the state will be extended to:

i) S1 = (<A:is-A>,
<f:is-int>,
<N:is-int>,
<m:is-int>,
<n:is-int>)

The set of elements which remain to be arranged will be stored in A(m:n). The job of FIND can be performed by an algorithm which initally puts all elements in this class and then reduces it step by step until it contains exactly one (the required) element:

ii) FIND = m:=1;n:=N; **while** m≠n **do** BODY1

The requirements on the initialisation are:-

iii) (m:=1;n:=N)::S1

iv) T<m:=1;n:=N>post-INIT1

v) post-INIT1(s^1,s^2) = s^2 = a(n,N^1,a(m,1,s^1))

The body of the loop must ensure that the required element never gets absorbed into either outer set (and should assume this for its input states and that the task is incomplete) as well as making an ordered split of the previously unarranged elements and leaving alone items of no interest. In order to prove termination it is also required that some progress is made at each iteration. Formally:

vi) BODY1::S1

vii) pre-BODY1<BODY1>post-BODY1

viii) pre-BODY1(s^1) = $m^1 \leq f^1 \leq n^1 \wedge m^1 \neq n^1$

ix) post-BODY1(s^1, s^2) =
$m^2 \leq f^2 \leq n^2 \wedge$
$(f^2, N^2) = (f^1, N^1) \wedge$
undisturbed$(A^1, A^2, m^1, n^1) \wedge$
ordered-split$(A^1(m^1:n^1), A^2(m^1:m^2-1),$
$A^2(m^2:n^2), A^2(n^2+1:n^1)) \wedge$
progress(m^1, m^2, n^2, n^1)

The predicate undisturbed in true only if the values in the arrays correspond in all positions below m and above n:-

x) undisturbed(A^1, A^2, m, n) =
$1 \leq x < m \vee n < x \leq N \supset A^2[x] = A^1[x]$

The predicate progress is true only if neither m nor n have "moved backwards" and at least one of them has moved:

xi) progress(m^1, m^2, n^2, n^1) =
$m^1 \leq m^2 \wedge n^2 \leq n^1 \wedge$
$(n^2 - m^2) < (n^1 - m^1)$

Having assumed the above properties, two tasks remain. Firstly it is necessary to justify the argument that if we had such operations (in particular BODY1) using them as in ii would satisfy the requirements on FIND. Secondly, assuming no operation with the properties of BODY1 exists in our programming language, it must be further developed. The first task is tackled here, the second in section 6.3.

Defining:-

xii) pre-LOOP1(s^1) = $m^1 \leq f^1 \leq n^1$

xiii) post-LOOP1(s^1, s^2) =
$f^2 = f^1 \wedge$
ordered-split$(A^1(1:N), A^2(1:m^2-1), A^2(n^2:n^2), A^2(n^2+1:N))$

xiv) term1(s^1) = $n^1 - m^1$

The proof of (cf. 6.1(ix)):-

1. pre<m:=1;n:=N; <u>while</u> $m \neq n$ <u>do</u> BODY1>post

can be constructed according to section 4.2 as follows:-

For 4.2.2(vi)a:-

2. pre$(s^1) \wedge$ post-INIT1$(s^1, s^2) \supset$ pre-LOOP1(s^2)

is immediate from 6.1(x), v, xii

For 4.2.2(vi)b:-

3. pre$(s^1) \wedge$ post-INIT1$(s^1, s^2) \supset$ post-LOOP1(s^2, s^2)

is seen to be vacuously true from v, xiii, 6.1(xii)

For 4.2.2(vii):-
4. pre-LOOP1(s^1) ∧ $m^1 \neq n^1$ ⊃ pre-BODY1(s^1)

is immediate from xii, viii

For 4.2.2(viii):-
5. pre-BODY1(s^1) ∧ post-BODY1(s^1,s^2) ⊃ pre-LOOP1(s^2)

is immediate from ix, xii

Now for 4.2.2(ix) consider:-
6. post-LOOP1(s^1,s^2) ∧ post-BODY1(s^2,s^3)

From 6 and xiii:-
7. $f^2 = f^1$
8. ordered-split($A^1(1:N)$,$A^2(1:m^2-1)$,$A^2(m^2:n^2)$,$A^2(n^2+1:N)$)

and from 6 and ix:-
9. $(f^3,N^3) = (f^2,N^2)$
10. undisturbed(A^2,A^3,m^2,n^2)
11. ordered-split($A^2(m^2:n^2)$,$A^3(m^2:m^3-1)$,$A^3(m^3:n^3)$,$A^3(n^3+1:n^2)$)

Thus from 7 and 9:-
12. $f^3 = f^1$

Now by a lemma an ordered-split (which can be proved without difficulty from 6.1(xii), 6.1(xiii), 6.1(xiv),x
8, 10 and 11 give:-
13. ordered-split($A^1(1:N)$,$A^3(1:m^3-1)$,$A^3(m^3:n^3)$,$A^3(n^3+1:N)$)

Thus from 12 and 13 and xiii:-
14. post-LOOP1(s^1,s^3)

which from 6 is the required result for 4.2.2 (ix).

Now for 4.2.2(x) consider:-
15. post-INIT1(s^1,s^2) ∧ post-LOOP1(s^2,s^3) ∧ pre-LOOP1(s^3) ∧ $n^3 = n^3$

from 15, v:-
16. $A^2 = A^1$ ∧ $f^2 = f^1$

and from 15, xiii:-
17. $f^3 = f^2$
18. ordered-split($A^2(1:N)$,$A^3(1:m^3-1)$,$A^3(m^3:n^3)$,$A^3(n^3+1:N)$)

and from 15, xii:-
19. $m^3 = n^3 = f^3$

Thus from 18, 17, 19, 16:-
20. ordered-split($A^1(1:N), A^3(1:f^1-1), A^3(f^1:f^1), A^3(f^1+1:N)$)

or:-
21. post(s^1, s^3)

which from 15 is the result required by section 4.2.2(x).

Finally for termination:-
For 4.2.2(xii):-
22. pre-LOOP1(s^1) $\supset$ term1(s^1) ≥ 0

is immediate from xii, xiv

For 4.2.2(xiii):-
23. term1(s^1) = 0 $\equiv \sim(m^1 \neq n^1)$

is immediate from xiv

For 4.2.2(xiv):-
24. post-BODY1(s^1, s^2) $\supset$ term1(s^2) < term1(s^1)

is immediate from ix, xi, xiv.

So all of the required conditions of section 4.2.2 have been proved and this gives the required result 1.

6.3 Second Stage of Development

It is now necessary to show how BODY1 of the last section can be achieved by more basic operations. In order to collect the extra low and high elements which BODY1 provides for the outer sets, another two indices are used. These are used to record the extent of the new small and large elements where "size" is determined by comparison to one element of the array. Thus the state will be extended to:

i) S2 = (<A:is-A>,
<f:is-int>,
<N:is-int>,
<m:is-int>,
<n:is-int>,
<i:is-int>,
<j:is-int>,
<r:is-U>)

The task defined for BODY1 in section 6.2 will now be achieved by:-

ii) BODY1 = r:=A[f];SPLIT2;JOIN2

Where (r:=A[f]) is a total function which simply stores a value into the r component of the state:-

iii) (r:=A[f])::S2
iv) T<r:=A[f]>post-CHOOSE2
v) post-CHOOSE2(s^1,s^2) =
$s^2 = a(r,A^1[f^1],s^1)$

The operation SPLIT2 will, providing the value in r is one of those in the to be arranged area, so permute the values in that portion of A to yield an ordered split:-

vi) SPLIT2::S2
vii) pre-SPLIT2<SPLIT2>post-SPLIT2
viii) pre-SPLIT2(s^1) =
$(\exists x)\,(m^1 \le x \le n^1 \wedge r^1 = A^1[x]) \wedge m^1 \neq n^1$
ix) post-SPLIT2(s^1,s^2) =
$j^2 < i^2 \wedge$
undisturbed(A^1,A^2,m^1,n^1) $\wedge$
ordered-split($A^1(m^1:n^1)$,$A^2(m^1:j^2)$,
$A^2(j^2+1:i^2-1)$,$A^2(i^2:n^1)$) $\wedge$
$m^1 < i^2 \wedge j^2 < n^1 \wedge$
$(f^2,N^2,m^2,n^2) = (f^1,N^1,m^1,n^1)$

Notice that SPLIT2 is <u>not</u> required to so arrange the middle portion to include the fth position. This is the way in which development has proceeded: the required operation is now simpler.

The operation JOIN2 has the task of joining up the segments in such a way that this extra requirement on f is satisfied:

x) JOIN2::S2
xi) T<JOIN2>post-JOIN2
xii) post-JOIN2(s^1,s^2)
$f^1 \le j^1 \supset s^2 = a(n,j^1,s^1)$
$i^1 \le f^1 \supset s^2 = a(m,i^1,s^1)$
$j^1 < f^1 < i^1 \supset s^2 = a(n,f^1,a(m,f^1,s^1))$

Once again, of the two tasks remaining, this section shows that the assumed operations perform the required function (i.e.ii). Further development of SPLIT2 is done in section 6.4 whilst, for the purposes of this paper, JOIN2 is assumed to have an obvious equivalent in a programming language.

The proof of (cf. 6.2(vii)):-

1. pre-BODY1<r:=A[f];SPLIT2;JOIN2>post-BODY1

Follows:-

For 4.2.1(iii):-
2. pre-BODY1(s^1) ⊃ T

is vacuously true

For 4.2.1(iv):-
3. pre-BODY1(s^1) ∧ post-CHOOSE2(s^1,s^2) ⊃ pre-SPLIT2(s^2)

is immediate from 6.2(viii), v, viii

For 4.2.1(iv):-
4. pre-BODY1(s^1) ∧ post-CHOOSE2(s^1,s^2) ∧ post-SPLIT2(s^2,s^3) ⊃ T

is vacuously true

Now for 4.2.1(v) consider:-
5. pre-BODY1(s^1) ∧ post-CHOOSE2(s^1,s^2) ∧ post-SPLIT2(s^2,s^3) ∧ post-JOIN2(s^3,s^4)

From 5, 6.2(vii):-
6. $m^1 \leq f^1 \leq n^1 \wedge m^1 \neq n^1$

From 5, v:-
7. $(f^2,A^2,N^2,m^2,n^2) = (f^1,A^1,N^1,m^1,n^1)$

From 5, ix:-
8. $j^3 < i^3$
9. undisturbed(A^2,A^3,m^2,n^2)
10. ordered-split($A^2(m^2:n^2)$,$A^3(m^2:j^3)$,$A^3(j^3+1:i^3-1)$,$A^3(i^3:n^2)$)
11. $m^2 < i^3 \wedge j^3 < n^2$
12. $(f^3,N^3,m^3,n^3) = (f^2,N^2,m^2,n^2)$

From 5, xii:-
13. $(f^4,A^4,N^4) = (f^3,A^3,N^3)$

Thus from 13, 12, 7:-
14. $(f^4,N^4) = (f^1,N^1)$

and from 13, 9, 7:-
15. undisturbed(A^1,A^4,m^1,n^1)

To obtain the rest of the result sought it is necessary to consider three distinct cases.

Firstly, consider the case:-
16. $f^3 \leq j^3$

Then 5, xii gives:-
17. $(m^4,n^4) = (m^3,j^3)$

Thus from 17, 12, 7, 6 and 14:-
18. $m^4 \leq f^4$

and from 17, 16 and 14:-
19. $f^4 \leq n^4$

Now from 10 and an easily proved lemma from 6.1 (xii):-
20. ordered-split($A^2(m^2:n^2)$, $A^3(m^2:m^2-1)$, $A^3(m^2:j^3)$, $A^3(j^3+1:n^2)$)

Thus substituting with 7,13,12,7:-
21. ordered-split($A^1(m^1:n^1)$, $A^4(m^1:m^4-1)$, $A^4(m^4:n^4)$, $A^4(n^4+1:n^1)$)

and from 7, 12, 17 with 11, 6.2 (xi) gives:-
22. progress(m^1, m^4, n^4, n^1)

The second case:-
23. $i^3 \leq f^3$

is proved in a completely analagous way.

The remaining case is:-
24. $j^3 < f^3 < i^3$

Then 5, xii gives:-
25. $(m^4, n^4) = (f^3, f^3)$

Thus from 13:-
26. $m^4 \leq f^4 \leq n^4$

Now from 8, 10, 24 and an easily proved lemma from 6.1 (xii):-
27. ordered-split($A^2(m^2:n^2)$, $A^3(m^2:f^3-1)$, $A^3(f^3:f^3)$, $A^3(f^3+1:n^2)$)

Thus substituting with 7, 13, 25:-
28. ordered-split($A^1(m^1:n^1)$, $A^4(m^1:m^4-1)$, $A^4(m^4:n^4)$, $A^4(n^4+1:n^1)$)

and 6, 25 with 6.2 (xi) gives:-
29. progress(m^1, m^4, n^4, n^1)

Therefore the results necessary for:-
30. post-BODY1(s^1, s^4)

have been proven in all cases - 18, 19, 14, 15, 21, 22 or
26, 14, 15, 28, 29.

This is the result required by 4.2.1 to be proven under assumption 5.

Thus the proof of correctness of this stage is complete.

6.4 Third Stage of Development

This stage of development is very similar to the first in that the two pointers (i,j introduced in section 6.2) are used to delimit part of the array which has still to be considered. Elements over which the i (j) pointer moves are less than or equal (greater than or equal) the selected value r. In order to ensure termination the two scans are made also to stop at elements equal to r. In order to define this an irreflexive form of the ordering relation is used.

i) $<_. : U \times U \rightarrow \{T,F\}$
$v^1 <_. v^2 \equiv v^1 \leq_. v^2 \wedge \sim(v^2 \leq_. v^1)$

The job of SPLIT2 is to be performed by an iterative operation:-
ii) SPLIT2 = i:=m; j:=n; <u>while</u> i≤j <u>do</u> BODY3

The requirements on the initialisation are:-
iii) (i:=m; j:=n)::S2
iv) T<i:=m; j:=n>post-INIT3
v) post-INIT3(s^1,s^2) =
$s^2 = a(j,n^1,a(i,m^1,s^1))$

The body of the loop will operate providing there are elements to stop it "running-away" and there is still a gap in which to work. In order to ensure it can guarantee to meet its requirement it is also necessary to know that i and j have moved "off base" or this is the first iteration. The output requirements, other than specifying things which must not change, require that the resultant (overlapping?) segments are ordered with respect to the stored element; that progress has been made; and that if this were the first iteration both i and j have moved "off-base".

vi) BODY3::S2
vii) pre-BODY3<BODY3>post-BODY3
viii) pre-BODY3(s^1) =
$(\text{first-it}(A^1,m^1,n^1,i^1,j^1,r^1) \vee (n^1<i^1 \wedge j^1<n^1)) \wedge$
$\text{stoppers}(A^1,m^1,i^1,j^1,n^1,r^1) \wedge$
$i^1 \leq j^1$
ix) post-BODY3(s^1,s^2) =
$(f^2,N^2,m^2,n^2,r^2) = (f^1,N^1,m^1,n^1,r^1) \wedge$
$\text{undisturbed}(A^1,A^2,i^1,j^1) \wedge$
$\text{ordered}(A^2(i^1:i^2-1),\{r\},A^2(j^2+1:j^1)) \wedge$
$\text{progress}(i^1,i^2,j^2,j^1) \wedge$
$((\exists x)(i^1 \leq x \leq j^1 \wedge r^1 = A^1[x]) \supset i^1<i^2 \wedge j^2<j^1)$

Where:-

x) first-it(A,m,n,i,j,r) =
m=i ∧ j=n ∧
(∃x)(i≤x≤j ∧ r = A[x])

xi) stoppers(A,m,i,j,n,r) =
(∃x)(i≤x≤n ∧ ~A[x] <. r) ∧
(∃x)(m≤x≤j ∧ ~r<.A[x])

xii) perm(A^1,A^2,i,j) =
{A^1[x]|i≤x≤j} = {A^2[x]|i≤x≤j}

In order to prove that, given an operation BODY3, using it as in ii fulfills the requirements on SPLIT2, the auxilliary predicates are defined:-

xiii) pre-LOOP3(s^1) =
(first-it(A^1,m^1,n^1,i^1,j^1,r^1) ∨ (n^1<i^1 ∧ j^1<n^1)) ∧
(stoppers(A^1,m^1,i^1,j^1,n^1,r^1) ∨ j^1<i^1)

xiv) post-LOOP3(s^1,s^2) =
(f^2,N^2,m^2,n^2,r^2) = (f^1,N^1,m^1,n^1,r^1) ∧
undisturbed(A^1,A^2,m^1,n^1) ∧
perm(A^1,A^2,m^1,n^1) ∧
ordered(A^2(m^1:i^2-1),{r},A^2(j^2+1:n)) ∧
m^1≤i^2 ∧ j^2≤n^1

xv) term3(s^1) =
j^1<i^1 → 0,
T → j^1-i^1+1

The proof of:-
1. pre-SPLIT2<i:=m;j:=n; <u>while</u> i≤j <u>do</u> BODY3>post-SPLIT2

(cf 6.3(vii)) follows:-

For 4.2.2(vi) consider:-
2. pre-SPLIT2(s^1) ∧ post-INIT3(s^1,s^2)

From 2 and 6.3(viii):-
3. (∃x)(m^1≤x≤n^1 ∧ r^1 = A^1[x])

Also from 2 and v:-
4. (A^2,m^2,n^2,i^2,j^2,r^2) = (A^1,m^1,n^1,i^1,j^1,r^1)

Thus from x:-
5. first-it(A^2,m^2,n^2,i^2,j^2,r^2)

and from xi:-
6. stoppers(A^2,m^2,i^2,j^2,n^2,r^2)

Thus (see xiii):-
7. pre-LOOP3(s^2)

and from xiv:-

8. post-LOOP3(s^2,s^2)
is vacuously true.

For 4.2.2(vii):-
9. pre-LOOP3$(s^1) \wedge i^1 \leq j^1 \supset$ pre-BODY3(s^1)
is immediate from xiii, viii.

Consider for 4.2.2(viii):-
10. pre-BODY3$(s^1) \wedge$ post-BODY3(s^1,s^2)

The 10 and viii gives:-
11. first-it$(A^1,m^1,n^1,i^1,j^1,r^1) \vee (m^1<i^1 \wedge j^1<n^1)$
12. stoppers$(A^1,m^1,i^1,j^1,n^1,r^1)$

and from 10 and ix:-
13. perm(A^1,A^2,i^1,j^1)
14. ordered$(A^2(i^1:i^2-1),\{r\},A^2(j^2+1:j^1))$
15. progress(i^1,i^2,j^2,j^1)
16. $(\exists x)(i^1 \leq x \leq j^1 \wedge r^1 = A^1[x]) \supset i^1<i^2 \wedge j^2<j^1$

Consider the case:-
17. first-it$(A^1,m^1,n^1,i^1,j^1,r^1)$

From 16, 17, x gives:-
18. $i^1<i^2 \wedge j^2<j^1$
19. $(m^1,n^1) = (i^1,j^1)$

and from ix, 10:-
20. $m^2<i^2 \wedge j^2<n^2$

In the other case:-
21. ~first-it$(A^1,m^1,n^1,i^1,j^1,r^1)$

From 11 and 21:-
22. $m^1<i^1 \wedge j^1<n^1$

and from 15, 6.2(xi),ix, 10:-
23. $m^2<i^2 \wedge j^2<n^2$

which concludes the first clause of pre-LOOP3.

Consider the case:-
24. $i^2 \leq j^2$
25. undisturbed(A^1,A^2,i^1,j^1)

From 14, 24, 25 and a lemma from 6.1(xv) gives:-
26. stoppers$(A^2,m^2,i^2,j^2,n^2,r^2)$

The alternative case:-

27. $j^2<i^2$

is immediately the second clause of pre-LOOP3.

Thus:-
28. pre-LOOP3(s^2)

Now for 4.2.2(ix) consider:-
29. post-LOOP3(s^1,s^2) ∧ post-BODY3(s^2,s^3)

From 29 and xiv:-
30. $(f^2,N^2,m^2,n^2,r^2) = (f^1,N^1,m^1,n^1,r^1)$
31. undisturbed(A^1,A^2,m^1,n^1)
32. perm(A^1,A^2,m^1,n^1)
33. ordered$(A^2(m^1:i^2-1),\{r\},A^2(j^2+1:n^1))$
34. $m^1\leq i^2 \wedge j^2\leq n^1$

From 29 and ix:-
35. $(f^3,N^3,m^3,n^3,r^3) = (f^2,N^2,m^2,n^2,r^2)$
36. undisturbed(A^2,A^3,i^2,j^2)
37. perm(A^2,A^3,i^2,j^2)
38. ordered$(A^3(i^2:i^3-1),\{r\},A^3(j^3+1:j^2)$
39. progress(i^2,i^3,j^3,j^2)
40. $(\exists x)(i^2\leq x\leq j^2 \wedge r^2 = A^2[x]) \supset i^2<i^3 \wedge j^3<j^2$

Using 35 and 30:-
41. $(f^3,N^3,m^3,n^3,r^3) = (f^1,N^1,m^1,n^1,r^1)$

From 31, 34 and 36:-
42. undisturbed(A^1,A^3,m^1,n^1)

and from 32, 36, 37:-
43. perm(A^1,A^3,m^1,n^1)

Then from 38:-
44. ordered$(A^3(m^1:i^3-1),\{r\},A^3(j^3+1:n^1))$

and from 34, 39:-
45. $m^1\leq i^3 \wedge j^3\leq n^1$

Thus:-
46. post-LOOP3(s^1,s^3)
which is the required result from 29 by 4.2.2(ix).

Consider for 4.2.2(x):-
47. post-INIT3(s^1,s^2) ∧ post-LOOP3(s^2,s^3) ∧ pre-LOOP3(s^3) ∧ $j^3<i^3$

From 47 and v:-

48. $(A^2,f^2,N^2,m^2,n^2,r^2) = (A^1,f^1,N^1,m^1,n^1,r^1)$

and from 47 and xiv:-
49. $m^2<i^3 \wedge j^3<n^2$
50. $(f^3,N^3,m^3,n^3,r^3) = (f^2,N^2,m^2,n^2,r^2)$
51. $undisturbed(A^2,A^3,m^2,n^2)$
52. $perm(A^2,A^3,m^2,n^2)$
53. $ordered(A^3(m^2:i^3-1),\{r\},A^3(j^3+1:n^2))$

From 48 and 51:-
54. $undisturbed(A^1,A^3,m^1,n^1)$

and from 48, 51, 52,47 and 49:-
55. $gd\text{-}split(A^1(m^1:n^1),A^3(m^1:j^3),A^3(j^3+1:i^3-1),A^3(i^3:n^1))$

From 53, 48, 47:-
56. $ordered(A^3(m^1:j^3),A^3(j^3+1:i^3-1),A^3(i^3:n^1))$

Using 6.1(xii), 55, 56 give:-
57. $ordered\text{-}split(A^1(m^1:n^1),A^3(m^1,j^3),A^3(j^3+1:i^3-1),A^3(i^3:n^1))$

From 48, 49:-
58. $m^1<i^3 \wedge j^3<n^1$

From 48, 50:-
59. $(f^3,N^3,m^3,n^3) = (f^1,N^1,m^1,n^1)$

Thus from 47, 54, 57, 58, 59:-
60. $post\text{-}SPLIT2(s^1,s^3)$
which is the result required by 4.2.2(x).

The termination proofs, 4.2.2(xii) etc:-
61. $pre\text{-}LOOP3(s^1) \supset term3(s^1) \geq 0$
62. $term3(s^1) = 0 \equiv i^1>j^1$
63. $post\text{-}BODY3(s^1,s^2) \supset term3(s^2) < term3(s^1)$
all follow easily from their definitions.

6.5 Fourth Stage of Development

This is the last stage to be considered in detail. It shows how the BODY3 task of the last section can be broken into operations which locate high (low) elements, with respect to r, from the bottom (top) and finally an operation which exchanges the located elements providing the task is as yet incomplete.

i) BODY3 = FINDHI;FINDLO;SWOP
ii) FINDHI::S2
iii) $pre\text{-}HI<FINDHI>post\text{-}HI$
iv) $pre\text{-}HI(s^1) = (\exists x)(i^1 \leq x \leq n^1 \wedge \sim A^1[x] <. r^1)$

v) post-HI(s^1,s^2) =
(A^2,f^2,N^2,m^2,n^2,r^2,j^2) = (A^1,f^1,N^1,m^1,n^1,r^1,j^1) ∧
$i^1 \leq i^2$ ∧
$i^1 \leq x < i^2 \supset A^2[x] <. r$ ∧
$\sim(A^2[i^2] <. r)$

vi) FINDLO::S2

vii) pre-LO<FINDLO>post-LO

viii) pre-LO(s^1) = $(\exists x)(m^1 \leq x \leq j^1 \wedge \sim r^1 <. A^1[x])$

ix) post-LO(s^1,s^2) =
$s^2 = a(j,j',s^1)$
where:- $j' \leq j^1$ ∧
$j' < x \leq j^1 \supset r <. A^1[x]$ ∧
$\sim(r <. A^1[j'])$

x) SWOP::S2

xi) T<SWOP>post-SWOP

xii) post-SWOP(s^1,s^2) =
$j^1 < i^1 \supset s^2 = s^1$
$i^1 \leq j^1 \supset ((f^2,N^2,m^2,n^2,r^2) = (f^1,N^1,m^1,n^1,r^1)$ ∧
$x \neq i^1 \wedge x \neq j^1 \supset A^2[x] = A^1[x]$ ∧
$A^2[i^1] = A^1[j^1]$ ∧
$A^2[j^1] = A^1[i^1]$ ∧
$(i^2,j^2) = (i^1+1, j^1-1)$

In order to show (under the above assumptions) that the combination given in i fulfills the requirements on BODY3 (cf 6.4 (vii)) it is necessary to prove:-

1. pre-BODY3<FINDHI;FINDLO;SWOP>post-BODY3

Now for 4.2.1(iii), 4.2.1(iv):-

2. pre-BODY3(s^1) ⊃ pre-HI(s^1)
3. pre-BODY3(s^1) ∧ post-HI(s^1,s^2) ⊃ pre-LO(s^2)

are both immediate from 6.4 (viii), 6.4(xi) and iv, viii.

4. pre-BODY3(s^1) ∧ post-HI(s^1,s^2) ∧ post-LO(s^2,s^3) ⊃ F

is vacuously true.

Consider for 4.2.1(v):-

5. pre-BODY3(s^1) ∧ post-HI(s^1,s^2) ∧ post-LO(s^2,s^3) ∧
post-SWOP(s^3,s^4)

From 5 and 6.4(viii):-

6. $i^1 \leq j^1$
7. stoppers(A^1,m^1,n^1,j^1,n^1,r^1)

From 5 and v:-

8. (A^2,f^2,N^2,m^2,n^2,j^2) = (A^1,f^1,N^1,m^1,n^1,j^1)
9. $i^1 \leq i^2$
10. $i^1 \leq x < i^2 \supset A^2[x] <. r \wedge \sim(A^2[i^2] <. r)$

From 5 and ix and 8:-
11. $(A^3,f^3,N^3,m^3,n^3,i^3) = (A^1,f^1,N^1,m^1,n^1,i^2)$
12. $j^3 \leq j^1$
13. $j^3 < x \leq j^1 \supset r <. A^3[x] \wedge \sim(r <. A^3[j^3])$

Thus from:-
14. ordered$(A^3(i^1:i^3-1),\{r\},A^3(j^3+1:j^1))$

Now consider the case:-
15. $j^3 < i^3$

Then 5 and xii give:-
16. $s^4 = s^3$

Thus 16, 11, 8 give:-
17. $(A^4,f^4,N^4,m^4,n^4,i^4,j^4) = (A^1,f^1,N^1,m^1,n^1,i^2,j^3)$

From 10, 13 and 6.1(xv):-
18. ordered$(A^4(i^1:i^4-1),\{r\},A^4(j^4+1:j^1))$

Then 6, 15, 9, 12 give:-
19. progress(i^1,i^4,j^4,j^1)

Thus, since the last clause is vacuously true:-
20. post-BODY3(s^1,s^4)

In the other case:-
21. $i^3 \leq j^3$

5, xii and 11 give:-
22. $(f^4,N^4,m^4,n^4) = (f^1,N^1,m^1,n^1)$
23. $x \neq i^3 \wedge x \neq j^3 \supset A^4[x] = A^1(x)$
24. $A^4[i^3] = A^1[j^3]$
25. $A^4[j^3] = A^1[i^3]$
26. $(i^4,j^4) = (i^2+1,j^3-1)$

Then 9, 12, 8, 17 with 24-26:-
27. undisturbed(A^1,A^4,i^1,j^1)

From 17,24,25,26,10,13:-
28. ordered$(A^4(i^1:i^4-1),\{r\},A^4(j^4+1:j^1))$

From 9,12,26:-
29. progress(i^1,i^4,j^4,j^1)

and:-
30. $i^1 < i^4 \wedge j^4 < j^1$

Thus:-

31. post-BODY3(s^1,s^4)

6.6 Collection of the Algorithm

This section collects together the parts of the algorithm so far developed. The process is simply one of collecting the expansions of the non-basic operations into the place they were used. (Strictly JOIN, FINDHI and FINDLO should be further expanded). Inspection of 6.2(ii),6.3(ii),6.4(ii), 6.5(i), 6.5(v),6.5(ix),6.5(xii),6.3(xii) gives:

FIND=

```
m:=1;
n:=N;
while m≠n do
        (r:=A[f];
         i:=m;
         j:=n;
         while i≤j do
               (while A[i] <. r do i:=i+1;
                while r <. A[j] do j:=j-1;
                if i≤j then
                     (w:= A[i];
                      A[i] := A[j];
                      A[j] := w;
                      i:=i+1;
                      j:=j-1 ) );
         if f≤j then n:=j
         else if i≤f then m:=i
         else m:=n:=f )
```

7. EVOLUTION OF DATA

This section resumes the general outline of Formal Development, begun in section 5, by considering how the data on which operations work can also evolve during a development. An example illustrating the idea is given in section 8.

Consider how abstract data objects might arise. It might well be possible to give the specification of the task to be performed (the "what") using far more abstract objects (e.g. sets) than exist in most programming languages. It would certainly be a mistake to require more detail than is necessary to describe the task since the freedom thus lost might have led to a more natural or efficient solution. For any reasonably sized task the development of the solution (the "how") will also introduce auxiliary objects which are not part of the specification. It is frequently the case that the role these objects are to perform (e.g. mappings) is far easier to describe than their eventual machine representation (e.g. pointers etc).

The main problem in the construction and subsequent proofs of whole algorithms was claimed to be the level of detail required. The idea of bringing in at each stage of development only those properties required is of paramount importance in avoiding this problem. It is in fact the abstraction of data types which distinguishes "FD" from top-down development where interfaces are specified in detail, often too early.

The process then is to use as abstract a form of the data as possible in order to develop and justify the current part of the algorithm. When this has been done the data can be mapped onto a more detailed structure and the correctness of this mapping proven. This process will be moving towards, and eventually terminate at, data objects of the machine/language to be used. In general, the more detailed data representation will bring with it new implementation problems requiring further development.

A good example of the way in which this method breaks up the development is seen in reference 2. The task considered is a table driven parser and the specification is given in terms of (a relation defined in terms of) whether rules of particular forms are members of the set of rules comprising the given grammar. The overall algorithm is stated in terms of "state-sets" which record the status of all possible top-down parses. Only after the main operations on state sets have been shown to create sets with the required properties, is the problem of mapping these sets onto lists considered and appropriate development of the operation made. Having now determined in what way the algorithm uses the grammar it is possible to consider the efficiency of

commonly used operations in designing a mapping for the "set of rules" onto data types of PL/I.

The evolution of data types should always be from the abstract to the more concrete. Thus it is possible to think of the mapping onto a new data type as the addition of properties. For example, suppose we have a set of elements:

$$M \subset \beta(U)$$

then mapping this set onto a portion of an array can be thought of as adding an addressing mechanism such that:

$$(m \leq i \leq n \wedge v = A[i]) \equiv v \in M$$

(Notice that without any explicit mention section 6 has already used the process of adding components to its state which is a simple form of mapping. The reader will see below that such a simple mapping did not require explicit justification.)

The remainder of this section gives the formal properties required to prove the correctness of a "mapping" stage of Formal Development. As in section 4 the rules presented do not attempt to be the most general but are those required for the planned example.

Suppose some stage of development uses:

i) OPd :: D
ii) pred<OPd>postd

That is:-
iii) $pred(d^1) \supset (\exists d^2)(d^1[OPd]d^2)$
iv) $pred(d^1) \wedge d^1[OPd]d^2 \supset postd(d^1,d^2)$

Then a new operation on a new domain:-
v) OPe :: E
vi) pree<OPe>poste

or:-
vii) $pree(e^1) \supset (\exists e^2)(e^1[OPe]e^2)$
viii) $pree(e^1) \wedge e^1[OPe]e^2 \supset poste(e^1,e^2)$

is an acceptable model of OPd providing firstly a mapping relation:-
ix) map : D x E → {T,F}

can be found. It is often the case, and is so in section 8 that the relation is total in the direction of yielding the less

abstract object:-
x) $(\exists e)(map(d,e))$

and partial in the other direction:-
xi) $mappable(e) \supset (\exists d)(map(d,e))$

Secondly, all elements created in the new domain are mappable back into the old:-
xii) $pred(d^1) \wedge map(d^1,e^1) \wedge poste(e^1,e^2) \supset mappable(e^2)$

Thirdly, any element of the domain of the old function is mapped into an element of the domain of the new function:-
xiii) $pred(d^1) \wedge map(d^1,e^1) \supset pree(e^1)$

Lastly, that under the mapping the same function is computed:-
xiv) $map(d^1,e^1) \wedge poste(e^1,e^2) \wedge map(d^2,e^2) \supset postd(d^1,d^2)$

In order to show that the above properties are adequate to ensure that the same function is computed, it must be proven that:-

1. $pred(d^1) \supset (\exists d^2)(map(d^1,e^1) \wedge poste(e^1,e^2) \wedge map(d^2,e^2))$
2. $pred(d^1) \wedge map(d^1,e^1) \wedge poste(e^1,e^2) \wedge map(d^2,e^2) \supset postd(d^1,d^2)$

Assume:-
3. $pred(d_1)$

Then from x:-
4. $(\exists e)(map(d_1,e))$ letting e_1 be such:-
5. $map(d_1,e_1)$

Then from xiii:-
6. $pree(e_1)$ 3,5

Using vii gives:-
7. $(\exists e^2)(e_1[OPe]e^2)$ 6

Letting e_2 be such:-
8. $e_1[OPe]e_2$

Then from viii:-
9. $poste(e_1,e_2)$ 6,8

and from xii:-
10. $mappable(e_2)$ 3,5,9

So from xi:-
11. $(\exists d^2)(map(d^2,e_2))$ 10

Therefore:-

12. $(\exists d^2)(map(d_1,e_1) \wedge poste(e_1,e_2) \wedge map(d^2,e_2))$ 4,7,11

and letting d_2 be an instance in 11:-

13. $map(d_2,e_2)$

Finally from xiv:-

14. $postd(d_1,d_2)$ 5,9,13

The above discussion is confined to single operations and implies a need for input/output functions to perform the mappings. It is often the case that at some point in a development there are a number of operations working on the same state and they are all to be mapped onto the same new state. In this case it should be clear that, providing they are combined in a proper way, the combination of the mapped operations avoids all of the intermediate mappings to and from the old domain.

8. EXAMPLE OF DATA EVALUATION

The development shown in section 6 used, basically, the same data structure throughout. This section illustrates the process described in section 7 by reconsidering the same example. Unfortunately the FIND algorithm is so closely linked to arrays that it is far from ideal for the purposes and the reader is asked to consider the method rather than the example.

The first step is to restate the specification in terms of sets in section 8.1. The development stage given in section 8.2 is included for completeness and comparison with section 6.2, the main interest is in the mapping stage given in section 8.3.

8.1 Specification

The state of the required program will be:-

i) $T = (\langle L : \text{is-U-set}\rangle, \langle M : \text{is-U-set}\rangle, \langle H : \text{is-U-set}, \langle F : \text{is-int}\rangle)$

That is the "low", "medium" and "high" state components are each capable of holding finite sets of U (the given) elements.

Then the properties required of:-

ii) FIND :: T

are:-

iii) pre<FIND>post

where the assumptions about the input state are:-

iv) $pre(t) = card(M) \geq 1 \wedge 1 \leq F \leq card(M)$

The function card yields the number of elements of a set.

Given such states, the three sets should contain in the output state a split of the elements in the M component of the input state such that F-1 elements are in L, one element in M and the rest in H:

v) $post(t^1,t^2) = ordered\text{-}split(M^1,L^2,M^2,H^2) \wedge card(L^2) = F^1-1 \wedge card(M^2) = 1$

ordered-split etc are as in 6.1(xii) to 6.1(xv).

8.2 First Stage of Development

As in section 6.2, the first step is to show how the required task can be accomplished by using an operation, which breaks off high and low elements, in an initialised while loop.

i) FIND = L:=Φ; H:=Φ; while card(M)≠1 do BODYS

The requirements on the initialisation are:-
ii) (L:=Φ; H:=Φ) :: T
iii) T<L:=Φ; H:=Φ>post-INITS
iv) post-INITS(t^1,t^2) = t^2 = a(H,Φ,a(L,Φ,t^1))

The requirements on the body are:-
v) BODYS::T
vi) pre-BODYS<BODYS>post-BODYS
vii) pre-BODYS(t^1) =
intermed(F^1,card(L^1),card(M^1)) ∧
card(M^1)>1 ∧
pdis(L^1,M^1,H^1)
viii) post-BODYS(t^1,t^2) =
intermed(F^2,card(L^2),card(M^2)) ∧
F^2 = F^1 ∧
(L^2,M^2,H^2) = (L^1 ul,m,H^1 uh) ∧
ordered-split(M^1,l,m,h) ∧
l≤card(m)<card(M^1)

Where:-
ix) intermed(F,lel,mel) =
lel<F≤lel+mel

To prove the correctness of i, define:-
x) pre-LOOPS(t^1) = intermed(F^1,card(L^1),card(M^1)) ∧
card(M^1)≥1 ∧
pdis(L^1,M^1,H^1)
xi) post-LOOPS(t^1,t^2) = F^2 = F^1 ∧
ordered-split(M^1,L^2,M^2,H^2)
xii) termS(t^1) = card(M^1)-1

The proof of (cf 8.1(iii)):-
1. pre<L:=Φ; H:=Φ; while card M≠1 do BODYS>post
can be given according to section 4.2.2 as follows:-

For 4.2.2(vi) first part:-
2. pre(t^1) ∧ post-INITS(t^1,t^2) ⊃ pre-LOOPS(t^2)
is, given that card(Φ) = 0, immediate from 8.1(iv),iv,x.

For 4.2.2(vi) second part:-
3. pre(t^1) ∧ post-INITS(t^1,t^2) ⊃ post-LOOPS(t^2,t^2)
is seen to be vacuously true from iv, xi.

For 4.2.2(vii):-
4. pre-LOOPS(t^1) ∧ card(M^1)≠1 ⊃ pre-BODYS(t^1)
is immediate from x, vii.

For 4.2.2(viii):-
5. pre-BODYS(t^1) ∧ post-BODYS(t^1,t^2) ⊃ pre-LOOPS(t^2)
follows from vii, viii, x and a lemma on ordered-split.

For 4.2.2(ix) consider:-
6. post-LOOPS(t^1,t^2) ∧ post-BODYS(t^2,t^3)

Using xi:-
7. $F^2 = F^1$
8. ordered-split(M^1,L^2,M^2,H^2)

and using viii:-
9. $F^3 = F^2$
10. (L^3,M^3,H^3) = (L^2 ∪ l,m,H^2 ∪ h)
11. ordered-split(M^2,l,m,h)

Thus from 7,9:-
12. $F^3 = F^1$

By a lemma an ordered-split, 8, 10, 11, give:-
13. ordered-split(M^1,L^3,M^3,H^3)

Observing xi, 12 and 13 give:-
14. post-LOOPS(t^1,t^3)

For 4.2.2(x) consider:-
15. post-INITS(t^1,t^2) ∧ post-LOOPS(t^2,t^3) ∧ pre-LOOPS(t^3) ∧ card(M^3) = 1

Using iv:-
16. $M^2=M^1$ ∧ $F^2=F^1$

Using xi:-
17. $F^3=F^2$
18. ordered-split(M^2,L^3,M^3,H^3)

Using x and ix:-
19. card(L^3) $<F^3 \leq$ card(L^3) + card(M^3)

Since card(M^3)=1 (see 15):-
20. card(L^3) = F^3-1

From 17, 16:-
21. card(L^3) = F^1-1

From 18, 16:-
22. ordered-split(M^1,L^3,M^3,H^3)

Observing 8.1(v),22,21 and 15 give:-
23. post(t^1,t^3)

For 4.2.2(xi):-
24. pre-LOOPS(t^1) ⊃ termS(t^1)≥0
is immediate from x, xii.

For 4.2.2(xii):-
25. termS(t^1) = 0 ≡ ~(card(M)≠1)
is immediate from xii.

For 4.2.2(xiii):-
26. post-BODYS(t^1,t^2) ⊃ termS(t^2)<termS(t^1)
is immediate from xii,viii.

8.3 Mapping Stage

The purpose of this stage is to map the operations of section 8.2, which work on state T, to operations which work on state S1 (see section 6.2). Of the three operations used in 8.2(i) the initialisation is omitted because of the special problems, considered in section 9, of input routines.

The relation:-
i) map: TxS1 → {T,F}

is used to define the mapping:-
ii) map(t,s) = δ(A,1,m-1) ∧ δ(A,m,n) ∧ δ(A,n+1,N) ∧
(L,M,H) = (A(1:m-1),A(m:n),A(n+1:N)) ∧
F=f

Where the requirement that an array segment has no duplicates is:-
iii) δ(A,x,y) = x≤i,j≤y ∧ i≠j ⊃ A[i] ≠ A[j]

Considering section 7, notice that for 7(x):-
1. (∃s)(map(t,s))
is true since m can be set to card(L)+1
is true since n can be set to card(L)+card(M)
is true since N can be set to card(L)+card(M)+card(H)
and A can be set up without duplicates in the sections.

For 7(xi):-
2. δ(A,1,m-1) ∧ δ(A,m,n) ∧ δ(A,n+1,N) ⊃ (∃t)(map(t,s))
is true since it simply defines L,M and H.

Now the operation:-
<card(M)≠1>

Can be modelled by:-
iv) <m≠n>

Since the original was a total predicate causing no state changes, 7(xii) and 7(xiii) are irrelevant.

For 7(xiv):-
3. $map(t^1,s^1) \supset (card(M^1)\neq 1 \equiv m^1\neq n^1)$
which is immediate from ii.

Now the operation:-
pre-BODYS<BODYS>post-BODYS

can be modelled by:-
v) pre-BODY<BODY>post-BODY
vi) $pre\text{-}BODY(s^1) = m^1\leq f^1\leq n^1 \wedge m^1\neq n^1 \wedge \delta(A^1,1,N^1)$
vii) $post\text{-}BODY(s^1,s^2) = m^2\leq f^2\leq n^2 \wedge (f^2,N^2) = (f^1,N^1) \wedge undisturbed(A^1,A^2,m^1,n^1) \wedge ordered\text{-}split(A^1(m^1:n^1),A^2(m^1:n^2-1),A^2(m^2:n^2),A^2(n^2+1:n^1)) \wedge progress(m^1,m^2,n^2,n^1) \wedge \delta(A^2,1,N^2)$
which is proven as follows.

For 7(xii):-
4. $post\text{-}BODY(s^1,s^2) \supset \delta(A^2,1,N^2)$
is immediate from vii and sufficient from 2.

For 7(xiii) consider:-
5. $pre\text{-}BODYS(t^1) \wedge map(t^1,s^1)$

Using 8.2(vii), 8.2(ix):-
6. $card(L^1)<F^1\leq card(L^1)+card(M^1)$
7. $card(M^1)>1$
8. $pdis(L^1,M^1,H^1)$

Using ii:-
9. $\delta(A^1,1,m^1-1) \wedge \delta(A^1,m^1,n^1) \wedge \delta(A^1,n^1+1,N^1)$
10. $(A^1(1:m^1-1),A^1(m^1:n^1),A^1(n^1+1:N^1)) = (L^1,M^1,H^1)$
11. $f^1=F^1$

From 8, 10, 9:-
12. $\delta(A^1,1,N^1)$

From 9,6,10,11:-
13. $m^1 \leq f^1 \leq n^1$

From 7,9,10:-
14. $m^1 \neq n^1$

Observing vi, 13, 14 and 12 give:-
15. pre-BODY(s^1)

For 7(xiv) consider:-
16. map(t^1,s^1) ∧ post-BODY(s^1,s^2) ∧ map(t^2,s^2)

Using ii:-
17. ($A^1(1:m^1-1)$,$A^1(m^1:n^1)$,$A^1(n^1+1:N^1)$) = (L^1,M^1,H^1)
18. $F^1=f^1$

Using vii:-
19. $m^2 \leq f^2 \leq n^2$
20. (f^2,N^2) = (f^1,N^1)
21. undisturbed(A^1,A^2,m^1,n^1)
22. ordered-split($A^1(m^1:n^1)$,$A^2(m^1:m^2-1)$,$A^2(m^2:n^2)$,$A^2(n^2+1:n^1)$)
23. progress(m^1,m^2,n^2,n^1)

Again using ii:-
24. (L^2,M^2,H^2) = ($A^2(1:m^2-1)$,$A^2(m^2:n^2)$,$A^2(n^2:N^2)$)
25. $F^2=f^2$

From 24, 19, 25 and 8.2(ix):-
26. intermed(F^2,card(L^2),card(M^2))

From 25, 20, 18:-
27. $F^2=F^1$

Let:-
28. (l,m,h) = ($A^2(m^1:m^2-1)$,$A^2(m^2:n^2)$,$A^2(n^2+1:n^1)$)

Then 24, 21, 17, 23 give:-
29. (L^2,M^2,H^2) = (L^1 ul,m,H^1 uh)

and 17, 22, 28 give:-
30. ordered-split(M^1,l,m,h)

and 19, 28 give:-
31. 1≤card(m)

and 23, 28, 17 give:-
32. card(m) <card(M^1)

Observing 8.2(viii) 26, 27, 29, 30, 31, 32 give:-
33. post-BODYS(t^1,t^2)

This concludes the proof of the correctness of the mapping. Notice the similarity between the conditions v-vii and 6.2(vi) - 6.2(ix): in fact the additional property of not generating duplicates is, of course, true of the algorithm developped in section 6. Thus the development via sets has been mapped onto a stage where it could be completed using arrays.

It is worth stressing that this is not a good example of the potential advantages of using abstract data types - a better example could be created by rewriting reference 2 in the notation of the current paper.

9. POSSIBLE EXTENSIONS

The paper has so far discussed only some isolated details of programming languages. It is beyond the scope of the current paper to give a formal model for the whole of even a very simple language. The aim of this section is to indicate that the material of section 4 can be extended to cover more complicated programming constructs and to suggest what form the extension might take. The material that follows is thus rather sketchy.

Section 3 contained a way of dividing programming constructs:

a) Those which change values in store are discussed in section 9.3.
b) Those which change the structure of store are discussed in section 9.4.
c) Those which serve to combine constructs are discussed in section 9.2.

9.1 On the Formal System

Before considering how to extend the system in the indicated areas it is worth explaining the reason behind choosing the style used to present those rules given so far in section 4.

As an alternative to the rules of section 4.2.1 it would have been possible to, for example, identify "pre" with "pre-1" and to use general rules like:

Given:

$$pre\langle OP\rangle post$$
$$\text{strong-pre}(\sigma^1) \supset pre(\sigma^1)$$
$$post(\sigma^1,\sigma^2) \supset \text{weak-post}(\sigma^1,\sigma^2)$$

Then it is valid to conclude:-

$$\text{strong-pre}\langle OP\rangle post$$

and:-

$$pre\langle OP\rangle \text{weak-post}$$

This idea is in fact used in some of the proofs based on ref 6. The choice against it here has been deliberate because the combination rules of section 4 appear to fit better with their use in formal development: this direction would be followed in any future extensions.

9.2 Combination Constructs

This section considers matters relating to programming constructs which combine other constructs.

9.2.1 Input/Output

The overall specification of a task will often be that of a function: it must accept arguments and produce an appropriate result rather than cause some change to a state. Of course, the task may read the arguments into its own storage, manipulate that state, then output a result. The necessity for input/output can also arise, as has been mentioned, where a mapping stage of development is used.

The approach to input/output is illustrated by giving the types only of the first stage of development from a definition for FIND of say:

$$FIND: \beta(U) \times I \rightarrow \beta(U) \times \beta(U) \times \beta(U)$$

Then given T as in section 8.2 the stage might use:-

$IN::T:\beta(U) \times I \rightarrow \Phi$
$FIND\beta::T$
$OUT::T:\Phi \rightarrow \beta(U) \times \beta(U) \times \beta(U)$

9.2.2 Side-Effects in Predicates

The introduction of function references brings with it the possibility of side-effects. It is easy to extend the conditional rules (cf section 3.2.2) to cover this:

$$\sigma^1[p]\sigma^2, T \supset \sigma^1[\underline{\text{if}}\ p\ \underline{\text{then}}\ OP1\ \underline{\text{else}}\ OP2]\sigma^3 \equiv \sigma^2[OP1]\sigma^3 \text{ etc.}$$

Similar extensions are possible for while.

9.2.3 Procedures

The extended operation of section 3.3 can indicate the types of procedure calls:-

$PROC\text{-}CALL::\Sigma:A \rightarrow \Phi$

or function references:-

$FN\text{-}REF::\Sigma:A \rightarrow \Omega$

9.2.4 Goto

The concept of statements appointing their own successors fits least naturally into the proposed system. One approach would be

to add an abnormal return component to the state (cf. ref 12) which would be tested by the composition rule.

9.2.b Parallelism

Bekic has pointed out in ref 5 that the effect of parallel combination of two operations cannot be determined solely from their input/output relations. The approach taken there is to have enough information to determine the effect of arbitrary combination. Any attempt to extend the current system would be made by disciplining co-operation by giving extra predicates which all intermediate states must satisfy.

9.3 Operations Changing Values

The need for rules to deduce properties about those constructs which change the values in states has been largely side-stepped above although it is this property which characterises operations. The main reason has been the emphasis on decomposition of problems in formal development.

The obvious example of a statement of this sort is assignment. In a language with no sharing (e.g. by name argument passing) or function references, the property might be:

$$pre\langle v \leftarrow e \rangle post$$
$$pre(\sigma^1) = (\exists val)(val = \Psi(e, \sigma^1))$$
$$post(\sigma^1, \sigma^2) = \sigma^2 = a(v, \Psi(e, \sigma^1), \sigma^1)$$

Function references and their ability to cause side-effects could be handled with an expansion of Ψ providing the order of expression evaluation is defined. If the order of expression evaluation is not defined by the language, the extension is complicated by non-determinism.

Input statements would also be in this class.

9.4 Operations which change State Structure

Programming languages have blocks and procedures which cause new states to be defined. In fact it is precisely the declarations which define the states.

To indicate how this problem could be approached, consider a language in which blocks introduce a single local variable. Suppose the context in which the block is to be used is given by:

$$(\underline{\text{begin}}\ \underline{\text{var}}\ v;\ OP\ \underline{\text{end}}) :: \Sigma$$

Then:

$OP :: \Sigma$ extended by v

and using "?" as the uninitialised value:-

$$a(v,?,\sigma^1)[OP]\sigma^2 \supset \sigma^1[\underline{\text{begin}}\ \underline{\text{var}}\ v;\ OP\ \underline{\text{end}}]a(v,c(v,\sigma^1),\sigma^2)$$

Procedures could be approached in a similar way but allocate would be more difficult because of the need to handle pointers.

REFERENCES

1. J McCarthy "A Formal Description of a subset of ALGOL" in "Formal Language Description Languages" (ed. T B Steel) 1964 Baden Conference Proceedings.

2. C B Jones "Formal Development of Correct Algorithms: An Example Based on Earley's Recogniser" in SIGPLAN NOTICES Vol 7 No 1, January 1972

3. P Lucas and K Walk "On the Formal Description of PL/I" Annual Review in Automatic Programming Vol 6 Part 3 1969.

4. J W de Bakker and D Scott "A Theory of Programs". Notes on a seminar at IBM Laboratory Vienna August 1969.

5. H Bekic "Towards a Mathematical Theory of Processes" IBM Laboratory Vienna report TR 12.125, December 1971

6. C A R Hoare "The Axiomatic Basis for Computer Programming" in CACM October 1969.

7. C A R Hoare "Proof of a Program: FIND" in CACM January 1971

8. R W Floyd "Assigning Meanings to Programs" in Proc. Symposium in Applied Mathematics. "Mathematical Aspects of Computer Science" AMS, 1967.

9. E W Dijkstra "Notes on Structured Programming" EWD 249, August 1969.

10. N Wirth "Program Development by Stepwise Refinement" in CACM April 1971.

11. E W Dijkstra "A Short Introduction to the Art of Programming" EWD 316 August 1971

12. C D Allen, D N Chapman, C B Jones "A Formal Definition of Algol 60" IBM Laboratory Hursley report TR 12.105, August 1972.

13. M Foley and C A R Hoare "Proof of a Recursive Program: Quicksort" in CJ November 1971.

14. S C Kleene "Mathematical Logic" John Wiley & Sons 1967.