ALGORITHM VERIFICATION

BY

W. D. Maurer

Memorandum No. ERL-M290

January 1971

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ALGORITHM VERIFICATION

Ward Douglas Maurer

Introduction

Any experienced programmer is keenly aware of the seemingly infinite variety of errors that can creep into his programs. Many programmers despair of our ever being able to classify errors in programs by type. Now, however, it appears that algorithm verification,-- the mathematical proof that a given program has no errors in it -- is finally within our grasp. It is our purpose here to discuss the general problem of verifying algorithms. Our methods will be quite general purpose, applying to programs written in algebraic languages, business languages, assembly languages, and higher-level languages. At the conclusion of this paper, we give, as examples, verifications of three FORTRAN programs. In a companion paper [12], we discuss applications to FORTRAN, ALGOL, LISP, SNOBOL, BASIC, NELIAC, CDC 6400 assembly language, and IBM 1130 assembly language. Included in [12] are verifications by students of table searching and sorting algorithms, list manipulation, prime number calculation, Gauss and Gauss-Seidel methods, tic-tac-toe and knight's tour programs, conversion from prefix to infix notation, and applications such as amplifier cost minimization and computer dating.

Our treatment will be rigorously mathematical, beginning with the specification of programs, expressions, statements, instructions, and the like as mathematical objects much like groups or vector spaces. Our fundamental notions are the cartesian product set, or "p-set" for short, and the function on a p-set, or from one p-set to another.

Using this as a basis, we shall define algorithms, we shall define what it means for an algorithm to be correct, and we shall then state and prove a fundamental verification theorem upon which all our later work is based. This theorem allows us to translate our intuitive notions of why a program works into "condition structures," which are collections of assertions about the variables of our program at any given stage in the computation. Provided that these structures satisfy certain fundamental requirements, our verification theorem asserts that our given algorithm is correct.

Many of our results have been independently studied by others. The fundamental theorem on "partial correctness" of an algorithm is due to Floyd [3]; a complementary result on termination of algorithms first appeared in a paper by Manna [10]. Neither of these papers used the idea of a cartesian product set as basic to programming. On the other hand, there have been a large number of papers, not primarily concerned with verification of algorithms, in which this concept has been studied. Elements of a p-set are called "state vectors" by McCarthy [13], "content functions" by Elgot and Robinson [2], and simply "states" in an early paper of the author [11]. The "region of influence," or set of variables altered by a given statement or instruction, was first rigorously defined in [2]. The set of variables used by a statement or instruction was first defined in [11]; in this paper we also stressed the fact that a cartesian product of arbitrary sets (of at least two elements), rather than of n copies of the same domain, is the most reasonable model for programming, and also introduced an axiom (later called the "finite support property") under which the number of variables in a program is allowed to be infinite. This allows us to include every square on every input and output tape as part of the index set over which the cartesian product is taken, and thus allows us to include input-output in our model, answering a question posed in [13].

To our knowledge, the only work which has so far been done on verification using a cartesian product model is contained in the doctoral dissertations of King [5] and Good [4]. Both of these dissertations are concerned with program verification programs, and both use McCarthy's term, "state vectors." In the present paper we extend the concepts of "input region" and "output region" introduced in [11] (renaming them <u>effective domain</u> and <u>effective range</u>, respectively), which allows us to achieve certain simplifications. In particular, we shall define a general concept of <u>memory extension</u> of a function from one p-set to another, with respect to which the fundamental character of assignment statements becomes clear. Expressions, terms, factors, and the like are viewed as functions from a set of state vectors into the real numbers, the integers, or in general, a set determined uniquely by the type of the given expression. A general combinatorial operation on such functions, which we call the "star-extension," allows functions for terms, for example, to be built up from the functions for the factors contained in the given terms, and is also quite generally applicable in programming languages in the construction of complex state vector functions from simpler ones in natural ways. In [12] we define semantic extensions of BNF, using and extending a method first suggested by Knuth [7]. This provides a method of defining the semantics of a language in terms of its effect on the relevant p-set. This is done directly, rather than indirectly as in the so-called "Vienna method" (see, e. g., [9]), where PL-I programs are defined as abstract trees.

Algorithm verification is, of course, subject to certain well-known limitations. Strictly speaking, a program has not been verified until it has been checked for the effects of roundoff error, truncation error, arithmetic overflow, and errors arising from approximations. In our mathematical model, this corresponds to the fact that the sets over which the cartesian product is taken are not the real numbers with real arithmetic operations, but rather the floating point numbers for a specific computer, with floating point operations. It is possible for the same FORTRAN program to give correct results on one computer and not on another; thus our semantic specifications and our verifications must often be with respect to a specific implementation of a language, and not merely with respect to a specific language. A verified program may fail to work properly due to errors in the compiler or interpreter; this suggests that compilers and interpreters be the first large programs to be themselves verified, and it is hoped that our methods will aid those seeking to work in this direction. A verified program may even fail due to hardware errors; the verification of hardware is beyond the scope of this paper, although it is interesting that a specification of the performance expected of the hardware may be obtained by using our methods. Finally, a large number of well working programs cannot be verified because, stated simply, they do not always work properly; they are subject to the rule of "garbage in, garbage out," and do not, for example, check their input for all possible errors. In such cases it can merely be pointed out that these possibilities exist; or, alternatively, it can merely be proved that *if* the input satisfies certain criteria of reasonableness then the given program is correct.

Despite these considerations, however, algorithm verification can become extremely useful. Verification of a program could very easily take the place of informal documentation. This would allow extensions to be made to a program without fear; one merely extends the verification. There are many practical situations in which it is difficult or expensive to check that a given computer program has worked properly, such as in consumer credit or vote counting. Under these conditions, some sort of verification of programs is necessary, and the most certain verification will be a mathematical one.

The use of the computer in the verification process is a subject that is bound to assume increasing importance, particularly since program verifications are so long and tedious. One has not proved that a computer program has no errors in it if there are errors in the proof. We emphasize, however, that verifying an algorithm should initially be regarded as a human activity, just as programming is. That is, one writes a program in a language and then asks the computer to compile it before executing, and in the same way one writes a proof in a verification language and then asks the computer to check its logic. The construction of proofs of programs by computer is akin to mathematical theorem proving, and may here by regarded as a second stage in the subject -- somewhat like the construction by computer of programs themselves, as the output of a general problem-solving program. A start in this direction has been made by King [5].

Program verification programs have other limitations, besides the fact that, in general, they cannot be expected to generate their own proofs. One of these has to do with the use of mathematical facts in proving programs correct. When writing a sine routine, for example, the first thing we normally do is to divide the argument by 2π and take the remainder, using the identity $\sin (x+2\pi) = \sin x$. This identity will be used in the proof that the given sine routine actually calculates the sine of x; but it will not itself be a consequence of any algorithmic arguments involving the graphical structure of this routine. Similarly, if an algorithm calculates an element of a convergent sequence and then loops back to calculate the next element, then the fact that the sequence converges may be used in proving that the algorithm works properly. This fact must, however, be supplied as input to the program verification program, unless this is also a mathematical theorem prover.

Recursion, rather surprisingly, is not fundamental to our development of this subject. It is true that we will sometimes be required to verify a recursive program; but for the verification of a program in a language such as FORTRAN, where recursion is not allowed, recursive methods are not necessary and are not used here. Also, we shall not be concerned with the formal structure of the predicate calculus, or with sequential machines or Turing machines. Therefore, the prerequisites for understanding this treatment of the subject are nothing more than a knowledge of programming techniques and a small amount of basic set theory, which we shall now review. (For the understanding of our programming language semantic specifications in [12], it is necessary to be acquainted with BNF and with the elements of context-free languages.)

The notion of a set and of a function from one set to another are taken as primitive. If f is any function, X is the domain of f, and Y contains the range of f, we speak of the total function $f: X \rightarrow Y$. If X merely contains the domain of f, we speak of the partial function $f: X \rightarrow Y$. If $f: X \rightarrow Y$ and $X' \subseteq X$, we write $f|X'$ for the function $g: X' \rightarrow Y$ defined by $g(x) = f(x)$ for all $x \in X'$ (for which f is defined, if it is partial); this is the <u>restriction</u> of f to X'. If $f: X \rightarrow Y$ and $g: Y \rightarrow Z$, then the function $h: X \rightarrow Z$ defined by $h(x) = g(f(x))$ is the <u>composition</u> of f and g; here all these functions may be either total or partial. We write $h = f \bullet g$ in this case (rather than $h = g \bullet f$, as followed by some authors); thus $f \bullet g$ means, intuitively, "first apply f, then apply g."

The cartesian product $X \times Y$ of any two sets X and Y is defined as the set of all maps $f: \{1, 2\} \rightarrow X \cup Y$ such that $f(1) \in X$ and $f(2) \in Y$; it may be interpreted as the set of all ordered pairs (x, y) with $x \in X$ and $y \in Y$. (We may equivalently use $\{0, 1\}$ or $\{$<u>true</u>, <u>false</u>$\}$ in place of $\{1, 2\}$). The cartesian product $X_1 \times \ldots \times X_n$ of any finite collection of sets X_i is defined as the set of all maps $f: \{1, \ldots, n\} \rightarrow X_1 \times \ldots \times X_n$ such that $f(i) \in X_i$, $1 \leq i \leq n$; it may be interpreted as the set of all ordered n-tuples $(x_1, \ldots, x_n)$ with $x_i \in X_i$, $1 \leq i \leq n$. Where a collection of sets X_i is indexed by a set I, so that there is one set X_i for each $i \in I$, the set $\bigcup_{i \in I} X_i$ is the set of all x such that $x \in X_i$ for some $i \in I$, and the set $\prod_{i \in I} X_i$ is the set of all maps $f: I \rightarrow \bigcup_{i \in I} X_i$ such that $f(i) \in X_i$ for all $i \in I$; this may be interpreted as the set of all n-tuples or all "infinituples" $\{x_i\}$, with $x_i \in X_i$ for each $i \in I$, depending on whether I is finite or infinite. (McCarthy writes <u>c</u>(<u>var</u>, ξ) ([13], p. 25) for the value of the variable <u>var</u>, or the contents of the cell <u>var</u>, assigned by the "state vector" ξ; in our terminology, we would write simply ξ(<u>var</u>).)

A <u>relation</u> on a set X is a subset of $X \times X$. If the relation is called R, we write xRy if the ordered pair (x, y) is in the subset. A relation R is <u>reflexive</u> if xRx is always true and <u>irreflexive</u> if xRx is always false. It is <u>symmetric</u> if xRy implies yRx and <u>antisymmetric</u> if xRy and yRx imply $x = y$. It is <u>transitive</u> if xRy and yRz imply xRz. It is an <u>equivalence relation</u> if it is reflexive, symmetric, and transitive; a <u>partial ordering</u> if it is reflexive, antisymmetric, and transitive; and a <u>strict ordering</u> if it is irreflexive, antisymmetric, and transitive. For a strict ordering, xRy implies that yRx is false.

Let X be any set. A partition or a decomposition $\mathfrak{D}$ of X is a collection of disjoint subsets of X whose union is X. Given any decomposition $\mathfrak{D}$ of X, there is an equivalence relation R on X, under which xRy if and only if x and y belong to the same member of the decomposition $\mathfrak{D}$. Conversely, given an equivalence relation R, the collection of all distinct sets of the form $\{x \in X: xRy\}$ for some fixed y is a decomposition of X. Thus there is a one-to-one correspondence between decompositions of X and equivalence relations on X.

Let $\geq$ be any partial ordering on a set X; we may define the corresponding strict ordering $>$ by $x > y$ if $x \geq y$ and it is false that $x = y$. Given a strict ordering $>$, we may define the corresponding partial ordering $\geq$ by $x \geq y$ if either $x > y$ or $x = y$. Thus there is a one-to-one correspondence between partial orderings and strict orderings of a set. If $\geq$ is a partial ordering, then its inverse $\leq$, defined by $x \leq y$ if $y \geq x$, is a partial ordering; if $>$ is a strict ordering, then the strict ordering $<$ is defined similarly. In practice, if any one of the four orderings $<$, $\leq$, $>$, and $\geq$ is defined, we shall consider the other three to be defined in the obvious way.

Let $>$ be a strict ordering on a decomposition $\mathfrak{D}$ of a set X. Then $>$ extends in the obvious way to a strict ordering of X itself; that is, $x > y$ in X if $x \in D_x$ and $y \in D_y$, with $D_x \in \mathfrak{D}$, $D_y \in \mathfrak{D}$, and $D_x > D_y$ in $\mathfrak{D}$. The same construction may be made with a partial ordering, but the result is not a partial ordering.

A set G together with a relation $\rightarrow$ on G may be called a directed graph. The elements of G are called the nodes of the graph and the elements $x \rightarrow y$ of $G \times G$ are called the links of the graph. If x and y are nodes, then we say that there is a directed path of length n from x to y if there exist nodes $x_0, \ldots, x_n$, $x_0 = x$, $x_n = y$, with $x_{i-1} \rightarrow x_i$ for $1 \leq i \leq n$. There is always by definition a directed path of length zero from x to x. A directed path of length greater than zero from x to x is called a directed cycle. A graph with no directed cycles is called acyclic. (By "graph" we always mean a directed graph.) In any graph, the relation R defined by xRy if there exists a directed path (of length ≥ 0) from x to y is reflexive and transitive, and, if there are no directed cycles, it is also antisymmetric and is therefore a partial ordering. Accordingly, we sometimes refer to an acyclic graph as an ordered graph.

A node x of a graph is initial if $y \rightarrow x$ does not hold for any node y; it is terminal if $x \rightarrow y$ does not hold for any node y. An element x of a set X with a partial order relation R (called a partially ordered set) is maximal if $y \geq x$ implies $y = x$; it is minimal if $x \geq y$ implies $x = y$. A partially ordered set has a smallest element x if $y \geq x$ for all y; it has a greatest element x if $x \geq y$ for all y. A simple ordering is a partial ordering for which, given any two distinct elements x and y, either $x \geq y$ or $y \geq x$. A minimal element of a set with a simple ordering (called a simply ordered set) must be its smallest element; a maximal element of such a set must be its greatest element. Smallest and greatest elements are always unique.

A relation R satisfies the chain condition (or finite chain condition) if there are no infinite sequences $x_1, x_2, x_3, \ldots$, with x_1Rx_2, x_2Rx_3, ... If X is a set, R is a relation on X which satisfies the chain condition, and X' is an arbitrary subset of X, then X' must contain a minimal element. Otherwise, we could start with an arbitrary element x_1; since x_1 is not minimal, there exists x_2 with x_1Rx_2; since x_2 is not minimal, there exists x_3 with x_2Rx_3, and we may continue indefinitely in this way, contradicting the chain condition. For a simple ordering which satisfies the chain condition, this implies that every subset has a <u>smallest</u> element; i. e., that the given set is a <u>well-ordered</u> <u>set</u>.

Let $\mathcal{S} = \prod_{x \in M} V_x$ and let $S \in \mathcal{S}$. Since S is a map whose domain is M, we may speak of $S(x)$, for $x \in M$, or of $S|M'$, for $M' \subset M$. If $\mathcal{S}$ is interpreted as a collection of n-tuples, we may interpret $S(x)$ as the "x co-ordinate" of S, i. e., as one of the <u>n</u> objects which makes up the n-tuple; likewise, we may interpret $S|M'$ as an m-tuple for $m < n$, obtained by picking m of the given n objects, specifically those in certain fixed given positions in the n-tuple, and combining these into an m-tuple.

Let G be an ordered graph and let A and B be subsets of G. We say that $A > B$ if for each $a \in A$ there exists $b \in B$ with $a > b$ in the strict ordering of G. We say that $A \geq B$ if for each $a \in A$ there exists $b \in B$ with $a \geq b$ in the partial ordering of G.

Let $f: X \to Y$ be any function and let X' be a subset of X. We write $f(X')$ for the subset Y' of Y defined by $\{y \in Y: \exists x \in X'$ with $f(x) = y\}$.

The null set will be denoted by ϕ.

P-Sets and P-Functions

Our fundamental model will be based on the cartesian product of sets (p-set) and the function on a p-set, or from one p-set to another (p-function). P-sets and p-functions provide a unified approach to programming science.

Computer memories and sets of variables are examples of index sets M upon which to build p-sets $\mathcal{S} = \prod_{x \in M} V_x$.

Instructions, (executable) statements, computers, computations, programs, algorithms may be interpreted as p-functions on a p-set (that is, from a p-set to itself).

Messages, expressions, terms, factors, n-ary functions, predicates may be interpreted as p-functions from one p-set to another.

DEFINITION. A p-set is any set of the form $\mathcal{S} = \prod_{x \in M} V_x$, for some set M and, for each $x \in M$, a specified set V_x. (The V_x need not all be distinct.)

DEFINITION. A total p-function is a function $p: \mathcal{S}_1 \rightarrow \mathcal{S}_2$, where $\mathcal{S}_1$ and $\mathcal{S}_2$ are p-sets. A partial p-function is a function $p: \mathcal{S}_1' \rightarrow \mathcal{S}_2'$, where $\mathcal{S}_1' \subset \mathcal{S}_1$, and $\mathcal{S}_1$ and $\mathcal{S}_2$ are p-sets. If $\mathcal{S}_1 \neq \mathcal{S}_2$, we speak of a (total or partial) p-function from $\mathcal{S}_1$ to $\mathcal{S}_2$. If $\mathcal{S}_1 = \mathcal{S}_2$ (= $\mathcal{S}$), we speak of a (total or partial) p-function on $\mathcal{S}$.

If $M = \phi$ (the null set), $\prod_{x \in M} V_x$ has by definition one element, which we denote by φ. If M has exactly one element x, with $V_x = X$, then $\prod_{x \in M} V_x$ may be identified with X; any set X may be regarded as a p-set in this sense. The most commonly occurring p-sets of this kind are called types. The type integer, for example, is here identified with the set of all integers, for an ideal algorithm, or the set of all integers which may actually occur (in single precision) in a given computer, for an actual algorithm.

In an analogous way, we may define the types <u>real</u> and <u>Boolean</u> (or LOGICAL); in PL-I, a type is uniquely identified by a particular base, scale, mode, and precision. If M contains several elements x_i for all of which $V_{x_i} = X$ for the same set X, then we also refer to X as a <u>type</u>, and each x_i is called a <u>variable</u>, specifically a variable of type X, or an X-variable. Thus a given set M may include integer variables, real variables, and so on.

If any set $V_x = \phi$, then $\prod_{x\in M} V_x = \phi$, and there are no p-functions from it or to it whatsoever. If any V_x contains exactly one element, it may be eliminated from discussion; specifically, if $M' = \{x \in M: V_x \text{ has exactly one element}\}$ and $M'' = M - M'$, then there is a natural one-to-one correspondence between $\prod_{x\in M} V_x$ and $\prod_{x\in M''} V_x$. We shall assume from now on that each V_x contains at least two elements. If V_x contains exactly two elements, i. e., if $V_x = \{0, 1\}$, $\{\underline{\text{on}}, \underline{\text{off}}\}$, $\{\underline{\text{true}}, \underline{\text{false}}\}$, or the like, then x is called a <u>binary element</u> of M. In a computer memory M, all the elements are binary elements; they are such things as cores in core memory, flip-flops or "positions" in registers, and bit positions on tape or disk. In order to represent integer variables or real variables in such a memory, one must use a trick which may be stated in general terms as follows: if $\mathcal{S} = \prod_{x\in M} V_x$ and $\mathcal{D}$ is a decomposition of M, then there is a natural one-to-one correspondence between $\mathcal{S}$ and $\prod_{D\in\mathcal{D}}(\prod_{x\in D} V_x)$, and, for each $D \in \mathcal{D}$, $\prod_{x\in D} V_x$ may be taken as the type of D. Thus we decompose our set of binary elements into words and registers; the type of each word is the set of bit patterns that may appear in it.

Let p be a total or partial p-function from $\mathcal{S}_1 = \prod_{x\in A} V_x$ to $\mathcal{S}_2 = \prod_{x\in B} V_x$. Let $A' \supset A$ and let $A' \cup B \supset B' \supset B$. We assign to each $x \in A'$ an arbitrary set V_x, if such a set has not already been assigned because $x \in A$. We now define a p-function, p', from

$\mathcal{S}_1' = \prod_{x \in A'} V_x$ to $\mathcal{S}_2' = \prod_{x \in B'} V_x$ by $p'(S) = S'$, where $S'(x) = S(x)$ for $x \notin B$ (this is well-defined because $A' \cup B \supset B'$), and, if $S'' = p(S|A)$, then $S'(x) = S''(x)$ for $x \in B$. We refer to p' as a memory extension of p; it is total or partial according as p is total or partial. In what follows, we shall use the same letter for both functions when there can be no confusion as to meaning; this reflects the fact that when we add new elements to a memory, the p-functions on that memory retain their character. In particular, we have the following special cases:

(a) $B' = B$. This allows us to define p-functions for expressions. Any set X may be treated as a p-set, as mentioned above; if $f: X \rightarrow Y$ is an arbitrary function, it may be treated as a p-function from the p-set X to the p-set Y. It may then be extended to a function $p: \mathcal{S} \rightarrow Y$, where $\mathcal{S} = \prod_{x \in M} V_x$ for any set M which contains an element x with $V_x = X$. We shall refer to this p-function as $\{f(x)\}$; it clearly depends on the choice of $x \in M$. If $X = X_1 \times \ldots \times X_n$, so that the elements of X are n-tuples and $f: X \rightarrow Y$ is a function of n arguments, then f may be extended to a function $p: \mathcal{S} \rightarrow Y$, where $\mathcal{S} = \prod_{x \in M} V_x$ for a set M which contains elements $x_1, \ldots, x_n$ with $V_{x_i} = X_i$, $1 \leq i \leq n$. We shall refer to this p-function as $\{f(x_1, \ldots, x_n)\}$; it clearly depends on the choices of the x_i, which need not be all distinct. These p-functions are total or partial according as f is total or partial. If Y = integer, a function $p: \mathcal{S} \rightarrow Y$ will be called an integer p-function; in an analogous way, we may define real p-functions, Boolean p-functions, and so on. The set Y may be called the type of the p-function.

(b) $A' = B'$. This allows us to define assignments. Either the function f or the function $\{f(x)\}$ of the previous example may be extended, as above, to a p-function $q: \mathcal{S} \rightarrow \mathcal{S}$, where $\mathcal{S} = \prod_{x \in M} V_x$

for any set M containing an element x with $V_x = X$ and an element y with $V_y = Y$. We shall refer to this function as the <u>assignment</u> $\{y \leftarrow f(x)\}$; it clearly depends on the choice of x and y. In particular, if X = Y, we may have f(x) = x, giving rise to the assignment $\{y \leftarrow x\}$. Or we may have x = y, which produces assignments of the form $\{x \leftarrow f(x)\}$; in particular, if $c \in X$ and $f(x) \equiv c$ is a constant function, then $\{x \leftarrow c\}$ is the corresponding constant assignment. In a similar way, we may define the assignment $\{y \leftarrow f(x_1, \ldots, x_n)\}$; it, too, clearly depends on the choice, in M, of the elements x_1, ..., x_n and y.

Another type of extension which produces a p-function is as follows. Let p_i, $1 \leq i \leq n$, be total or partial p-functions from p-sets $\mathcal{S}_i$ to sets X_i, and let $f: X_1 \times \ldots \times X_n \rightarrow Y$. Let $\mathcal{S}_i = \prod_{x \in M_1} V_x$, let $M = \bigcup_{i=1}^{n} M_i$, and, for this M, let $\mathcal{S} = \prod_{x \in M} V_x$. The function $g: \mathcal{S} \rightarrow Y$, defined by $g(S) = f(p_1(S|M_1), \ldots, p_n(S|M_n))$ will be called a <u>star-extension</u> $f^*(p_1, \ldots, p_n)$ of f. If n = 1, the star-extension $f^*(p_1)$ reduces to the ordinary composition $p_1 \circ f$. Common examples of star-extended functions include:

(a) <u>Binary and unary operations</u>. If f(a,b) = a+b may be interpreted as addition on Y, then $f^*(e,e')$, for any two p-functions e and e' from $\mathcal{S}$ to Y, is another such function, which may be denoted by e+e'. If Y = <u>integer</u>, then e+e' is an integer p-function derived from the integer p-functions e and e'. The same construction will produce e-e', e*e', etc., as well as -e; and all of these may be real p-functions, complex p-functions, or, in general, p-functions of any type on which operations are defined. The process of defining operations on $\prod_{x \in D} V_x$ so as to make it look like <u>integer</u> or <u>real</u>, where D is a word in a computer, each $x \in D$ is a bit in that word, and each $V_x = \{0, 1\}$, leads to certain inaccuracies which are identified and analyzed as <u>roundoff error</u>, <u>truncation</u>

error, and the like. It is not necessary, however, to have an "exact" model of arithmetic operations in order to define star-extensions; in an actual computer, $e+e'$, for example, is the star-extension of the actual addition function defined on $\prod_{x\in D} V_x$, taken as the set real, where e and e' are real p-functions (i. e., p-functions of type $\prod_{x\in D} V_x$).

(b) Relations. Let Boolean = {true, false}. A relation on a set Y is a subset R of $Y \times Y$; if (y_1, y_2) is in this subset, for $y_1, y_2 \in Y$, we write $y_1 R y_2$. There is a natural correspondence between subsets of any set and maps from that set into Boolean; the image of an element in the set is true if and only if it is in the given subset. Thus, if R is a relation on Y and e and e' are two p-functions of type Y, then $R^*(e,e')$ is a Boolean p-function, which may be denoted by {eRe'}. Turning the argument around, {eRe'} is a subset of $\mathcal{S}$, where e and e' are p-functions on $\mathcal{S}$. The elements of $\mathcal{S}$ may be called states, since they "describe the system" by assigning a value to each variable; subsets of $\mathcal{S}$, or, alternatively, Boolean p-functions on $\mathcal{S}$, may then be called state conditions. If x is any Boolean variable, the memory extension of the given correspondence between V_x and Boolean is a state condition {x}; these, together with the constant state conditions {true} and {false} and the various {eRe'} as mentioned above, constitute the fundamental state conditions of any p-set $\mathcal{S}$. These may be further combined by means of Boolean operations, which are star-extensions of the well-known binary operations on the set Boolean, such as and and or. Thus if C and C' are any two state conditions, so are C and C' and C or C', these being defined as and*(C, C') and or*(C, C') respectively. Interpreting these as subsets of $\mathcal{S}$, we have C and C' = $C \cap C'$ and C or C' = $C \cup C'$. Similar constructions may be applied to the unary operation not on the set

Boolean (in particular, not C = $\mathcal{S}$ - C) and to and and or treated as n-ary functions on Boolean.

Actual state conditions occurring in programs may be much more complex than the ones mentioned above. For example, suppose that we have a routine to take the variables A(1), ..., A(N) and sort them, keeping the original values of the A(i) and putting the new values in B(1), ..., B(N). The condition that the B(i) are sorted is then $\bigcup_{i=1}^{N-1} \{B(i) \leq B(i+1)\}$. The condition that the B(i) are "the same" as the A(i) is a bit harder to state. Let X = $\{1, \ldots, N\}$, and let G be the set of all one-to-one maps from X onto X; then the condition above is $\bigcup_{f \in G} (\bigcap_{i=1}^{N} \{A(i)=B(f(i))\})$. As an even more complex example, let A be an arbitrary data structure, such as, for example, a particular S-expression in the LISP language. Following d'Imperio, we use the term "data structure" for mathematical expressions -- such as S-expressions in LISP -- which are independent of any particular computer, and "storage structure" for a particular method of representing a given data structure in a particular computer memory. In general, a data structure may be represented in numerous ways as a storage structure, and the fact that a particular memory contains a storage structure which represents a given data structure is a state condition on that memory. Programs which operate on data structures will always have state conditions of this kind associated with them.

In denoting the intersection of two state conditions, the comma may be used in place of the word and; thus the condition $\{i=3, j=4\}$ is the condition $\{i=3$ and $j=4\}$. In general, we speak of a "set of conditions" $C_1, \ldots, C_n$, when we mean their intersection; if the intersection C of a set of conditions "includes" a particular C_i, then, as subsets of $\mathcal{S}$, $C_i \supset C$ (not $C_i \subset C$!). If $\mathcal{S}$ is a p-set and $\mathcal{S}'' \subset \mathcal{S}' \subset \mathcal{S}$, then $\mathcal{S}''$ may be called a subcondition of $\mathcal{S}'$. If $\mathcal{S}'$ is thought of as a B-expression on $\mathcal{S}$ and $p: \mathcal{S} \to \mathcal{S}$, then the B-expression $p \circ \mathcal{S}'$ will be denoted by $p(\mathcal{S}')$; this is also a subset of $\mathcal{S}$, namely $p(\mathcal{S}') = \{S \in \mathcal{S} : \exists S' \in \mathcal{S}', p(S') = S\}$.

(c) Arrays. An array $x[a_1:b_1, \ldots, a_n:b_n]$, where the a_i are lower bounds and the b_i are upper bounds, consists of a map $x: X_1 \times \ldots \times X_n \to M$, where $X_i = \{a_i, a_{i+1}, \ldots, b_i\}$ and M unerlies a p-set $\mathcal{S} = \prod_{x \in M} V_x$. The star-extension $a^*(e_1, \ldots, e_n)$, where each e_i is a p-function whose values lie in the range $a_i \leq e_i(S) \leq b_i$, is then a p-function whose values are elements $x \in M$ of this array. In general, p-functions of type M will be called address functions. If T is an arbitrary type and $M_T \subset M$ is the set of all elements of M having type T, then a p-function of type M_T is a T-address function. For example, if $\underline{a}$ is a real array, and $T = \underline{\text{real}}$, then $\underline{a}: X \to M_T$ may be star-extended to the real address function $a^*(e)$. Address functions allow us to define more general assignments than those discussed earlier; if $\underline{s}$ is a T-address function and $\underline{e}$ is a p-function of type T, we may define the assignment $\{\underline{s} \leftarrow \underline{e}\}$ as a function $p: \mathcal{S} \to \mathcal{S}$, where $p(S) = S'$ for $S'(z) = S(z)$ whenever $z \neq x = s(S)$ and $S'(x) = e(S)$. Each assignment $\{y \leftarrow e\}$ as discussed earlier is an assignment $\{s \leftarrow e\}$ for the constant T-address function $s(S) \equiv y$; the more general type of assignment allows us to assign values to subscripted variables where the subscript must itself be calculated.

A generalized version of the star-extension allows us to treat functions with side-effects. Let p_i, $1 \leq i \leq n$, be total or partial p-functions from p-sets $\mathcal{S}_i$ to sets X_i, and suppose that each p_i has a "side effect" p_i', which may be an arbitrary p-function on an arbitrary p-set $\mathcal{S}_i'$. Let $\mathcal{S}_i = \prod_{x \in M_i} V_x$, let $\mathcal{S}_i' = \prod_{x \in M_i'} V_x$, let M be the union of all the M_i and all the M_i', and, for this M, let $\mathcal{S} = \prod_{x \in M} V_x$. (It is assumed that the X_i are separate and disjoint from both the M_i and the M_i'.) Let $g: \mathcal{S} \to Y$ be defined by $g(S) =$

$f(p_1(S_1|M_1), \ldots, p_n(S_n|M_n))$, where $S_1 = S$ and $S_{i+1} = p_i(S_i)$, $1 \leq i < n$. Let $g': \mathcal{S} \to \mathcal{S}$ be defined by $g'(S) = p_n(S_n)$, where the sequence $S_1, \ldots, S_n$ is defined above. Then the pair $(g, g') = f^*((p_1, p_1'), \ldots, (p_n, p_n'))$ is the <u>dynamic</u> <u>star-extension</u> of the pairs (p_i, p_i'). If f also has an associated side-effect f', we define $g'(S) = f'(p_n(S_n))$, and $(g, g') = (f, f')^*((p_1, p_1'), \ldots, (p_n, p_n'))$ is a still more general form of dynamic extension. If the side effect functions p_i' and f' are the identity, the dynamic star-extension reduces to the ordinary star-extension; the more general formulation allows us to treat such expressions as $\{i+j\}$ in ALGOL, for example, where i and j may both be references to procedures (i. e., subroutines) with no arguments, possibly using and/or setting the same data.

Both kinds of star-extension may be used to define the effect of calling a function or a subroutine in FORTRAN or using a procedure in ALGOL when the actual parameters are taken to be general expressions. If $f(x_1, \ldots, x_n)$ is defined for x_i of type T_i, then the function denoted in programming languages by $f(e_1, \ldots, e_n)$, where e_i has a p-function of type T_i, is actually $f^*(e_1, \ldots, e_n)$. This may be taken in either the static or dynamic sense, which allows any of the e_i to have side effects. The way in which a parameter is called determines the type T_i for that parameter. If T_i = <u>real</u>, <u>integer</u>, or the like, we have "call by value" as in ALGOL; if $T_i = M_U$ for U = <u>real</u>, <u>integer</u>, or the like, we have "call by address" as in FORTRAN. Calling by name in ALGOL was originally defined as a function which transformed one ALGOL program into another so as to eliminate the calling statements; but, using Jensen's device, we can interpret this type of parameter usage as above by identifying the elements of T_i with p-functions of type M_U, for U = <u>real</u>, <u>integer</u>, or the like.

Programs

Let P be a finite set of statements F_i which make up a program. From our discussion of assignments, which are functions from a p-set $\mathcal{S}$ to itself, one might expect that, in general, each F_i would be of this form. But there is more to the story than that, because each statement in a program not only performs some action — indicated by a p-function -- but it also indicates which statement, if any, is to be performed next. This depends, in general, on the values of the program variables, i. e., on the state $S \in \mathcal{S}$, and it is thus a function from $\mathcal{S}$ into P itself, or a p-function of type P.

DEFINITION. A program on a p-set $\mathcal{S}$ is a finite set P of statements $F_i = (P_i, N_i)$, where P_i is a total or partial p-function on $\mathcal{S}$ and N_i is a total or partial function from $\mathcal{S}$ into P. The function P_i is called the program function of F_i, and N_i is called the next-statement function of F_i.

A program is thus a finite set of function pairs; but it is also, in another sense, a single p-function, or, more precisely, it is so in two different and related senses.

DEFINITION. Let P be a program on $\mathcal{S} = \prod_{x \in M} V_x$, let $M' = M \cup \{\lambda\}$, and let $V_\lambda = P$; the set $\mathcal{T} = \prod_{x \in M'} V_x$ may thus be identified with $\mathcal{S} \times P$. Then the execution function of P is then defined to be the function $f: \mathcal{T} \to \mathcal{T}$ defined by $f(S, F_i) = (P_i(S), N_i(S))$, where $F_i = (P_i, N_i)$.

The motivation for this definition is not too difficult to discern. Suppose that we are given an element $S \in \mathcal{S}$ and a particular statement F_i to be the value of λ. Then, by executing this statement, we get a new element of $\mathcal{S}$, or a new configuration of the variables of the program; and we also get a new statement F_i, namely the statement given by the next-statement function N_i applied to S. Thus, starting with an element of $\mathcal{S} \times P$, we obtain in a natural way another element of $\mathcal{S} \times P$. This leads us to a p-function on this expanded p-set, which will be a partial function if any of the P_i or the N_i are partial functions. We shall denote the execution function of a program P by P itself; this will generally cause no confusion, because in one sense P is a set and in the other sense it is a function.

DEFINITION. Let P be a program on $\mathcal{S}$ and let F_1 be an arbitrary statement of P. The <u>computation</u> of P with respect to F_1 is defined to be the p-function $g: \mathcal{S} \rightarrow \mathcal{S}$ as follows. Let $S \in \mathcal{S}$, let $T_0 = (S, F_1)$, and set $T_i = P(T_{i-1})$ for as long as this sequence is defined, where P is here taken to be the corresponding execution function as defined above. If the sequence T_i terminates because some $T_k = (S', F_j)$ for $F_j = (P_j, N_j)$ such that $N_j(S)$ is not defined, we set $g(S) = P_j(S')$. If it terminates because $P_j(S)$ is not defined, or if it does not terminate at all, then $g(S)$ is undefined. The sequence T_i, for as long as it is defined, starting with any (S, F_1), is the <u>computation sequence</u> of S with respect to F_1.

The computation of a program is the p-function obtained by viewing it, in a sense, as a single step. That is, a configuration $S \in \mathcal{S}$ of the variables of the program is given; the program is then run in the usual way, that is, its execution function is performed over and over until the program exits. When this happens, we have a new configuration of the variables of the program, and the correspondence between the old S and the new leads to a p-function on $\mathcal{S}$. Exiting from a program is defined by inability to find a next statement; in particular, it is perfectly permissible for N_i to be a partial function in the degenerate sense that it is left <u>completely</u>

undefined. In this case, $F_i = (P_i, N_i)$ is an <u>exit statement</u> of the program (and the values of P_i are immaterial). It is not necessary, however, for an actual program to have any exit statements, so long as it has <u>conditional exit statements</u> $F_i = (P_i, N_i)$ where N_i is a general partial function. If all the functions N_i are total, then it is impossible to define a computation of the program.

We emphasize that any statement in a program which is not an exit statement may be taken as F_1 in the above definition. The statements in a program form an ordinary unordered set. When we use the computation of a program P with respect to a statement F_1, we shall generally denote it by P'. There is a more restricted definition of a program which suffices for many purposes, although it also has certain drawbacks.

DEFINITION. An <u>ordered program</u> on a p-set $\mathcal{S}$ is a finite sequence P of <u>statements</u> $F_i = (P_i, N_i)$, $1 \leq i \leq n$, where P_i is a total or partial p-function on $\mathcal{S}$ and N_i is a total or partial function from $\mathcal{S}$ into P. The <u>unordered program</u> Q <u>of</u> P is the set $F_i' = (P_i, N_i')$, for the P_i as above, where $N_n' = N_n$ and, for $i < n$, we have $N_i'(S) = N_i(S)$ whenever $N_i(S)$ is defined and $N_i'(S) = F_{i+1}$ otherwise. The <u>execution function</u> of P is the execution function of Q; the <u>computation</u> of P is the computation of Q with respect to F_1. Other computations of P are those of Q.

An ordered program thus has a computation, without any qualifying phrase; it has a first statement and a last statement, and, in particular, it can have only one last statement. Most of the statements in an ordered program may be specified by simply giving a function P_i and leaving N_i completely undefined, since in an ordered program the "next statement" is by default the next in order.

In an unordered program we may easily distinguish conditional and unconditional statements. An unconditional statement F_i is one for which N_i is a constant function. This includes ordinary assignment statements and subroutine or procedure calls as well as unconditional transfers; it also includes conditional assignments such as (if A = B then set C = D). A true conditional statement is one for which N_i has more than one value. A transfer is a statement F_i for which P_i is the identity function; most conditional statements in practice are conditional transfers. A brief digression about conditional quantities in programs may be in order at this point. If $f: A \rightarrow B$ and $g: B \rightarrow C$ are functions, we may always speak of the composition $f \bullet g = h: A \rightarrow C$. When B is the set $\{$true, false$\}$, a special situation arises, because the formal definition of the cartesian product $C \times C$ is the set of maps from $\{0, 1\}$ into C. An element of $C \times C$ is normally thought of as an ordered pair (c, c'), but, as the above discussion shows, it can serve as the function g in the above composition equation. Thus, identifying $\{0, 1\}$ with $\{$true, false$\}$ in the obvious way, one can always speak of the composition of the function f above with the ordered pair (q_1, q_2), where $q_1, q_2 \in Q$ for any set Q. If f is a Boolean p-function, the result is a p-function of type Q. This construction may be used to obtain all the conditional quantities that are found in programs. If Q = P, we have a conditional transfer $f \bullet (F_i, F_j)$, or if f then go to F_i else go to F_j; if Q = real or integer, we have the real or integer p-function of a conditional expression, such as if f then 5 else 7 (which is $f \bullet (5, 7)$); finally, if q_1 and q_2 are p-functions on $\mathcal{S}$, we get a conditional assignment or the like.

There is a certain analogy between a program in this sense and a finite automaton as defined by several authors, which is a finite set of states with an initial state, a collection of final states, and a method of getting from one state to the next. There is a difference, however, between the two concepts, because the method of proceeding from one state of an automaton to the next is dependent upon an input from outside the automaton, and the sequence of these inputs is taken to be infinite. In contrast, the method of proceeding from one statement of a program to the next is dependent upon the current element $S \in \mathcal{S}$, and in all practical cases $\mathcal{S}$ is large but finite. It is to be emphasized that the <u>states</u> of an automaton are analogous to the <u>statements</u> of a program; programs also have states, which are elements $S \in \mathcal{S}$, but these do not figure in the analogy.

To the types <u>real</u>, <u>integer</u>, and the like, we can now add the type <u>label</u>, which is either the set of statements P itself or a separate set L together with a label mapping $\ell: L \rightarrow P$. Variable names appearing in ASSIGN statements in FORTRAN are of this type, and the function N_i for an assigned GO TO K statement has the form $N_i(S) = S(K)$. Switches in ALGOL and computed GO TO statements in FORTRAN correspond to mappings $f: X \rightarrow P$, where $X = \{1, \ldots, n\}$; if <u>e</u> is a p-function whose values lie in this range, then $f^*(\underline{e})$, or $e \circ f$, is the function N_i for a <u>go to</u> statement which makes reference to a switch. By analogy with the composition of a Boolean p-function and an ordered pair discussed earlier, we may write this composition as $e \circ (x_1, \ldots, x_n)$, where each x_i is an element of P or, by extension, of L, so that $(x_1, \ldots, x_n)$ is the switch itself.

Complete Programs

DEFINITION. A program is complete if it has one statement $X = (P_X, N_X)$, where P_X is the identity and N_X is nowhere defined, and for every other statement $F_i = (P_i, N_i)$, N_i is a total function.

DEFINITION. Let P be any program. The completion of P is the program obtained by adding to P a statement $X = (P_X, N_X)$, where P is the identity and N_X is nowhere defined, and redefining each of the other statements $F_i = (P_i, N_i)$ of P to be (P_i, N_i'), where $N_i'(S) = N_i(S)$ whenever $N_i(S)$ is defined and $N_i'(S) = X$ whenever $N_i(S)$ is undefined.

Clearly the completion of a program is complete.

THEOREM. Let P be a program, let F_1 be a statement of P, and let Q be the completion of P. Then the computation of Q with respect to F_1 is the same as the computation of P with respect to F_1.

PROOF. Let $T_0, \ldots, T_m$ be an arbitrary computation sequence of P, where $T_0 = (S, F_1)$. Set $T_i = (S_i, F_i')$, where $F_i' = (P_i'', N_i'')$. Since $N_i''(S_i)$ is always defined for $0 \leq i < m$, the computation sequence of Q beginning with T_0 includes $T_0, \ldots, T_m$. At this point, the computation sequence of P ends, and $P'(S) = P_m''(S_m)$, while the computation sequence of Q continues to $T_{m+1} = (P_m''(S_m), X)$. Since N_X is nowhere defined, this computation sequence stops here, and $Q'(S) = P_X(P_m''(S_m)) = P_m''(S_m)$ since P_X is the identity; i. e., $Q'(S) = P'(S)$. Similarly, if T_0 begins an infinite computation sequence of P, then, since all the values of the next-statement functions in this sequence are defined, the sequence will also be a computation sequence for Q. Hence in all cases the computation sequences of P and Q are the same, starting from (S, F_1), and hence the computations of P and Q with respect to F_1 are the same. This completes the proof.

This theorem may be used to simplify the proofs of theorems about computations. In a complete program, a computation sequence $T_0, \ldots, T_m$ which begins with $T_0 = (S, F_1)$ will end, if it ends at all, with $T_m = (S', X)$, and $P'(S) = S'$, where P' is the computation of P with respect to F_1. In an incomplete program, it is necessary to add one extra step at the end. It is possible in many theorems about programs to assume that they are complete, because any program which is not complete may be replaced by its completion with no change in its computation.

Languages

Let Σ be a finite character set, let Σ^* be the set of all strings of characters in Σ, and let $L \subset \Sigma^*$ be the set of legal strings in some programming language. That is, each string in L represents a program. In line with our previous discussion, we should like to associate a set M, a set V_x for each $x \in M$, and a p-function on $\mathcal{S} = \prod_{x \in M} V_x$ with each such string. This may be done along lines suggested by Knuth [7]. Knuth considers the productions of a language and proposes the association of quantities of various kinds with each symbol in the language. The terminal symbols are directly associated with quantities, which may be integers, sets, or various other objects; the associations with non-terminal symbols are defined in terms of the productions by which these are built up. We have seen how this may be done for a programming language by means of star-extensions. Specifically, if t_1 and t_2 are terms in an expression e, and the string e is the concatenation of the three strings t_1, '+', and t_2, then the p-function e' of e is the star-extension $f^*(t_1', t_2')$, where $f(a, b) = a+b$ for integers or real numbers a and b, and t_1' and t_2' are the respective p-functions of t_1 and t_2.

Some of the statements in a language, such as assignment statements, may be associated with p-functions in this way. Another very common way to define a statement in a language is to specify how it could be eliminated from its program by changing the program to an equivalent one. Specifically, we define a mapping from L to another language L', which has fewer productions than L. The length of a result string in L' may be much greater than that of the string in L which produced it; but, in a computer, this may not matter, because it is not necessary for the computer to contain the entire result string; all we need is an algorithm by which the characters of the result string may be successively output, and often this is easily derivable from the original string in L. The macro statements in PL-I provide a good example of this type of mapping, as do the programmer-defined statement types of various extendible languages.

Many well-known statements in programming languages are susceptible to both of these types of treatment. As we have seen, one of these is the procedure in ALGOL with parameters called by name. In general, any procedure call statement may be replaced, within a program, by the entire procedure, with parameters substituted according to various rules dependent on the type of call; this serves to eliminate procedure call statements. Procedure calls inside a statement are a bit trickier; one must decompose the statement into parts. For example, z:=a/b + f(c,d) + e-f, where f(x,y) is defined by a complex program, would have to be replaced by something like z:=a/b followed by the statements of the program suitably modified to produce a value t for f(c,d), and then followed by z:=z+t+e-f. If we do decide to associate a p-function with a procedure call, this may be done in two ways. We may use the computation of the function being called, which is a p-function on a set $\mathcal{S} = \prod_{x\in M'} V_x$, and, by identifying the parameters and the value of the function as elements of M, we obtain a p-function of type equal to the type of the value; the computation of the function then serves as the side effect. Or we can build a p-function which effectively saves a return address, and consider all of our statements as memory extensions to a set $\mathcal{S} = \prod_{x\in M} V_x$ for a single global set M, which includes the variables of the various procedures or subroutines as well as those of the main program.

Another statement type which may be treated in both of the above ways is the iteration statement. If A, B, and C are constants, the FORTRAN statement DO n v=A, B, C followed by a group of statements G, or the ALGOL statement for v:=A step C until B do G where G may possibly be a compound statement or a block, may be replaced, within its program, by a number of iterations of G, in each of which any occurrences of the variable v have been replaced by expressions of the form A+kC. If any of A, B, or C is allowed to be variable, this will not work. However, in all cases we may replace iteration statements in programs by initialization, incrementation, and conditional transfer statements. In FORTRAN this is done by replacing DO n v=A, B, C by v=A and, if the following statement is labelled k (where k is added to the program if that statement is unlabelled), inserting v=v+C (v=v+1 if C is missing) and IF (v.LE.B) GO TO k immediately following statement n; this process must be done in order from innermost loops to outermost loops ending at the same statement. In ALGOL this is done by replacing for v:=L1,L2,...,Ln do g by a series of statements corresponding to the elements Li of the for-list as follows, where g has been coded as a procedure within the current block and the labels F and G are chosen dynamically as new symbols not currently appearing in the given block:

TYPE OF FOR-LIST ELEMENT	RESULTING ALGOL STATEMENTS
e	v:=e; g;
x step y until z	v:=x; F: if (v-z)×sign(y)>0 then go to G; g; v:=v+y; go to F; G:
p while q	F: v:=p; if ¬q then go to G; g; go to F; G:

These language transformations for iteration statements, unlike those discussed earlier, never materially increase the size of the program involved. For this reason, when we are verifying a program or defining the semantics of a programming language, <u>we shall always assume that this particular transformation has already been carried out</u>, so that, in the resulting program, there are no explicit iteration statements whatsoever.

We remark that every programming language has a "universal" p-function $p_L : \mathcal{S} \rightarrow \mathcal{S}$, where $\mathcal{S} = \prod_{x \in M} V_x$, as follows. The set M is the union of two disjoint subsets A and B. The set A is the set of all possible variable names which may appear in programs written in the given language; if $x \in A$, then V_x is the set of all possible values that such a variable can have. The set B corresponds to the natural numbers, and, for each $n \in B$, V_n is the character set of the given language. Given $S \in \mathcal{S}$, it is assumed that the characters S(1), ..., S(n), for $1 \in B$, ..., $n \in B$, for some n, specify a program in the given language. It is also assumed that the language contains an END statement or its equivalent, so that by scanning the characters S(1), S(2), ..., in the forward direction for a given state S, the value of <u>n</u> above may be determined (provided that S actually specifies a legal program). In this case S also specifies a state of the variables of the program upon input, and p(S) will specify their state upon output. The resulting function is partial in a rather extreme sense, since only if a legal program is specified in the set B is the function p_L defined at all. This example may be extended to cover the case in which the characters of the source program are not contained in memory but are produced as the result of an algorithm acting upon the characters of a program in a more complex language, as discussed earlier.

Correctness

Let P be a program on $\mathcal{M}$ and let P' be the computation of P with respect to $F_1 \in P$. If X is either an element or a subset of $\mathcal{M}$ and Y is either an element or a subset of P, then we may identify a corresponding subset (X, Y) of $\mathcal{J} = \mathcal{M} \times P$. If Y = P, we abbreviate (X, Y) by X; thus if $\mathcal{M}' \subset \mathcal{M}$ we may also write $\mathcal{M}' \subset \mathcal{J}$. (This is also true by memory extension, if $\mathcal{M}'$ is interpreted as a Boolean p-function on $\mathcal{M}$.) If X = $\mathcal{M}$, we have the conditions $\{\lambda = F_i\}$ for various $F_i \in P$, and $\{\lambda = Q\}$ for various subsets $Q \in P$. The special condition $\{(S, F_i) \in \mathcal{J}: N_i(S)$ is not defined$\}$ will be denoted by $\{$<u>exit</u>$\}$; it is the condition that P has terminated.

We are now in position to ask the question: What do we mean when we say that P "works properly"? One thing, of course, that we always mean is that it terminates — that is, it does not go into an endless loop or execute an illegal instruction. (This is true even in continuously operating computers, which never actually stop; the overall program of the computer may not terminate, but it is not this program which we would want to verify, but rather the individual, terminating computations which it performs.) It is also necessary, however, for P to "compute the right answers," a concept whose mathematical formulation is not at all obvious. Suppose first that the program P is supposed to compute a function f of n arguments, regarded as a function from $X_1 \times \ldots \times X_n$ into Y. This implies that we have chosen certain special variables $x_1, \ldots, x_n \in M$ and $y \in M$, where $\mathcal{M} = \prod_{x \in M} V_x$, such that $V_{x_i} = X_i$, $1 \leq i \leq n$, and $V_y = Y$. If the assignment $\{y \leftarrow f(x_1, \ldots, x_n)\}$ for these choices of y and the x_i, or the memory extension of this assignment to $\mathcal{M}$, were actually the computation of P with respect to some given starting statement $F_1 \in P$, then P would certainly compute f properly. This, however, is too unrealistic a condition to ask. In general, any program which computes f will use certain temporary variables, and when the program is finished these variables will remain set. Any memory extension of $\{y \leftarrow f(x_1,\ldots,x_n)\}$, however, cannot change the values of any temporary variables.

Let us weaken this a bit. Suppose that v_i is an arbitrary element of V_{x_i}, $1 \leq i \leq n$. Then $\{x_1=v_1, \ldots, x_n=v_n\}$ is a state condition $C \in \mathcal{S}$, and similarly $\{y=f(x_1,\ldots,x_n)\}$ is a state condition $C' \subset \mathcal{S}$. What we would really like to require of the computation P' of P with respect to F_1 is that $P'(C) \subset C'$; that is, if $S \in C$, then $P'(S)$ is defined and $P'(S) \in C'$. The condition that $P'(S)$ be defined is, of course, the condition mentioned above that the algorithm terminates. The statement $P'(C) \subset C'$ is clearly equivalent to the intuitive condition that P computes f; furthermore, it covers the more general case where the "value" of f is expressed by the conditions of several result variables, or when the purpose or partial purpose of f is to change the values of some or all of its input variables. We formalize this discussion by making the following definition.

<u>DEFINITION</u>. Let P be a program on $\mathcal{S}$, let F_1 be a statement of P, let $C \subset \mathcal{S}$, and let $C' \subset \mathcal{S}$. (Here C is a condition presumed to hold before the program starts, while C' is a condition desired after the program is finished.) Then P is <u>correct</u> with respect to F_1, C, and C' if $P'(C) \subset C'$, where P' is the computation of P with respect to F_1.

Partial Correctness and Floyd's Theorem

The basic ~~first~~ step in proving a program correct was first stated in print by Floyd in [3], although Floyd credits the idea to Perlis and Gorn. It essentially involves dividing the correctness problem into two parts.

DEFINITION. Let P be a program on $\mathcal{S}$, let F_1 be a statement of P, let $C \subset \mathcal{S}$, and let $C' \subset \mathcal{S}$. Then P is partially correct with respect to F_1, C, and C' if $P'(S) \in C'$ whenever $S \in C$ is such that $P'(S)$ is defined. (The terms "correct" and "partially correct" in this sense are due to Manna [10].)

Clearly, if we can prove that a program is partially correct, then all we have to do to prove it correct is to show that it terminates -- i. e., that $P'(S)$ is actually defined for all $S \in C$. Later, we shall discuss methods of proving that an algorithm terminates. What Floyd showed is that the problem of partial correctness can be settled by purely local arguments, i. e., arguments involving the flow of control from one statement to another in the program. The basic idea is as follows. Let F_i be a statement of the program P, and let us associate with F_i a state condition $\mathcal{S}_i \subset \mathcal{S}$. This is presumed to be a condition satisfied by the variables of the program just before F_i is executed. In particular, $\mathcal{S}_1$ is associated with F_1. If we start the program at F_1 with a state $S \in \mathcal{S}_1$, we should like to prove that, as the program proceeds, each (S_i, F_i) in the computation sequence is such that S_i is contained in $\mathcal{S}_i$. But this will follow if for every pair of statements F_i and F_j which are "next to each other" -- i. e., control passes from F_i to F_j in the execution of the program -- the condition $S_i \in \mathcal{S}_i$ implies $S_j \in \mathcal{S}_j$ whenever the next statement after (S_i, F_i) is (S_j, F_j). The following intuitive method may then be used to prove a program partially correct:

(1) Understand the program well enough that you can specify a condition $\mathcal{S}_i$ for each statement $F_i = (P_i, N_i)$, such that $\mathcal{S}_i$ is the condition satisfied by the variables just before F_i is performed. Also, specify an exit condition C'.

(2) Prove separately for each pair of statements F_i and F_j that $S_i \in \mathcal{S}_i$ implies $P_i(S_i) \in \mathcal{S}_j$ whenever $N_i(S_i) = F_j$.

(3) Prove for each statement F_i that if $N_i(S_i)$ is not defined and $S_i \in \mathcal{S}_i$, then $P_i(S_i) \in C'$.

The process of identifying which statements in a program are "next to each other" is facilitated by defining the graph of the program.

<u>DEFINITION</u>. Let P be a program on $\mathcal{S}$. The <u>directed graph</u> of P is the directed graph whose nodes are the statements F_i of P, and such that $F_i \rightarrow F_j$ if and only if there exists $S \in \mathcal{S}$ with $N_i(S) = F_j$.

A flowchart of a program (in complete detail) is a representation of its directed graph. An <u>overall</u> flowchart of a program is a representation of the directed graph of the program obtained by "collapsing" certain groups of statements of the original program into single statements; we shall return to this idea later when we consider factoring of programs. Step 2 above now needs to be performed only for each <u>link</u> in the directed graph of the given program. We could, if we wanted to, associate the conditions $\mathcal{S}_i$ with the links in the graph, rather than with individual statements; this, in fact, is what was done by Floyd in [3]. Also, Floyd uses a programming model based on the predicate calculus, rather than our p-set model.

We now formalize the above arguments.

<u>DEFINITION</u>. Let P be a program on $\mathcal{S}$. A <u>precondition structure</u> for P is a set of <u>preconditions</u> $\mathcal{S}_i \in \mathcal{S}$, one for each statement F_i of the program, and an <u>exit condition</u> $\mathcal{S}_X \subset \mathcal{S}$. The precondition

structure is consistent if $S \in \mathcal{S}_i$ implies $P_i(S) \in \mathcal{S}_X$ if $N_i(S)$ is not defined, or $P_i(S) \in \mathcal{S}_j$ if $N_i(S) = F_j$.

THEOREM (Floyd). Let P be a program on $\mathcal{S}$, let F_1 be a statement of P, let $C \subset \mathcal{S}$, and let $C' \subset \mathcal{S}$. Then P is partially correct with respect to F_1, C, and C' if and only if there exists a consistent condition structure $\{\mathcal{S}_i, \mathcal{S}_X\}$ for P with $C \subset \mathcal{S}_1$ and $\mathcal{S}_X \subset C'$.

PROOF. First, let P be partially correct. We set $\mathcal{S}_i$ equal to the set of all S_i such that (S_i, F_i) appears as T_j, $0 \leq j \leq m$, in some computation sequence $T_0, \ldots, T_m$ beginning with $T_0 = (S, F_1)$ for $S \in C$. We set $\mathcal{S}_X$ equal to the set of all $S_X = P_i(S_i)$, where S_i belongs to some $\mathcal{S}_i$ and $N_i(S_i)$ is not defined. If $S_i \in \mathcal{S}_i$ and $(S_i, F_i) = T_j$ as above, then $N_i(S_i)$ is defined if and only if $j \neq m$. In this case, T_{j+1} is defined and is equal to $(P_i(S_i), N_i(S_i))$; setting $N_i(S_i) = F_j$, we have by definition $P_i(S_i) \in \mathcal{S}_j$. If $j = m$, then by definition $P_i(S_i) \in \mathcal{S}_X$, and thus the given structure is consistent. If $S \in C$, then $(S, F_1) = T_0$ in a computation sequence, so that $S \in \mathcal{S}_1$; thus $C \subset \mathcal{S}_1$. Finally, if $S_X \in \mathcal{S}_X$, then $S_X = P'(S)$ for some $S \in C$, by the definition of the computation P'; this means that $S_X \in C'$ since P is partially correct, and thus $\mathcal{S}_X \subset C'$.

Conversely, let a consistent condition structure exist which satisfies the given conditions. Let $S \in C$ and let $T_0 = (S, F_1)$. We wish to prove that if the computation sequence beginning with T_0 is actually a finite sequence $T_0, \ldots, T_m$, where $T_m = (S'_m, F'_m)$, then $P'(S) \in C'$. Setting $F'_j = (Q_j, N'_j)$, $0 \leq j \leq m$, we have $P'(S) = Q_m(S'_m)$. We show by induction on j that $S'_j \in \mathcal{S}'_j$, where $\mathcal{S}'_j$ is the precondition associated with F'_j. For $j = 0$, $S'_0 = S \in C \subset \mathcal{S}_1 = \mathcal{S}'_0$; if $S'_{j-1} \in \mathcal{S}'_{j-1}$, then $(Q_{j-1}(S'_{j-1}), N'_{j-1}(S'_{j-1})) = T_j$, and thus $N'_{j-1}(S'_{j-1}) = F'_j$; since the structure is consistent, $S'_j = Q_{j-1}(S'_{j-1}) \in \mathcal{S}'_j$. Thus, in particular, $S'_m \in \mathcal{S}'_m$, and $N'_m(S'_m)$ is not defined. By the consistency of the structure, $Q_m(S'_m) \in \mathcal{S}_X \subset C'$; i. e., $P'(S) \in C'$. This completes the proof.

The statement of this theorem may be considerably simplified by considering subsets of $\mathcal{T} = \mathcal{S} \times P$, rather than subsets of $\mathcal{S}$. If $\{\mathcal{U}_i, \mathcal{S}_X\}$ is a consistent precondition structure for P, then the set $\mathcal{T}' \in \mathcal{T}$ defined by $(S_i, F_i) \in \mathcal{T}'$ if and only if $S_i \in \mathcal{S}_i$ is a single subset of $\mathcal{T}$ comprising all of the original $\mathcal{S}_i$. The consistency condition becomes, approximately, $P(\mathcal{T}') \subset \mathcal{T}'$, where P is now taken as an execution function. This condition must actually be modified to take account of $\mathcal{S}_X$. If $T' \in \mathcal{T}' \cap \{\underline{\text{exit}}\}$, then, writing $T' = (S, F_i)$ where $F_i = (P_i, N_i)$, we have $P_i(S) \in \mathcal{S}_X$, but $N_i(S)$ (and therefore also $P(T')$) is not defined.

<u>DEFINITION</u>. Let P be a program on $\mathcal{S}$ and let $\mathcal{T} = \mathcal{S} \times P$. Then a subset $\mathcal{T}' \subset \mathcal{T}$ is a <u>consistent</u> <u>universal</u> <u>condition</u> for P if $P(\mathcal{T}' - \{\underline{\text{exit}}\}) \subset \mathcal{T}'$ and, if $(S, F_i) \in \mathcal{T}' \cap \{\underline{\text{exit}}\}$, where $F_i = (P_i, N_i)$, then $P_i(S) = S''$ is defined. The set of all such S'' is the <u>exit</u> <u>condition</u> of $\mathcal{T}'$, and, for each fixed $F_i \in P$, the set $\mathcal{S}_i$ of all S_i with $(S_i, F_i) \in \mathcal{T}'$ is the <u>precondition</u> <u>associated</u> <u>with</u> F_i <u>by</u> $\mathcal{T}'$. A program P with a consistent universal condition $\mathcal{T}'$ may be called a <u>program</u> <u>on</u> $\mathcal{T}'$.

<u>COROLLARY</u>. Let P be a program on $\mathcal{S}$, let F_1 be a statement of P, let $C \subset \mathcal{S}$, and let $C' \subset \mathcal{S}$. Then P is partially correct with respect to F_1, C, and C' if and only if there exists a consistent universal condition $\mathcal{T}'$ for P whose exit condition is contained in C', and such that C is contained in the precondition associated with F_1 by $\mathcal{T}'$.

<u>PROOF</u>. If $\mathcal{T}'$ is consistent, we may show that its exit condition $\mathcal{S}_X$ and the preconditions $\mathcal{S}_i$ associated with the various F_i by $\mathcal{T}'$ form a consistent precondition structure. In fact, if $S \in \mathcal{S}_i$, then $(S_i, F_i) \in \mathcal{T}'$; if $(S_i, F_i) \in \{\underline{\text{exit}}\}$, where $F_i = (P_i, N_i)$, then, since $\mathcal{T}'$ is consistent, $P_i(S_i) \in \mathcal{S}_X$. If $(S_i, F_i) \notin \{\underline{\text{exit}}\}$, then $P(S_i, F_i) = (P_i(S_i), N_i(S_i)) \in \mathcal{T}'$; writing $N_i(S_i) = F_j$, this implies

$P_i(S_i) \in \mathcal{S}_j$ by the definition of the associated precondition $\mathcal{S}_j$. Thus, by the theorem, P is partially correct. Conversely, if P is partially correct, we may set $\mathcal{T}'$ equal to the set of all $T \in \mathcal{T}$ whatsoever which appear in computation sequences starting with some (S, F_1) for $S \in C$. If $T \in \mathcal{T}'$ and $T \notin \{\underline{\text{exit}}\}$, then T is not the last element of such a computation sequence, and thus P(T) is defined and $P(T) \in \mathcal{T}'$. If $T \in \{\underline{\text{exit}}\}$, then, writing $T = (S_i, F_i)$ with $F_i = (P_i, N_i)$, we have $P_i(S_i) \in \mathcal{S}_X$, and every element of $\mathcal{S}_X$ is of this form; this shows that $\mathcal{T}'$ is consistent, and also, since $P_i(S_i) = P'(S)$, where the given computation sequence started with (S, F_1), it shows that $S_X \subset C'$ (since P is partially correct) and $C \subset \mathcal{S}_1$ as before. This completes the proof.

Partial correctness may actually be used to characterize p-functions. Specifically, let $p: \mathcal{S} \to \mathcal{S}$ be an arbitrary p-function, and let n be a function which is nowhere defined. Then there exists a program P having exactly one statement $F_1 = (p, n)$. The set of all pairs (C, C') such that P is partially correct with respect to F_1, C, and C' may be used to specify p uniquely. Furthermore, this set, or, equivalently, the condition $V(C, C')$ which is true if and only if P is partially correct with respect to F_1, C, and C', is expressible entirely in terms of predicates, and may be used (as, for example, in [3]) to specify the action of p within a predicate calculus model of programming.

The convention that a program P with a consistent universal condition $\mathcal{T}'$ may be called a program <u>on</u> $\mathcal{T}'$ is fundamental for many of the definitions of objects which are associated with a program, to be defined in the sequel. For the sake of generality, each of these will be defined with respect to a program on $\mathcal{T}'$, rather than a program on $\mathcal{S}$. No generality is lost in this way, because any program P on $\mathcal{S}$ may be viewed as a program on $\mathcal{T}' = \{(S, F_1) \in \mathcal{S} \times P:$

$P_i(S)$ is defined$\}$, and this is the largest $\mathfrak{J}'$ upon which such a definition could be made. As an example of the idea of defining objects associated with a program on $\mathfrak{J}'$, we now redefine the graph of a program in these terms.

DEFINITION. Let P be a program on $\mathfrak{J}' \subset \mathfrak{S} \times P$. The directed graph of P is the directed graph whose nodes are the statements F_i of P, and such that $F_i \rightarrow F_j$ if and only if there exists $(S, F_i) \in \mathfrak{J}'$ with $N_i(S) = F_j$.

It will be seen that the directed graph of the program P on $\mathfrak{J}'$ has the same nodes, and some, but not necessarily all, of the same links, as the directed graph of the program P on $\mathfrak{S}$. Thus, for example, the directed graph of the program P on $\mathfrak{J}'$ may show us that certain statements are never executed, or that certain branches are never taken; these will not appear in the graph. This information can sometimes be helpful when proving that a program terminates.

The use of programs on $\mathfrak{J}'$ allows us to use the consistency of $\mathfrak{J}'$ to prove things about the given program, such as termination. But it does much more than this: it allows us to forget about the possibility that some of our functions P_i might be partial functions, because, in a program on $\mathfrak{J}'$, this never happens. Every P_i in such a program is defined on every state S belonging to the precondition $\mathfrak{S}_i$ associated with $F_i = (P_i, N_i)$ by $\mathfrak{J}'$. This condition is, conversely, a constraint upon our construction of $\mathfrak{J}'$ in the first place. For every P_i involving a subscripted variable, for example, unless our language contains automatic subscript range checking (such as BALGOL or the ON SUBSCRIPTRANGE feature of PL-I), we must have a precondition at that point which constrains the subscript to be within its proper range. For every P_i involving an integer variable, we must show that there is no integer overflow, and the same for floating point.

Effective Domains and Ranges

Floyd's Theorem still does not tell us how the consistency of a precondition structure is verified. In particular, it says nothing about the verification of conditions which are unchanged by an instruction. If N_i is the constant function $N_i(S) \equiv F_j$, we might have the condition $K < N$ as part of both $\mathcal{S}_i$ and $\mathcal{S}_j$, where P_i is the assignment $\{L \leftarrow GCD(I,J)\}$. In this case, we might verify the condition $K < N$ as part of $\mathcal{S}_j$ by arguing that P_i is defined as a p-function on $\prod_{x \in M} V_x$ for a set M including only the variables I, J, and L, and that, in any memory extension of this p-function to a larger memory including the variables K and N, we must therefore have $S'(K) = S(K)$ and $S'(N) = S(N)$, where $S' = p(S)$, by definition of the memory extension. The condition $K < N$ is the condition $S(K) < S(N)$, and this is therefore the same as the condition $S'(K) < S'(N)$. This argument is complicated by the fact that the condition $K < N$ actually might not be preserved; in particular, K might be a count variable, and we might have coded the GCD function in such a way as to increase K by 1 each time GCD is used in order to count the total number of times it is used. This implies that we need a general way to identify the variables used by a particular p-function and the variables set by that p-function. This will now be done for general p-functions.

<u>DEFINITION</u>. Let $p: \mathcal{S} \rightarrow \mathcal{S}'$, where $\mathcal{S} = \prod_{x \in M} V_x$ and $\mathcal{S}' = \prod_{x \in M'} V_x$, and let $\mathcal{S}'' \subset \mathcal{S}$. The <u>effective range</u> $\rho(p, \mathcal{S}'')$ is the set $\{x \in M': \exists S \in \mathcal{S}'', S' = p(S)$ is defined, but not $(S(x) = S'(x))\}$. The <u>effective domain</u> $\Delta(p, \mathcal{S}'')$ is the set $\{x \in M: \exists S_1, S_2 \in \mathcal{S}'', y \in \rho(p, \mathcal{S}''), S_1' = p(S_1)$ is defined, $S_1(z) = S_2(z)$ for all $z \neq x$, $z \in M$, but not $(S_1'(y) = S_2'(y)$ where $S_2' = p(S_2))\}$. We write $\rho(p)$ for $\rho(p, \mathcal{S})$ and $\Delta(p)$ for $\Delta(p, \mathcal{S})$.

These concepts of effective domain and effective range are extensions of the concepts of input region and output region as defined by the author in [11]. The assertion "not $(S(x) = S'(x))$" is taken to include the possibility that $S(x)$ is not defined (because $x \notin M$); it is more general than the assertion "$S(x) \neq S(x)$". Similarly, the assertion "not $(S_1'(y) = S_2'(y)$ where $S_2' = p(S_2))$" is taken to include the possibility that $p(S_2)$ may not be defined. Clearly, if $\mathcal{S}_1 \subset \mathcal{S}_2 \subset \mathcal{S}$, then $\rho(p, \mathcal{S}_1) \subset \rho(p, \mathcal{S}_2)$ and $\Delta(p, \mathcal{S}_1) \subset \Delta(p, \mathcal{S}_2)$. If $x \in M'$ but $x \in M$, then $x \in \rho(p, \mathcal{S}'')$ for arbitrary $\mathcal{S}''$.

<u>THEOREM</u>. Let $\mathcal{S}_1 = \prod_{x \in A} V_x$, let $\mathcal{S}_2 = \prod_{x \in B} V_x$, let $A' \supset A$, let $A' \cup B \supset B' \supset B$, let $\mathcal{S}_1 = \prod_{x \in A'} V_x$, let $\mathcal{S}_2' = \prod_{x \in B'} V_x$, and let $p' \colon \mathcal{S}_1' \to \mathcal{S}_2'$ be the corresponding memory extension of $p \colon \mathcal{S}_1 \to \mathcal{S}_2$. Let $\mathcal{S}_1'' \subset \mathcal{S}_1$; we may regard $\mathcal{S}_1''$ as a subset of $\mathcal{S}_1'$ by treating it as a Boolean p-function and performing a memory extension. Then $\rho(p', \mathcal{S}_1'') = \rho(p, \mathcal{S}_1'')$ and $\Delta(p', \mathcal{S}_1'') = \Delta(p, \mathcal{S}_1'')$.

<u>PROOF</u>. We write $\mathcal{S}_1^*$ for the condition $\mathcal{S}_1''$ viewed as a subset of $\mathcal{S}_1'$. Let $x \notin \rho(p, \mathcal{S}_1'')$. If $x \in B$, then, for all $S \in \mathcal{S}_1''$, $S(x)$ is defined and $S(x) = S'(x)$, where $S' = p(S)$. Let $Q \in \mathcal{S}_1^*$; by the definition of the memory extension of $\mathcal{S}_1''$, there exists $S \in \mathcal{S}_1''$ with $Q(a) = S(a)$ for all $a \in A$. Therefore, $Q(x)$ is defined, and, if $Q' = p'(Q)$, then $Q'(x) = S'(x) = S(x)$, by definition of p'. Therefore, $x \notin \rho(p', \mathcal{S}_1'')$. If $x \notin B$, then $Q'(x) = Q(x)$ and thus again $x \notin \rho(p, \mathcal{S}_1'')$; therefore, in all cases, $x \in \rho(p, \mathcal{S}_1'')$ implies $x \notin \rho(p, \mathcal{S}_1'')$. Conversely, let $x \notin \rho(p', \mathcal{S}_1'')$; then, for all $Q \in \mathcal{S}_1^*$, $Q(x)$ is defined and $Q(x) = Q'(x)$, where $Q' = p'(Q)$. If $x \notin B$, then $x \notin \rho(p, \mathcal{S}_1'')$ because $\rho(p, \mathcal{S}_1'')$ is defined as a subset of B. If $x \in B$, then each $S \in \mathcal{S}_1''$ is of the form $Q|B$ for some $Q \in \mathcal{S}_1^*$, and $S(x) = Q(x) = Q'(x) = S'(x)$, where $S' = p(S)$. Therefore $x \notin \rho(p, \mathcal{S}_1'')$; thus, in all cases, $x \notin \rho(p', \mathcal{S}_1'')$ implies $x \notin \rho(p, \mathcal{S}_1'')$, and this, together with the previous result, gives $\rho(p, \mathcal{S}_1'') = \rho(p', \mathcal{S}_1'')$.

Now let $x \in A$, $x \in \Delta(p, \mathcal{S}_1'')$; then, for all $S_1, S_2 \in \mathcal{S}_1''$ with $S_1(z) = S_2(z)$ for $z \neq x$, $z \in A$, we have $S_1'(y) = S_2'(y)$ for all $y \in$

$\rho(p, \mathcal{S}''_1)$, where $S'_1 = p(S_1)$, $S'_2 = p(S_2)$. Let $Q_1, Q_2 \in \mathcal{S}^*_1$ be such that $Q_1(z) = Q_2(z)$ for $z \neq x$, $z \in A'$, and let $S_1 = Q_1|A$, $S_2 = Q_2|A$; then $S_1(z) = S_2(z)$ for $z \neq x$, $z \in A$, and, setting $Q'_1 = p'(Q_1)$, $Q''_2 = p'(Q_2)$, we have $Q'_1(y) = S'_1(y) = S'_2(y) = Q'_2(y)$ for all $y \in \rho(p', \mathcal{S}''_1) = \rho(p, \mathcal{S}''_1)$. Therefore $x \notin \Delta(p', \mathcal{S}''_1)$. If $x \notin A$, then $S_1 = Q_1|A = Q_2|A = S_2$ and again $Q'_1(y) = S'_1(y) = S'_2(y) = Q'_2(y)$ for all $y \in \rho(p', \mathcal{S}''_1)$. Hence in all cases $x \notin \Delta(p, \mathcal{S}''_1)$ implies $x \notin \Delta(p, \mathcal{S}''_1)$. Conversely, let $x \notin \Delta(p', \mathcal{S}''_1)$; then, for all $Q_1, Q_2 \in \mathcal{S}^*_1$ with $Q_1(z) = Q_2(z)$ for $z \neq x$, $z \in A'$, we have $Q'_1(y) = Q'_2(y)$ for all $y \in \rho(p', \mathcal{S}''_1)$, where $Q'_1 = p'(Q_1)$, $Q'_2 = p'(Q_2)$. If $x \notin A$, then $x \notin \Delta(p, \mathcal{S}''_1)$, because $\Delta(p, \mathcal{S}''_1)$ is defined as a subset of A. If $x \in A$, then let $S_1, S_2 \in \mathcal{S}''_1$ be such that $S_1(z) \neq S_2(z)$ for $z \neq x$. By the definition of the memory extension of $\mathcal{S}''_1$, there exist $Q_1, Q_2 \in \mathcal{S}''_1$ with $S_1 = Q_1|A$, $S_2 = Q_2|A$, and $Q_1(z) = Q_2(z)$ for $z \notin A$. Then $Q_1(z) = Q_2(z)$ for $z \neq x$, $z \in A'$, and thus, setting $S'_1 = p(S_1)$, $S'_2 = p(S_2)$, we have $S'_1(y) = Q'_1(y) = Q'_2(y) = S'_2(y)$ for all $y \in \rho(p, \mathcal{S}''_1) = \rho(p', \mathcal{S}''_1)$. Hence in all cases $x \notin \Delta(p', \mathcal{S}''_1)$ implies $x \notin \Delta(p, \mathcal{S}''_1)$, and this, together with the previous result, gives $\Delta(p, \mathcal{S}''_1) = \Delta(p', \mathcal{S}''_1)$. This completes the proof.

This theorem shows that <u>effective domains and effective ranges are invariant under all memory extensions</u>. As a corollary, we give an upper bound to the effective domain and range of an assignment.

<u>COROLLARY</u>. Let $\{a \leftarrow b\}$ be an arbitrary assignment, let A be the set of all variables in the definition of a, and let B be the set of all variables appearing in the definition of b. Then $\Delta(\{a \leftarrow b\}) \subset A$ and $\rho(\{a \leftarrow b\}) \subset B$.

<u>PROOF</u>. The assignment $\{a \leftarrow b\}$ is defined as the memory extension of a function $p: \prod_{x \in A} V_x \to \prod_{x \in B} V_x$. By definition of the effective domain and range, $\Delta(p) \subset A$ and $\rho(p) \subset B$. By the theorem, this remains true for any memory extension of p.

Note that we cannot, in general, show that $\Delta(\{a \leftarrow b\}) = A$ or $\rho(\{a \leftarrow b\}) = B$. Thus for $p = \{X \leftarrow X\}$, we have $A = \{X\}$ and $B = \{X\}$, but $\Delta(p) = \phi$ and $\rho(p) = \phi$. In general, $\rho(p) = \phi$ will be true only for the identity function p (and $\Delta(p) = \phi$ also), whereas we will have $\Delta(p) = \phi$ whenever p is any constant assignment or combination of constant assignments.

The converse of the above theorem is true when $\mathcal{S}''$ has the finite patching property and the finite support property.

DEFINITION. Let $\mathcal{S} = \prod_{x \in M} V_x$ and let $\mathcal{S}'' \subset \mathcal{S}$. Then $\mathcal{S}''$ has the finite patching property if, given $S_1, S_2 \in \mathcal{S}$ and the finite set $M' \subset M$, the state S_3 such that $S_3(x) = S_1(x)$ for $x \in M$ and $S_3(x) = S_2(x)$ for $x \notin M$ is a member of $\mathcal{S}''$. $\mathcal{S}''$ has the finite support property if for each $S_1, S_2 \in \mathcal{S}''$ we have $\{x \in M: S_1(x) \neq S_2(x)\}$ is finite.

The finite patching property is equivalent to the "elemental" patching property in which M' is restricted to have one element. If $\mathcal{S}''$ has the finite support property, then the finite patching property is equivalent to the general patching property, in which M' no longer need be finite. If M is finite, the finite support property is obvious, and if $\mathcal{S}''$ has the finite patching property, then it must be of the form $\prod_{x \in M} V'_x$ for some choice of $V'_x \subset V_x$ for each $x \in M$, and is thus determined uniquely by the choice of the V'_x. If M is infinite, and has the finite support property, it must be a subset of $\{S \in \mathcal{S} : \{x \in M: S(x) \neq S_0(x)\}$ is finite$\}$ for some fixed $S_0 \in \mathcal{S}$; if it also has the finite patching property, then it must be the intersection of a set of this form with $\prod_{x \in M} V'_x$ for some choice of $V'_x \subset V_x$ as before. Note that a set of the form $\prod_{x \in M'} V'_x$ for $M' \subset M$ may be identified with $\prod_{x \in M} V'_x$, where V'_x has exactly one element (not zero elements!) for each $x \notin M'$.

LEMMA. Let $p: \mathcal{S} \to \mathcal{S}'$, where $\mathcal{S} = \prod_{x \in M} V_x$, $\mathcal{S}' = \prod_{x \in M'} V_x$, and let $\mathcal{S}'' \subset \mathcal{S}$ have the finite patching property and the finite support property. Let $S_1, S_2 \in \mathcal{S}''$, and let $S_1' = p(S_1)$, $S_2' = p(S_2)$. If $S_1 | \Delta(p, \mathcal{S}'') = S_2 | \Delta(p, \mathcal{S}'')$, then $S_1' | \rho(p, \mathcal{S}'') = S_2' | \rho(p, \mathcal{S}'')$.

PROOF. Let $X = \{x \in M: S_1(x) \neq S_2(x)\}$. The set X is finite because $\mathcal{S}''$ has the finite support property, and we may thus write $X = \{x_1, \ldots, x_n\}$. Let $U_i \in \mathcal{S}''$, $0 \leq i \leq n$, be defined by $U_i(z) = S_1(z)$ $(= S_2(z))$ for $z \notin X$, $U_i(x_j) = S_1(x_j)$ for $j > i$, and $U_i(x_j) = S_2(x_j)$ for $j \leq i$. These U_i are in $\mathcal{S}''$ because $\mathcal{S}''$ has the finite patching property, and $U_0 = S_1$, $U_n = S_2$; also $U_{i-1}(z) = U_i(z)$ for all $z \neq x_i$, $1 \leq i \leq n$. Let $U_i' = p(U_i)$, $0 \leq i \leq n$. If there existed $y \in \rho(p, \mathcal{S}'')$ with $U_{i-1}'(y) \neq U_i'(y)$, we would have $x_i \in \Delta(p, \mathcal{S}'')$, but this is impossible, since $S_1(x_i) \neq S_2(x_i)$ and, by hypothesis, $S_1 | \Delta(p, \mathcal{S}'') = S_2 | \Delta(p, \mathcal{S}'')$. Therefore $U_{i-1}' | \rho(p, \mathcal{S}'') = U_i' | \rho(p, \mathcal{S}'')$ for each i, $1 \leq i \leq n$, and thus $S_1' | \rho(p, \mathcal{S}'') = U_0' | \rho(p, \mathcal{S}'') = U_n' | \rho(p, \mathcal{S}'') = S_2' | \rho(p, \mathcal{S}'')$. This completes the proof.

The finite support property is necessary for this lemma; in fact, one may construct an easy counterexample whenever the finite support property does not hold. The finite patching property is not necessary; for example, the removal of exactly one state S from $\mathcal{S}''$ does not alter the conclusion of the lemma. At present, no more general necessary and sufficient condition is known.

THEOREM. Let $p: \mathcal{S} \to \mathcal{S}'$, where $\mathcal{S} = \prod_{x \in A'} V_x$, $\mathcal{S}' = \prod_{x \in B'} V_x$, and let $\mathcal{S}'' \subset \mathcal{S}$ have the finite patching property and the finite support property. Let $A = \Delta(p, \mathcal{S}'') \subset A'$, $B = \rho(p, \mathcal{S}'') \subset B'$. Then p agrees on $\mathcal{S}''$ with the memory extension $p': \mathcal{S} \to \mathcal{S}'$ of a function $f: \mathcal{S}_0 \to \mathcal{S}_0'$, where $\mathcal{S}_0 = \prod_{x \in A} V_x$, $\mathcal{S}_0' = \prod_{x \in B} V_x$.

PROOF. Let $S \in \mathcal{S}''$, let $U = S|A$, let $S' = p(S)$, and let $U' = S'|B$. We define $f: \mathcal{S}_0 \to \mathcal{S}_0'$ by setting $f(U) = U'$. This makes f well defined by the lemma; if $S_1 \in \mathcal{S}''$ is any other state with $U = S_1|A$ and if $S_1' = p(S_1)$, then $U' = S_1'|B$. Furthermore, it is clear that each $U \in \mathcal{S}_0$ is the restriction to A of some $S \in \mathcal{S}''$. If p' is the memory extension of f to a function from $\mathcal{S}$ to $\mathcal{S}'$, and $S'' \in \mathcal{S}''$, then let $p(S'') = S$ and $p'(S'') = S'$; if $z \notin B$, then $S(z) = S''(z) = S'(z)$ by definition of $B = \rho(p, \mathcal{S}'')$, whereas if $z \in B$ then $S(z) = U'(z) = S'(z)$ by definition of U'. Therefore $S = S'$, and hence p and p' agree on $\mathcal{S}''$; this completes the proof.

COROLLARY. Let $p: \mathcal{S} \to \mathcal{S}'$, where $\mathcal{S} = \prod_{x \in A'} V_x$, $\mathcal{S}' = \prod_{x \in B'} V_x$, and A' is finite. Let $A = \Delta(p) \subset A'$, $B = \rho(p) \subset B'$. Then p is the memory extension to a function from $\mathcal{S}$ to $\mathcal{S}'$ of a function $f: \mathcal{S}_0 \to \mathcal{S}_0'$, where $\mathcal{S}_0 = \prod_{x \in A} V_x$, $\mathcal{S}_0' = \prod_{x \in B} V_x$.

This allows us to specify functions by specifying their effective domains and ranges and their underlying functions f as above. It also allows us to define <u>memory restrictions</u>, which are the opposite of memory extensions.

DEFINITION. Let $p: \mathcal{S} \to \mathcal{S}'$, where $\mathcal{S} = \prod_{x \in A'} V_x$, $\mathcal{S}' = \prod_{x \in B'} V_x$, and let $M' \subset A' \cup B'$, $M' \supset \Delta(p, \mathcal{S}'')$ for some $\mathcal{S}'' \subset \mathcal{S}$. Then the <u>memory restriction</u> of p to M' on $\mathcal{S}''$ is a function $f: \mathcal{S}_0 \to \mathcal{S}_0'$, where $\mathcal{S}_0 = \prod_{x \in A} V_x$, $\mathcal{S}_0' = \prod_{x \in B} V_x$, $A = A' \cap M'$, $B = B' \cap M'$, defined for each $S_0 \in \mathcal{S}_0$ such that $S_0 = S/A$ for $S \in \mathcal{S}''$ by $f(S_0) = p(S_0)|B$.

The above corollary then says that, if A' is finite, any function p: $\prod_{x\in A'} V_x \rightarrow \prod_{x\in B'} V_x$ is the memory extension of a suitable memory restriction -- namely, one that includes $\rho(p)$. Memory restrictions of p which do not include $\rho(p)$, however, are quite common. For a program which computes a function $f(x_1, \ldots, x_n) = y$, the memory restriction of the program to $M' = \{x_1, \ldots, x_n, y\}$ will be precisely the assignment $\{y \leftarrow f(x_1, \ldots, x_n)\}$. The statement that this program computes this function is precisely the statement that this memory restriction is equal to this assignment. Two programs may be called <u>equivalent</u> <u>over</u> <u>M'</u> if their memory restrictions to M' are the same; this implies, in particular, that their effective domains are both contained in M'.

Function references in programs in which the arguments of the function are allowed to be arbitrary expressions (of the proper type) provide still another example of the use of star-extensions. If t_i is the type of the i-th argument of a function $f(x_1, \ldots, x_n)$, and $p_i: \mathcal{S}_i \rightarrow t_i$ are the p-functions corresponding to certain expressions x_i of type t_i, then $f^*(p_1, \ldots, p_n)$ is the p-function corresponding to the use of f with arguments $p_1, \ldots, p_n$. This may be the star-extension in either the static or the dynamic sense. An expression of this type is normally treated in programming languages as equivalent to a single variable for the purposes of combining quantities into expressions. Just as we have formed $g^*(e_1,e_2) = e_1+e_2$, for example, where g is the ordinary addition function of two variables and e_1 and e_2 are p-functions, so we can likewise form $e_1+f(x_1, \ldots, x_n) = g^*(e_1, f^*(p_1, \ldots, p_n))$, provided that the types of all the given expressions are properly connected with each other, since $f^*(p_1, \ldots, p_n)$ is a function whose domain and range are such that it may take the place of e_2 in the above construction.

Consistency Calculations

We now show how to calculate consistency of state conditions using the concepts of effective domain and range. For this purpose we need a generalization of the Composition Theorem which we introduced in [11]. This theorem gives relations between the effective domains and ranges of two given functions and the effective domain and range of the composition of these two functions. Our first result is as follows.

THEOREM. Let $f: \mathcal{S}_1 \to \mathcal{S}_1$ and $g: \mathcal{S}_1 \to \mathcal{S}_2$ and let $f \circ g = h: \mathcal{S}_1 \to \mathcal{S}_2$, where $\mathcal{S}_1$ has the finite patching property and the finite support property, and further suppose that $\rho(f, \mathcal{S}_1) \cap \Delta(g, \mathcal{S}_1) = \phi$. Then, for any $y \in \rho(g, \mathcal{S}_1')$, we have $S_2(y) = S_2'(y)$, where $S_2 = g(S_1)$ and $S_2' = h(S_1)$ for any $S_1 \in \mathcal{S}_1$.

PROOF. Let $S_1 \in \mathcal{S}_1$ and let $S_1' = f(S_1)$. Let $x \in \Delta(g, \mathcal{S}_1)$; then $x \notin \rho(f, \mathcal{S}_1)$, and, since $S_1 \in \mathcal{S}_1$ and $S_1' \in \mathcal{S}_1$, we have $\mathcal{S}_1(x) = S_1'(x)$. In this way we see that $S_1|\Delta(g, \mathcal{S}_1) = S_1'|\Delta(g, \mathcal{S}_1)$, and we may now apply the lemma of the preceding section, obtaining $S_2|\rho(g, \mathcal{S}_1) = S_2'|\rho(g, \mathcal{S}_1)$, where $S_2 = g(S_1)$ and $S_2' = g(S_1') = g(f(S_1)) = h(S_1)$. This completes the proof.

We shall now apply this theorem to the preservation of a state condition when an instruction is executed. Let $\mathcal{S}' \subset \mathcal{S}$ be a state condition; then $\mathcal{S}'$, as mentioned before, may be viewed as a function $f: \mathcal{S} \to \{\underline{\text{true}}, \underline{\text{false}}\}$, with $f(S) = \underline{\text{true}}$ if and only if $S \in \mathcal{S}'$. As a Boolean p-function of this form, $\mathcal{S}'$ <u>has an effective domain</u> $\Delta(\mathcal{S}')$; if $\mathcal{S} = \prod_{x \in M} V_x$, then $\Delta(\mathcal{S}') = \{x \in M: \exists\ S_1 \in \mathcal{S}', S_2 \in \mathcal{S} - \mathcal{S}'$, with $S_1(z) = S_2(z)$ for all $z \neq x\}$. The effective domain of a state condition which is defined in terms of a set of variables M' is contained in M', because it is by definition a memory extension of a Boolean p-function on $\mathcal{S}' = \prod_{x \in M'} V_x$. We have $\Delta(\mathcal{S}' \cup \mathcal{S}'') \subset \Delta(\mathcal{S}') \cup \Delta(\mathcal{S}'')$ and $\Delta(\mathcal{S}' \cap \mathcal{S}'') = \Delta(\mathcal{S}') \cap \Delta(\mathcal{S}'')$. For the more general notion of effective domain, $\Delta(\mathcal{S}', \mathcal{S}'')$, where $\mathcal{S}'' \subset \mathcal{S}$, we have $\Delta(\mathcal{S}', \mathcal{S}'') \subset \Delta(\mathcal{S}')$; also, we always have $\Delta(\mathcal{S}', \mathcal{S}') = \phi$.

Now let $F_i = (P_i, N_i)$ be a statement of a program P on $\mathcal{J}' \subset \mathcal{S} \times P$ and let $\mathcal{S}_i$ be the associated precondition, so that $P_i: \mathcal{S}_i \to \mathcal{S}_i$. Usually $\mathcal{S}_i$ will be expressible as the intersection of a collection of state conditions, among which will be some which are not changed by F_i. Letting $\mathcal{S}'$ be a typical such condition, we wish to prove the consistency of including $\mathcal{S}'$ in the collection of state conditions whose intersection is the precondition $\mathcal{S}_j$ associated with some F_j in the range of N_i. To do this we must show that $S \in \mathcal{S}'$ implies $P_i(S) \in \mathcal{S}'$.

<u>THEOREM</u>. Let $f: \mathcal{S} \to \mathcal{S}$, where $\mathcal{S}$ has the finite patching property and the finite support property, let $\mathcal{S}' \subset \mathcal{S}$, and suppose that $\rho(f, \mathcal{S}') \cap \Delta(\mathcal{S}') = \phi$. Then $S \in \mathcal{S}'$ implies $f(S) \subset \mathcal{S}'$ whenever $f(S)$ is defined.

<u>PROOF</u>. Let $S_0 \in \mathcal{S}'$, let $S' = f(S_0)$, and define $g: \mathcal{S} \to \mathcal{S}$ by $g(S) = f(S)$ for $S \in \mathcal{S}'$ and $g(S) = S'$ for $S \notin \mathcal{S}'$. Then $\rho(g, \mathcal{S}) = \rho(g, \mathcal{S}') = \rho(f, \mathcal{S}')$; also f may be replaced by g in the conclusion. It is thus sufficient to replace the hypothesis by $\rho(f, \mathcal{S}) \cap \Delta(\mathcal{S}') = \phi$, and under these conditions $\rho(f, \mathcal{S}) \cap \Delta(\mathcal{S}', \mathcal{S}) = \phi$ since $\Delta(\mathcal{S}', \mathcal{S}) \subset \Delta(\mathcal{S}')$. The theorem now follows from the preceding theorem by considering $\mathcal{S}'$ as a Boolean p-function; the range of $\mathcal{S}'$, taken as a p-set, is a cartesian product of one set, which is the effective range of $\mathcal{S}'$, and the conclusion of the preceding theorem thus reads $S_2 = S_2'$, where S_2 is the truth value of $S_1 \in \mathcal{S}'$ and S_2' is the truth value of $f(S_1) \in \mathcal{S}'$. The proof is completed by obvious changes of notation.

This theorem implies, in particular, that a state condition is preserved whenever the variables involved in the state condition do not appear on the left side of an executed assignment. Thus, for example, if $\{I+J<K\}$ is a state condition associated with the assignment $\{L \leftarrow J-K\}$ as part of a precondition, then $\{I+J<K\}$ may be used as part of the precondition for the next statement (provided that there is no other way to get to this statement directly), since the variable L does not occur in this condition. Here the assignment $\{L \leftarrow J-K\}$ may be replaced by any p-function whose effective range is $\{L\}$, even if this effective range is with respect to the condition $\{I+J < K\}$ itself.

More complex functions P_i occurring in a program may consist of several assignments $A_1, \ldots, A_k$ performed one after the other. In this case, $P_i : \mathcal{S} \rightarrow \mathcal{S}$ is the composition $A_1 \circ \ldots \circ A_k$. This fact is itself a useful byproduct of the way in which we have defined assignments; the determination of the result of performing two successive assignments from their character string form is quite complex and involves a large number of special cases. The following theorem gives an upper bound on the effective domain and range of such a composition; we state it for general p-functions.

<u>THEOREM</u>. Let $f: \mathcal{S}_1 \rightarrow \mathcal{S}_2'$ and $g: \mathcal{S}_2 \rightarrow \mathcal{S}_3$ and let $f \circ g = h: \mathcal{S}_1 \rightarrow \mathcal{S}_3$. Let $\mathcal{S}_1' \subset \mathcal{S}_1$, $\mathcal{S}_2' \subset \mathcal{S}_2$, and suppose that $S_1 \in \mathcal{S}_1'$ implies $f(S_1) \in \mathcal{S}_2'$ whenever $f(S_1)$ is defined. Then $\Delta(h, \mathcal{S}_1') \subset \Delta(f, \mathcal{S}_1') \cup \Delta(g, \mathcal{S}_2')$ and $\rho(h, \mathcal{S}_1') \subset \rho(f, \mathcal{S}_1') \cup \rho(g, \mathcal{S}_2')$.

Note that if $\mathcal{S}_1 = \prod_{x \in M_1} V_x$, $\mathcal{S}_2 = \prod_{x \in M_2} V_x$, $\mathcal{S}_3 = \prod_{x \in M_3} V_x$, where M_1, M_2 and M_3 are not necessarily the same, there is more that we can say. For example, $\Delta(f, \mathcal{S}_1') \subset M_1$ and $\Delta(g, \mathcal{S}_2') \subset M_2$, so that the theorem implies $\Delta(h, \mathcal{S}_1') \subset M_1 \cup M_2$; but clearly, by definition, $\Delta(h, \mathcal{S}_1') \subset M_1$. Similarly, $\rho(f, \mathcal{S}_1') \subset M_2$ and $\rho(g, \mathcal{S}_2') \subset M_3$, then the theorem implies $\rho(h, \mathcal{S}_1') \subset M_2 \cup M_3$, but by definition we have $\rho(h, \mathcal{S}_1') \subset M_3$. Note also that we may extend f, g, and h to p-functions f', g', and h' on $\mathcal{S} = \prod_{x \in M} V_x$ where $M \supset M_1 \cup M_2 \cup M_3$, and the effective domains and ranges will, as noted earlier, be preserved under these memory extensions; but this fact does not reduce the proof of the theorem to the case $M_1 = M_2 = M_3$, because we may have $h' \neq f' \circ g'$ and, in particular, we do not necessarily have $\rho(f' \circ g', \mathcal{S}_1') \subset M_3$. (For example, the composition of $\{Y \leftarrow X\}$ and $\{Z \leftarrow Y\}$ is $\{Z \leftarrow X\}$ if we take $M_1 = \{X\}$, $M_2 = \{Y\}$, and $M_3 = \{Z\}$, but not if we take these as p-functions on $\mathcal{S} = \prod_{x \in M} V_x$ for $X, Y, Z \in M$.)

PROOF. Let $\mathcal{S}_1 = \prod_{x \in M_1} V_x$, $\mathcal{S}_2 = \prod_{x \in M_2} V_x$, $\mathcal{S}_3 = \prod_{x \in M_3} V_x$. Let $S \in \mathcal{S}_1'$ and let $S' = f(S)$ be defined; then $S' \in \mathcal{S}_2'$ and, if $S'' = g(S')$ is defined, we have $S'' = h(S)$. Let $x \notin \rho(f, \mathcal{S}_1')$, $x \notin \rho(g, \mathcal{S}_2')$; then $S'(x)$ is defined and $S'(x) = S(x)$, and likewise $S''(x)$ is defined and $S''(x) = S'(x) = S(x)$. Thus $x \notin \rho(h, \mathcal{S}_1')$, which shows that $\rho(h, \mathcal{S}_1') \subset \rho(f, \mathcal{S}_1') \cup \rho(g, \mathcal{S}_2')$. Now let $x \notin \Delta(f, \mathcal{S}_1')$, $x \notin \Delta(g, \mathcal{S}_2')$, and let $U, V \in \mathcal{S}_1'$ be such that $U(z) = V(z)$ for all $z \neq x$. If $y \in \rho(f, \mathcal{S}_1')$, then $U'(y) = V'(y)$, where $U' = f(U)$, $V' = f(V)$, since $x \notin \Delta(f, \mathcal{S}_1')$. If $y \notin \rho(f, \mathcal{S}_1')$, $y \neq x$, then $U'(y) = U(y) = V(y) = V'(y)$. Hence $U'(y) = V'(y)$ for all $y \neq x$, and, in addition, $U' = V'$ if $x \in \rho(f, \mathcal{S}_1')$. By exactly the same arguments, if $U'' = g(U') = g(f(U)) = h(U)$ and $V'' = g(V') = g(f(V)) = h(V)$, we have $U''(y) = V''(y)$ for all $y \neq x$, and, in fact, $U'' = V''$ when $x \in \rho(g, \mathcal{S}_2')$ -- and also when $x \in \rho(f, \mathcal{S}_1')$, because in this case $U'' = g(U') = g(V') = V''$ since $U' = V'$. But $\rho(h, \mathcal{S}_1') \subset \rho(f, \mathcal{S}_1') \cup \rho(g, \mathcal{S}_2')$, as was shown above, and this means that $U'' = V''$ whenever $x \in \rho(h, \mathcal{S}_1')$. Hence $x \notin \Delta(h, \mathcal{S}_1')$, and thus $\Delta(h, \mathcal{S}_1') \subset \Delta(f, \mathcal{S}_1') \cup \Delta(g, \mathcal{S}_2')$. This completes the proof.

COROLLARY. Let $f_i: \mathcal{S}_{i-1} \rightarrow \mathcal{S}_i$, $1 \leq i \leq n$, and let $f = f_1 \circ \ldots \circ f_n$. Let $\mathcal{S}_{i-1}' \subset \mathcal{S}_{i-1}$, $1 \leq i \leq n$, and suppose that $S_{i-1} \in \mathcal{S}_{i-1}'$ implies $f_i(S_{i-1}) \in \mathcal{S}_i'$ whenever $f_i(S_{i-1})$ is defined. Then $\Delta(f, \mathcal{S}_0') \subset \bigcup_{i=1}^{n} \Delta(f_i, \mathcal{S}_{i-1}')$ and $\rho(f, \mathcal{S}_0') \subset \bigcup_{i=1}^{n} \rho(f_i, \mathcal{S}_{i-1}')$.

In particular, if the f_i are assignments, the same principle holds as before in proving that a given state condition is preserved by $f = f_1 \bullet \ldots \bullet f_n$; namely, this will be the case whenever the total collection of variables set by the f_i is disjoint from the collection of variables involved in the given state condition.

In the general case, it is helpful to distinguish the current value of a variable as we follow a path from its initial value. This may be done by denoting the current value of J, for example, by 1J, or the current value of KAPPA by 1KAPPA. As an example, consider the following sequence of statements; with a condition given at the beginning and at the end:

```
CONDITION J+1=I*I      (1)  1I=I     1J=J
I ← I-1                (2)  1I=I-1    1J=J
J ← J-2*I              (3)  1I=I-1    1J=J-2*(I-1)
I ← I*I                (4)  1I=(I-1)*(I-1)    1J=J-2*(I-1)
CONDITION J = I        (5)  (I-1)*(I-1)=J-2*(I-1)
```

Suppose we are asked to verify that if we start at the top of this sequence, under the given imposed condition, we arrive at the bottom with J=I. We write the statements (1), (2), (3), (4), and (5) in that order; in each case we use current values. For example, in statement (3), we are setting J to J minus twice the <u>current</u> value of I, or what we have called 1I, which is I-1. The expression J-2*(I-1) thus becomes the current value of J, or 1J. At the end, we need only verify that $(i-1)^2 = j-2(i-1)$ follows from $j+1 = i^2$; setting $j = i^2-1$, we obtain $(i-1)^2 = i^2-1-2(i-1) = i^2-2i+1$, which is true.

There is another method of verifying a sequence like this, which is due to London [3] and is known as <u>back substitution</u>. In this case, we start from the final condition and work backwards, modifying the condition each time, rather than the values of the variables. When we come to an assignment v ← e, where v is a variable and e is an expression, we replace each occurrence of v, in the current condition, by e. Thus in the above case we would write

```
CONDITION J+1=I*I
I ← I-1                  (4)  J-2*(I-1) = (I-1)*(I-1)
J ← J-2*I                (3)  J-2*I = I*I
I ← I*I                  (2)  J = I*I
CONDITION J = I          (1)  J = I
```

in which the statements (1), (2), (3), and (4) are written in that order. The end result is exactly the same as before. Both the forward and the backward methods are extensively discussed in King [5] and Good [4].

It may happen that a statement P_i in a program is itself the computation of another program with respect to some starting statement. The second program may be a subroutine, or it may merely be a section of the first. In this case, we may assume that this second program already has a consistent precondition structure; the following theorem then gives an upper bound on the effective domain and range of such a computation.

THEOREM. Let P be a program on $\mathfrak{J}' \subset \mathfrak{X} \times P$, let F_1 be an arbitrary statement of P, let P' be the computation of P with respect to F_1, and for each $F_i \in P$ let $\mathfrak{X}_i$ be the associated precondition. Then $\rho(P', \mathfrak{X}_1)$ is contained in the union of all $\rho(P_i, \mathfrak{X}_i)$, while $\Delta(P', \mathfrak{X}_1)$ is contained in the union of all $\Delta(P_i, \mathfrak{X}_i)$ and all $\Delta(N_i, \mathfrak{X}_i)$.

We note that the inclusion in this theorem of the effective domains of the next-statement functions N_i is essential. For example, consider the ALGOL assignment i:=<u>if</u> m=n <u>then</u> 0 <u>else</u> 1. The effective domain of this assignment clearly includes m and n. The assignment can, however, be realized by the following three-step program:

1. If m = n, go to step 3.
2. Set i = 1 and exit ($N_2(S)$ is nowhere defined).
3. Set i = 0 and exit ($N_3(S)$ is nowhere defined).

In this program, the functions P_i do not include m and n in their effective domains. In particular, P_1 is the identity, and its effective domain is therefore the null set. However, m and n are in the effective domain of the function N_1.

<u>PROOF</u>. It is sufficient to assume that P is complete, because the given computation of P is the same as the corresponding computation of the completion of P, and similarly the P_i and the values of $\Delta(N_i, \mathcal{S}_i)$ are preserved when we take the completion of P.

Let $S \in \mathcal{S}_1$ and let $P'(S)$ be defined. This implies that there exists a computation sequence $T_0, \ldots, T_k$ with $T_0 = (S, F_1)$; setting $T_i = (S_i, F'_i)$, where $F'_i = (P'_i, N'_i)$, we have $S_k = P'(S)$. Let $x \notin \rho(P_i, \mathcal{S}_i)$ for all i; we must show $S_k(x) = S(x)$. In fact, we show $S_i(x) = S(x)$ inductively for all i, showing at the same time, as in Floyd's Theorem, that S_i is in the precondition $\mathcal{S}'_i$ associated with F'_i by $\mathcal{J}'$. These statements are clearly true for i = 0; we assume they are true for $i = j < k$. Then $S_j \in \mathcal{S}'_j$ and $S_j(x) = S(x)$; since $x \notin \rho(P'_j, \mathcal{S}'_j)$, and since $P'_j(S_j) = S_{j+1}$, we have $S_{j+1}(x) = S_j(x) = S(x)$, and also, by consistency, $S_{j+1} \in \mathcal{S}'_{j+1}$ for $j < k$. This completes the induction and thus $x \notin \rho(P', \mathcal{S}_1)$; therefore $\rho(P', \mathcal{S}_1)$ is contained in the union of all the $\rho(P_i, \mathcal{S}_i)$. Now let $x \notin \rho(P_i, \mathcal{S}_i)$ and $x \notin \Delta(N_i, \mathcal{S}_i)$ for all $F_i = (P_i, N_i)$, and let $S, S' \in \mathcal{S}_1$ be such that $S(z) = S'(z)$ for all $z \neq x$. Let $T_0 = (S, F_1)$, $T'_0 = (S', F_1)$, and consider the computation sequences $T_0, T_1, \ldots$ and $T'_0, T'_1, \ldots$, defined by $T_i = P(T_{i-1})$, $T'_i = P(T'_{i-1})$, for as long as these are defined. We set $T_i = (S_i, G_i)$ and $T'_i = (S'_i, G'_i)$. We prove inductively on i the assertion that the following statements are true if T_i is defined: (a) T'_i is defined; (b) $G_i = G'_i$; (c) S_i, $S'_i \in \mathcal{S}'_i$, where $\mathcal{S}'_i$ is the precondition associated with $G_i = G'_i$; (d) $S_i(z) = S'_i(z)$ for all $z \neq x$; (e) if x is in the union, over all $j < i$, of all $\rho(P'_j, \mathcal{S}'_j)$, where $G_j = (P'_j, N'_j)$, then $S_i = S'_i$. These conditions are clearly true for $i = 0$, including the last which is vacuous. Suppose they are true for $i = k$, and that T_{k+1} is defined; then $T_{k+1} = P(T_k)$, so that $S_{k+1} = P'_k(S_k)$ and $G_{k+1} = N'_k(S_k)$. In particular, G_k is not the exit statement, and since $G_k = G'_k$, we may define $S'_{k+1} = P'_k(S'_k)$ and $G'_{k+1} = N'_k(S'_k)$, which proves (a). Since $T_k(z) = T'_k(z)$ for all $z \neq x$, and $x \notin \Delta(N'_k, \mathcal{S}'_k)$, we have $N'_k(S_k) = N'_k(S'_k)$, i. e., $G_{k+1} = G'_{k+1}$, which proves (b); (c) then follows from the consistency of the precondition structure. If $z \in \rho(P'_k, \mathcal{S}'_k)$, then, since $P'_k(S_k) = S_{k+1}$ and $P'_k(S'_k) = S'_{k+1}$, and since $x \notin \Delta(P'_k, \mathcal{S}'_k)$ with S_k, $S'_k \in \mathcal{S}'_k$, we have $S_{k+1}(z) = S'_{k+1}(z)$. If $z \notin \rho(P'_k, \mathcal{S}'_k)$, $z \neq x$, we have $S_{k+1}(z) = S_k(z)$, $S'_{k+1}(z) = S'_k(z)$, and since $S_k(z) = S'_k(z)$ by the induction hypothesis, we have $S_{k+1}(z) = S'_{k+1}(z)$. Hence in all cases $S_{k+1}(z) = S'_{k+1}(z)$ for $z \neq x$, which proves (d). If x is in the union over all $j < k+1$ as given in condition (e), then either x is in this union over all $j < k$, in which case $S_k = S'_k$ and $S_{k+1} = P'_k(S_k) = P'_k(S'_k) = S'_{k+1}$, or else $x \notin \rho(P'_k, \mathcal{S}'_k)$, in which case the above argument shows that likewise $S_{k+1} = S'_{k+1}$, proving (e) and completing the induction. In particular, if one computation sequence terminates at T_m, then the other terminates at T'_m, where $S_m = P'(S)$ and $S'_m = P'(S')$. Conditions (d) and (e) for $i = m$ now show that $S_m(z) = S'_m(z)$ for $z \neq x$, and if $x \in \rho(P', \mathcal{S}_1)$ -- which means, as proved above, that x is in the union of all $\rho(P_i, \mathcal{S}_i)$, and, in particular, in the union of all $\rho(P'_i, \mathcal{S}'_i)$ -- then $S_m = S'_m$. Thus $x \notin \Delta(P', \mathcal{S}_1)$, and this shows, finally, that $\Delta(P', \mathcal{S}_1)$ is contained in the union of all $\Delta(P_i, \mathcal{S}_i)$ and all $\Delta(N_i, \mathcal{S}_i)$, completing the proof.

The above theorem does not prove as much as we might like. Specifically, it allows the effective domain and range of a computation to include the effective domains and ranges, respectively, of certain statements in the given program which make reference to temporary variables and registers. For the effective range, this is unavoidable; temporary values will normally be set in this way, and, as long as they really are temporary -- i. e., they are not variables whose values are needed later on -- this does not affect the result of the program. For the effective domain, it is a bit surprising; we should not like to see garbage entering our calculations and affecting our result. This, however, is normally taken care of by examining the condition structure of the given program. In any event, if this program is used as a subroutine of some other program, the consistency calculations in the second program will be affected only by considerations of the effective range in the subroutine. It is also true, that given any program whatsoever we may, theoretically, introduce an irrelevant assignment $\{U \leftarrow V\}$, for variables U and V which are not referenced anywhere else in the program, and V then becomes part of the effective domain of any computation of the program provided that the assignment $\{U \leftarrow V\}$ is executed in every computation sequence.

Global Conditions and Sufficient Substructures

The specification of a state condition at <u>every</u> point in the program is actually not necessary. We now consider ways of making consistency easier to prove.

Suppose first that associated with each statement F_i we have a state condition $\mathcal{S}_i = \mathcal{S}_{i1} \cap \ldots \cap \mathcal{S}_{ik_i}$. Then it is necessary only to show, for each $\mathcal{S}_{ij}$, that $S \in \mathcal{S}_k$ and $N_k(S) = F_i$ implies $P_k(S) \in \mathcal{S}_{ij}$. This is true because under these conditions $P_k(S)$ will be in all $\mathcal{S}_{ij}$, for fixed i, and will thus be in their intersection, which is precisely $\mathcal{S}_i$. This type of argument is similar to multiple mathematical induction, in which several propositions are being proved simultaneously and all of them may be assumed to hold in the case <u>k</u> when proving that each of them holds separately in the case <u>k</u>+1.

It may happen that a single condition $\mathcal{S}'$ occurs in this way as $\mathcal{S}_{ij_i}$ for <u>all</u> i (and some j_i depending on i). Such a condition may be called a <u>global condition</u>. Global conditions in programs are very common. Usually they are conditions on variables which are not changed at all during the course of a computation. In such a case, the global condition is automatically true over the entire program provided it is true at the beginning, and it does not need to be verified separately at each stage. It can, however, be used in the verification of other conditions. If we have set K equal to 1, for example, prior to a section of a program, and we never change K within that section, then $\{K=1\}$ is a global condition for the given section; it does not need to be verified separately for every statement of the program, but if, for example, we have $\{N \leftarrow N+K\}$ as an assignment in this program, we may consider it equivalent to $\{N \leftarrow N+1\}$. This is particularly important in the case of subscripted variables, because when we use such a variable our p-functions will not even be defined unless the subscripts are within their proper ranges. This fact is often expressed as a global condition.

Not all global conditions are of the form discussed above. We may, for example, have a global condition of the form $I > 0$, where I is given an initial value greater than zero and the only statements of the program which change I either increase I or give it new constant values greater than zero. We may also have conditions which hold because of some initialization carried out at the beginning of the program; these conditions are not quite global, because they do not hold throughout the entire initialization section of the program. The considerations discussed above, however, still hold when applied to the remainder of the program. Generalizing still further, we may have a condition which holds globally, but such that the proof of this fact involves recourse to the other preconditions which we have constructed for the program. As remarked above, there is no "circularity" or other impropriety in the use of one condition to verify another, followed by the use of the second to verify the first, just as this would be permissible in a multiple mathematical induction.

We now pass to the consideration of <u>substructures</u> of a condition structure, i. e., the association of preconditions with only the "important" statements in a program.

DEFINITION. Let P be a program on $\mathcal{T}' \subset \mathcal{S} \times P$, and let $\{\mathcal{S}_i, \mathcal{S}_X\}$ be the associated consistent precondition structure. Any subset of this structure which includes $\mathcal{S}_X$ is a (precondition) <u>substructure</u> for P. If $\mathcal{U}$ is a substructure, a <u>control path</u> of $\mathcal{U}$ is a path in the graph of P whose initial and final nodes are associated with preconditions belonging to $\mathcal{U}$.

To show that a precondition structure is consistent, it is necessary only to verify a condition along an arbitrary link $F_i \rightarrow F_j$ of the graph of the program. To show that a substructure is consistent, it is necessary to verify conditions along control paths. Specifically, we have the following definition and theorem.

<u>DEFINITION</u>. Let $\mathcal{U}$ be a substructure for the program P on $\mathcal{S}$. Let $F'_0, \ldots, F'_m$ be a control path of $\mathcal{U}$. Let $\mathcal{S}'$ be the precondition associated with F'_0, and let $\mathcal{S}''$ be the precondition associated with F'_m. Let $\mathcal{S}'_i$, $0 \leq i \leq m$, be defined inductively by setting $\mathcal{S}'_0 = \mathcal{S}'$ and $\mathcal{S}'_i = \{P_{i-1}(S_{i-1})\colon S_{i-1} \in \mathcal{S}'_{i-1}$ and $N_{i-1}(S_{i-1}) = F'_i\}$. Then the given control path is <u>consistent</u> if $\mathcal{S}'_m \subset \mathcal{S}''$ and if, for each $S_{i-1} \in \mathcal{S}'_{i-1}$ $(1 \leq i \leq m)$ for which $N_{i-1}(S_{i-1})$ is defined, so is $P_{i-1}(S_{i-1})$. The substructure $\mathcal{U}$ is <u>consistent</u> if all of its control paths are consistent.

<u>THEOREM</u>. Let P be a program on $\mathcal{S}$, let P' be the computation of P with respect to $F_1 \in P$, and let $\mathcal{S}' \subset \mathcal{S}$, $\mathcal{S}'' \subset \mathcal{S}$. Then P is partially correct with respect to F_1, $\mathcal{S}'$, and $\mathcal{S}''$ if and only if there exists a consistent substructure $\mathcal{U}$ for P which contains the precondition $\mathcal{S}_1$, with $\mathcal{S}' \subset \mathcal{S}_1$, and preconditions $\mathcal{S}_j$ for each F_j such that N_j is not total, with $S_j \in \mathcal{S}''$ for S_j in any such $\mathcal{S}_j$ whenever $N_j(S_j)$ is not defined.

<u>PROOF</u>. (a) $\rightarrow$ (b). This follows directly from the corollary to Floyd's Theorem, since an entire precondition structure may be regarded as a substructure of itself; the distinct control paths of this "substructure" are exactly the links in the graph of the program, and the consistency condition for such a path is precisely the consistency condition for the original precondition structure.

(b) $\rightarrow$ (a). Let $S' \in \mathcal{S}'$ be such that $P'(S')$ is defined. This means that the sequence $T_0 = (S', F_1)$, $T_i = P(T_{i-1})$ for $i > 0$, terminates at some $T_k \in \{exit\}$. Let $i_0 < \ldots < i_m$ be the values of i in this sequence such that $T_i = (S_i', F_i')$ for F_i' associated with a precondition $\mathcal{S}_i'$ in the given substructure. From the condition on the substructure, we clearly have $i_0 = 0$, $i_m = k$. Furthermore, for each $T_{i_j}, T_{i_j+1}, \ldots, T_{i_{j+1}}$, $0 \leq j < m$, the corresponding F'_{i_j}, $F'_{i_j+1}, \ldots, F'_{i_{j+1}}$ is a control path for the given substructure. We show inductively that each $S'_{i_j} \in \mathcal{S}'_{i_j}$; by hypothesis, $S'_{i_0} = S' \in \mathcal{S}' \subset \mathcal{S}_1 = \mathcal{S}'_{i_0}$. Suppose, inductively, that $T_{i_j} = (S'_{i_j}, F'_{i_j})$ where $S'_{i_j} \in \mathcal{S}'_{i_j}$; the condition of consistency on the control path F'_{i_j}, $F'_{i_j+1}, \ldots, F'_{i_{j+1}}$ then says precisely that $S'_{i_{j+1}} \in \mathcal{S}'_{i_{j+1}}$. Thus, in particular, $T_k = T_{i_m} = (S'_{i_m}, F'_{i_m})$ with $S'_{i_m} \in \mathcal{S}'_{i_m}$, and, since $T_k \in \{exit\}$, we have by hypothesis $P'(S') = S'_{i_m} \in \mathcal{S}'_{i_m} \subset \mathcal{S}''$. This completes the proof.

This theorem applies to an arbitrary substructure; in general, however, it will not be easy to apply it arbitrarily, because the consistency of a substructure requires the consistency of _every_ control path, and there may be an infinite number of these. In particular, the application of the theorem to the substructure consisting of the initial and terminal nodes alone, within the graph of the program, amounts to the obvious but relatively useless statement that one may verifying a program by looking at every possible computation sequence. What we need is a condition on substructures which guarantees that the total number of distinct control paths be finite. The following definition and theorem are based on an observation made by several authors which seems to have first appeared in print in King's thesis [5].

DEFINITION. The substructure $\mathcal{U}$ for the program P is _sufficient_ if we obtain an acyclic graph by removing from the graph of

P the nodes associated with preconditions of $\mathcal{U}$ and all links involving these nodes.

THEOREM. There are only a finite number of distinct control paths of a sufficient substructure.

PROOF. Each of the control paths, aside from the ends of the path, involves distinct nodes of the graph, for if any two of these nodes were the same, then they and all nodes occurring between them in the path would constitute a directed cycle in the graph of the program after the nodes corresponding to the substructure were removed, contradicting the definition above. Each control path is therefore indexed by its beginning, its end, and a distinct finite subset of the finite graph of the program, and the total number of such paths is therefore finite, completing the proof. We remark that a substructure which is not sufficient must have an infinite number of control paths provided that from every statement F_i of the program there is a path to a termination statement and similarly there is a path from the start statement of the program to F_i. This is a reasonable condition, since any F_i not satisfying these conditions may be eliminated from the program if it is to be proved correct. Thus, in a sense, the condition on a substructure of being sufficient is "best possible" at this point.

There are many ways of obtaining a sufficient substructure without drawing the graph of the program, such as:

(1) Associating a precondition with every <u>backward</u> <u>transfer</u> in the program.

(2) Associating a precondition with every <u>label</u> in the program (this does not work in assembly languages if constructions involving a current location symbol * -- such as *+3 -- are allowed).

(3) Associating a precondition with every <u>junction</u> <u>statement</u>

in the program, a junction statement being one which can be reached from more than one immediately preceding statement (this always gives only a finite number of control paths which need to be looked at, but is sufficient only in the presence of the graph connectedness condition mentioned above).

(4) Associating a precondition with every <u>branch</u> (or conditional statement) in the program; the same considerations hold as in the previous case. Note that by associating a precondition with every junction statement <u>and</u> with every branch, we obtain a bonus: every statement not a junction or a branch is contained in exactly one control path of the resulting substructure, again provided that the graph is connected as defined above. Thus in a sense there is no wasted work in this case.

Termination of Algorithms

We now turn to the question of proving that our programs, or what is the same, our computation sequences, terminate. It is clear that this will always be the case when there are no directed cycles in the graph of the program. A directed cycle in the graph of a program is called a loop. Most loops in programs have "controlled variables" which advance through a certain range until they reach their goal. For a loop to be executed from I = 1 to I = 100, we may consider the state conditions $\{I=1\}$, $\{I=2\}$, ..., $\{I=100\}$; as the program progresses, the states appearing in the computation sequence are progressively in each of these subsets of $\mathcal{S}$, and this fact may be used to show that the program "progresses normally." If the program contains two loops, one after the other, and each one involving the same controlled variable I, then our conditions must also be conditions of the program counter λ, which restrict λ to be within the first loop or within the second.

We are thus led to the general concept of a collection of subsets of $\mathfrak{T} = \mathcal{S} \times P$, together with an order relation on this collection. In the above example, we would have had $\{I=j\} \geq \{I=k\}$ for $j \geq k$, and the condition that the program progresses normally as examined above becomes the statement: if $T \in \mathfrak{T}_i$ and $P(T) \in \mathfrak{T}_j$, then we must have $\mathfrak{T}_j \geq \mathfrak{T}_i$. (We note, for later consideration, that this is still not enough, because we might have $\mathfrak{T}_j = \mathfrak{T}_i$.) The collection $\mathfrak{T}_i$ of the example is finite, but this is clearly not necessary; for example, we might have taken the conditions $\{I=k\}$ for all natural numbers k, in descending order. This collection satisfies the condition that there are no infinite decreasing sequences of sets $\mathfrak{T}_i$. It also, however, satisfies a stronger condition, which will turn out to be necessary and sufficient --

namely, that given any $\mathcal{T}_i$ there exists for it an absolute upper bound on the lengths of all ordered chains starting with $\mathcal{T}_i$. We therefore make the following definition.

DEFINITION. Let G be an arbitrary directed graph. We say that G has locally bounded chains if for each $x \in G$ there exists a natural number b such that no directed path beginning at x has length greater than b.

A graph with locally bounded chains must be acyclic, because a graph with directed cycles has infinite directed paths. It may therefore be considered as a partially ordered set, and for each $x \in G$, the corresponding b is also the bound for the lengths of ordered chains starting with x. In particular, any finite acyclic graph (or partially ordered set) satisfies this condition; so do the natural numbers in descending order; so do the strings in any programming language in descending order of their lengths; so does any collection of finite sets in descending order of inclusion. Any finite or infinite disjoint union of graphs with locally bounded chains also has locally bounded chains.

We first examine the case in which we may actually require that $\mathcal{T}_j > \mathcal{T}_i$, which is essentially the case examined by Floyd [3] in his discussion of termination of algorithms.

DEFINITION. A forward progress structure for a program P on $\mathcal{T}' \subset \mathcal{S} \times P$ is an ordered class of state conditions $\mathcal{T}_i \subset \mathcal{T}'$ having locally bounded chains, and such that $(S, F'_i) \in \mathcal{T}_i$ implies $P(S, F'_i) = (P'_i(S), N'_i(S)) \in \mathcal{T}_j > \mathcal{T}_i$ whenever $N'_i(S)$ is defined.

The Forward Progress Lemma may now be stated.

LEMMA. Let P be a program on $\mathcal{T}' \subset \mathcal{S} \times P$, let P' be the computation of P with respect to $F_1 = (P_1, N_1) \in P$, and let $\mathcal{S}' \subset \mathcal{T}' \cap \{\mathcal{U}=F_1\}$. Then $P'(\mathcal{S}')$ is defined (i. e., $P'(S')$ is defined for each $S' \in \mathcal{S}'$) if and only if (S', P_1), for each $S' \in \mathcal{S}'$, is contained in one of the state conditions of a forward progress structure for P.

PROOF. First of all, suppose that $P'(\mathfrak{S}')$ is defined. Let $S \in \mathfrak{S}'$; then there exist states $T_0, \ldots, T_k \in \mathfrak{S} \times P$, such that $T_0 = (S, F_1)$, $T_i = P(T_{i-1})$ for $1 \leq i \leq k$. Since $\mathfrak{S}' \subset \mathfrak{J}' \cap \{\lambda = F_1\}$, we have $T_0 \in \mathfrak{J}'$, and since P is a program on $\mathfrak{J}'$, we have therefore $T_i \in \mathfrak{J}'$, $0 \leq i \leq k$. We construct a graph from the sets of the form $\{T_i\}$ for all T_i appearing in all such sequences, setting $\{T_i\} \rightarrow \{T_j\}$ if and only if $T_j = P(T_i)$. This graph has locally bounded chains, because in fact the only chains starting at $\{T_i\}$ lead forward through the finite computation sequence which contains T_i, and the length of this sequence may therefore serve as the local bound. For each $T_i = (S_i', F_i')$, where $F_i' = (P_i', N_i')$, if $N_i'(S_i')$ is defined, then $T_{i+1} = P(T_i)$ is defined, and $P(\{T_i\}) > \{T_i\}$. Thus the given graph is a forward progress structure.

Conversely, suppose that the indicated forward progress structure exists. Let $S \in \mathfrak{S}'$, $T_0 = (S, P_1)$, $T_i = P(T_{i-1})$ for $i > 0$; we must show that the sequence T_i terminates. Since $\mathfrak{S}' \subset \mathfrak{J}' \cap \{\lambda = F_1\}$, we have $T_0 \in \mathfrak{J}'$; since P is a program on $\mathfrak{J}'$, we therefore have $T_i \in \mathfrak{J}'$ for the entire computation sequence T_i. By hypothesis, T_0 is contained in some $\mathfrak{J}_j$ in the forward progress structure, and there is an upper bound, b, on the lengths of paths starting at $\mathfrak{J}_j$. We proceed by induction on b, starting with an arbitrary $T_j = (S_j', F_j')$ in this computation sequence, where $F_j' = (P_j', N_j')$. If b = 0, then $\mathfrak{J}_j$ is terminal, which implies that $N_j'(S_j')$ is not defined, so that the computation sequence ends with T_j. Otherwise, consider $P(T_j) = T_{j+1}$; we have $T_{j+1} \in \mathfrak{J}_k > \mathfrak{J}_j$, and hence there is a directed path from $\mathfrak{J}_j$ to $\mathfrak{J}_k$. This path may be combined with an arbitrary path starting at $\mathfrak{J}_k$ to form a path whose length is not greater than b, and hence the bound on the lengths of paths starting at $\mathfrak{J}_k$ is strictly less than b. By the inductive hypothesis, the sequence starting with T_{j+1} is finite, and thus the sequence starting with T_j is finite. In particular, the sequence starting with T_0 is finite; this completes the proof.

The termination condition given above, involving locally bounded chains, is equivalent to two seemingly more general termination conditions proposed by Floyd in [3]. The first of these is the finite chain condition. An example of a graph which has this condition, but not the locally bounded chain condition, is the set of all natural numbers in descending order, together with ∞, where we write $\infty \rightarrow n$ for every natural number n. One may even choose any countable ordinal number whatsoever and view the set of all ordinals less than it, writing $a \rightarrow b$ whenever $a > b$, as a set with this condition. We have shown that for each terminating algorithm there exists a forward progress structure defined as having locally bounded chains, and all of our actual examples will be of this form. However, if a forward progress structure were defined more generally, to have merely the finite chain condition, existence of such a structure would still guarantee termination of the algorithm as above. To see this, we use an indirect proof. Let $\mathfrak{U}$ be that subclass of the forward progress structure consisting of state conditions $\mathfrak{T}_i$ such that each of them contains an element T_0 from which the sequence $T_i = P(T_{i-1})$, $i > 0$, is infinite. Then $\mathfrak{U}$ must contain a maximal element, since, as we saw before, we could otherwise construct an infinite chain inductively. Starting from this maximal element $\mathfrak{T}_i$ and its associated T_0, we could immediately derive a contradiction by considering $P(T_0)$.

The other condition introduced by Floyd is that of well-ordering. This is equivalent to the finite chain condition provided that the given set -- in this case, the forward progress structure -- is already simply ordered. It therefore follows from our previous argument that the existence of a well-ordered forward progress structure is sufficient for termination. Conversely, we may always construct, for any terminating algorithm, a well-ordered forward progress structure with locally bounded chains. This can be done by slightly modifying the proof of the Forward Progress Lemma; however, the easiest proof is probably the following. Let $\{\mathfrak{J}_i\}$ be any forward progress structure; we construct a new structure $\{\mathfrak{U}_i\}$, in which the $\mathfrak{U}_i$ are indexed by the natural numbers, by writing $T \in \mathfrak{U}_i$ whenever $T \in \mathfrak{J}_j$ for which the local bound $\underline{b}$ associated with $\mathfrak{J}_j$ is i, and $\mathfrak{U}_i \rightarrow \mathfrak{U}_j$ if $i > j$. It is now easy to see that the $\mathfrak{U}_i$ form a forward progress structure for the same algorithm, and, of course, the $\mathfrak{U}_i$ are well-ordered.

Imposed Order and Normal Progress

The forward progress condition $\mathfrak{J}_j > \mathfrak{J}_i$ is not really very useful. In many situations, all that we can get is $\mathfrak{J}_j \geq \mathfrak{J}_i$. For example, let J be the controlled variable in a loop, let $\mathfrak{J}_i$ be the state condition $\{J=i\}$ over some range of the integers i, and let $P(S_m, F_m) \in \mathfrak{J}_j$ for $(S_m, F_m) \in \mathfrak{J}_i$. If the only statements which affect J are of the form $J \leftarrow J+\underline{c}$, where $\underline{c}$ is constant, then certainly $\mathfrak{J}_j \geq \mathfrak{J}_i$, but it is quite possible that $\mathfrak{J}_j = \mathfrak{J}_i$. This will be true if $F_m = (P_m, N_n)$ where P_m is not of the form $J \leftarrow J+\underline{c}$. Of course, in such a case the program may not terminate at all; specifically, it may get stuck in an endless loop inside the original loop, provided that the inner loop does not contain any statements which increment J. The termination of the program in this case depends, then, on whether there are any directed cycles left in the program if the links corresponding to such statements are removed from the graph. If this condition holds, where $\mathfrak{J}_j \geq \mathfrak{J}_i$ as above, then the program should terminate. This argument motivates the following definitions of imposed order and normal progress structure.

DEFINITION. Let P be a program on $\mathcal{S}$, and let $\mathfrak{J}' \subset \mathcal{S} \times P$ (P is not necessarily a program on $\mathfrak{J}'$). Consider the graph defined as follows: its nodes are the elements of P, and $F_i \rightarrow F_j$ if and only if there exists $T \in \mathfrak{J}'$, $T = (S, F_i)$, with $P(T) = (S', F_j) \in \mathfrak{J}'$. Suppose that this graph has no directed cycles, i. e., is an ordered graph. Then we say that $\mathfrak{J}'$ imposes an order on P.

EXAMPLE 1. Consider any element $\mathfrak{J}_i$ of a forward progress structure. Then the graph of the above example has no links whatsoever, and thus has no directed cycles. Therefore $\mathfrak{J}_i$ imposes an order on P.

EXAMPLE 2. Consider the following program for summing the real numbers A(1) through A(N): SUM←0; A: SUM←SUM+A(I); I←I+1;

<u>if</u> I$\leq$<u>n</u> <u>then</u> <u>go to</u> A; STOP. (Note that we have purposefully forgotten to initialize I; we will return to this point later.) This program has the following graph:

$$\begin{tikzcd}\bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] \arrow[ll, bend left] & \bullet\end{tikzcd}$$

This is not an ordered graph, since it has a directed cycle. However, for any integer k, the state condition $\{J=k\}$ imposes an order on the given program. In fact, the graph of the above definition is

$$\begin{tikzcd}\bullet \arrow[r] & \bullet \arrow[r] & \bullet & \bullet \arrow[r] \arrow[ll, bend left] & \bullet\end{tikzcd}$$

Note that this graph is obtained from the graph of the program by removing certain links; we may say, informally, that the condition $\{J=k\}$ "cuts the graph" so as to leave an ordered graph remaining.

<u>DEFINITION.</u> A <u>normal</u> <u>progress</u> <u>structure</u> for a program P on $\mathfrak{J}' \subset \mathcal{S} \times P$ is an ordered class of state conditions $\mathfrak{J}_i \subset \mathfrak{J}'$ having locally bounded chains, such that each $\mathfrak{J}_i$ imposes an order on P, and such that $(S, F_i') \in \mathfrak{J}_i$ implies $P(S, F_i') = (P_i'(S), N_i'(S)) \in \mathfrak{J}_j \geq \mathfrak{J}_i$ whenever $N_i'(S)$ is defined.

<u>EXAMPLE</u> <u>1</u>. A forward progress structure is always a normal progress structure. The condition $\mathfrak{J}_j > \mathfrak{J}_i$ implies $\mathfrak{J}_j \geq \mathfrak{J}_i$; and each $\mathfrak{J}_i$ imposes an order on P, as mentioned in the example above.

<u>EXAMPLE</u> <u>2</u>. Consider the program given above for summing the real numbers A(1) through A(<u>n</u>). Let us consider the state conditions $C_i = \{I=i\}$ for all i. If we order these in the usual way, i.e., $C_i > C_j$ whenever $i > j$, then the condition $\mathfrak{J}_j \geq \mathfrak{J}_i$ as above is satisfied. Specifically, for any $(S, F_k') \in \mathfrak{J}_i$, if F_k' is the statement $I \leftarrow I+1$, then $P(S, F_k') \in \mathfrak{J}_j > \mathfrak{J}_i$; and if F_k' is not this statement, then $P(S, F_k') \in \mathfrak{J}_i$. As mentioned above, each C_i imposes an order on P. The collection of all the C_i as above does not have locally bounded chains, because i can become indefinitely large. We may,

however, construct a normal progress structure in the following way. Let C' be the state condition on the program counter λ that restricts it to indicate the statement I←I+1 or any preceding statement. Let the complement of C' in $\lambda \times P$ be C"; Let $C'_i = C_i \cap C'$ and let $C''_i = C_i \cap C''$. We order the C'_i and the C''_i by specifying, first, that $C'_i < C''_{i+1}$ for each i, so that if $(S, F_i) \in C'_i$ where F_i is the statement I←I+1 then $P(S, F_i) \in C''_{i+1} > C'_i$; and, second, that $C''_i < C'_i$ for each $i \leq n$, so that if $(S, F_i) \in C''_i$ where F_i is the *if* statement then $P(S, F_i) \in C'_i > C''_i$. In all other cases $(S, F_i) \in K$ implies $P(S, F_i) \in K$, where K is any C'_i or any C''_i. This structure has locally bounded chains, because for any $i \leq n$ the chains go from $C'_i \rightarrow C''_{i+1} \rightarrow C'_{i+1} \rightarrow \ldots \rightarrow C'_n \rightarrow C''_{n+1}$, and there they stop; whereas if $i \geq n$ the only chain is $C'_i \rightarrow C''_{i+1}$ of length 2. Each C'_i and each C''_i imposes an order on P, and the C'_i and the C''_i therefore constitute a normal progress structure.

It may seem that we have done more work than we somehow should have; and this, in fact, is true. The given loop has the controlled variable I, and we will shortly be proving some theorems which guarantee the existence of a normal progress structure whenever we have a controlled expression (which generalizes the idea of controlled variable) satisfying certain very general conditions. Normal progress structures are important because, as we shall show, the normal progress condition is necessary and sufficient; when there are no clearly visible controlled expressions, it may be necessary to fall back on the general normal progress concept, which is, in any event, easier to work with than forward progress. We recall that in setting up the above loop, we did not initialize I. What we have effectively proved, then, is that, even if we don't initialize I, the given program will always terminate, even though the answers may be wrong. This condition may be important, as when a programmer wishes, without having set up any preconditions for his program, to make a debugging run with the secure knowledge that it will not loop endlessly.

<u>EXAMPLE 3</u>. Let us change the preceding example by adding a new statement at the beginning which initializes I to 1. In this case, the C_i' and the C_i'' of the preceding example do <u>not</u> constitute a normal progress structure. Specifically, let $(S, F_1) \in C_i$, where F_1 is the new first statement. Then $P(S, F_1) \in C_1$, because I has been set to 1; and we may not have $C_1 \geq C_i$, specifically when $i > 1$. To remedy this, let D be the condition that the program counter indicates the new first statement, and let $D_i' = C_i' - D$, $D_i'' = C_i'' - D$. We write $D < D_1'$; then the D_i' and the D_i'', ordered in the same way that the C_i' and the C_i'' were, together with $D < D_1'$, forms a normal progress structure.

We now state the Normal Progress Lemma.

<u>LEMMA</u>. Any program having a forward progress structure has a normal progress structure and conversely. Any element of any set belonging to either structure may be taken as an element of some set belonging to the other.

<u>PROOF</u>. As we saw in Example 1 above, any forward progress structure is a normal progress structure. Conversely, let $\{\mathfrak{J}_i\}$ be a normal progress structure; for each $\mathfrak{J}_i$ and each $F_j \in P$ we define $\mathfrak{J}_{ij} = \mathfrak{J}_i \cap \{\lambda = F_j\}$. We order the $\mathfrak{J}_{ij}$ by specifying that $\mathfrak{J}_{ab} > \mathfrak{J}_{cd}$ if $\mathfrak{J}_a > \mathfrak{J}_c$ in the given normal progress structure, or if $\mathfrak{J}_a = \mathfrak{J}_c$ and $F_b > F_d$ in the order which is imposed by $\mathfrak{J}_a$ on the graph of the program. Then the $\mathfrak{J}_{ij}$ involve exactly the same elements $T \in \mathcal{S} \times P$ as do the $\mathfrak{J}_i$. The structure $\mathfrak{J}_{ij}$ has locally bounded chains; in fact, if <u>b</u> is the bound associated with a particular $\mathfrak{J}_i$ in the normal progress structure and <u>n</u> is the number of statements in the program, then <u>bn</u> is the bound associated with any of the $\mathfrak{J}_{ij}$ for this value of i. If $(S, F_k') \in \mathfrak{J}_{ij}$, where $F_k' = (P_k', N_k')$, and

$N_{\frac{1}{2}}'(S)$ is defined, then $P(S, F_{\frac{1}{2}}') = (S', F_n'') \in \mathcal{J}_{mn}$, where $\mathcal{J}_{ij} \subset \mathcal{J}_i$ and $\mathcal{J}_{mn} \subset \mathcal{J}_m$; here either $\mathcal{J}_m > \mathcal{J}_i$, in which case $\mathcal{J}_{mn} > \mathcal{J}_{ij}$, or else $\mathcal{J}_m = \mathcal{J}_i$, in which case there is a directed path (of length 1) from F_j to F_n in the graph of the program which indicates $F_n > F_j$ in the order imposed by $\mathcal{J}_i$ on P, and thus again $\mathcal{J}_{mn} > \mathcal{J}_{ij}$. This completes the proof.

<u>COROLLARY</u>. Let P be a program on $\mathcal{J}' \subset \mathcal{S} \times P$, let P' be the computation of P with respect to $F_1 = (P_1, N_1) \in P$, and let $\mathcal{S}' \subset \mathcal{J}' \cap \{\lambda = F_1\}$. Then $P'(\mathcal{S}')$ is defined if and only if (S', F_1), for each $S' \in \mathcal{S}'$, is contained in one of the state conditions of a normal progress structure for P.

We shall turn part of this corollary into a definition. A statement F_i of a program P on $\mathcal{J}'$ will be called an <u>entry point</u> for a normal progress structure for P if (S', F_i), for every $S' \in \mathcal{S}_i$ (where $\mathcal{S}_i$ is the precondition associated with F_i by $\mathcal{J}'$), is contained in one of the state conditions of that normal progress structure. A normal progress structure for which <u>every</u> statement of the program is an entry point will be called <u>universal</u>. A program on $\mathcal{J}'$ has a universal normal progress structure if and only if it always terminates when started in $\mathcal{J}'$; a program on $\mathcal{J} = \mathcal{S} \times P$ has a universal normal progress structure if and only if its computation is a total function. As we have seen above, any program whose graph is ordered satisfies this condition.

Any finite or infinite disjoint union of normal progress structures is itself a normal progress structure. It is therefore possible to set up different normal progress structures for different subsets of $\mathcal{S}'$. One can also parametrize a normal progress structure, so that its size or its structure depend on the values of variables which make up the condition $\mathcal{S}'$.

A <u>universal</u> <u>normal</u> <u>progress</u> <u>structure</u> for a program P on $\mathcal{S}$ is a normal progress structure which includes (S, P_1) for <u>every</u> S $\in \mathcal{S}$. A program has a universal normal progress structure if and only if its computation is a total function -- i. e., it <u>always</u> terminates. As we have seen above, any program whose graph is ordered has a universal normal progress structure, which is, in fact, a finite forward progress structure.

A <u>forward</u> (or <u>normal</u>) <u>progress</u> <u>sequence</u> is a forward (or normal) progress structure which is linearly ordered, i. e., for every $\mathfrak{T}_i$ and $\mathfrak{T}_j$ either $\mathfrak{T}_i > \mathfrak{T}_j$, $\mathfrak{T}_i = \mathfrak{T}_j$, or $\mathfrak{T}_i < \mathfrak{T}_j$. Any finite or countable forward (or normal) progress structure may be re-ordered so that it becomes a forward (or normal) progress sequence; in the finite case this follows from Szpilrajn's theorem on partially ordered sets, while in the countable case it follows from a relatively easy generalization: any graph with locally bounded chains may be mapped in a one-to-one, order-preserving manner onto the natural numbers in descending order. This map may be constructed inductively on the local bound for chains.

The Verification Theorem

THEOREM. Let P be a program on $\mathcal{J}' \subset \mathcal{S} \times P$, let P' be the computation of P with respect to $F_1 = (P_1, N_1) \in P$, and let $\mathcal{S}' \subset \mathcal{S}$, $\mathcal{S}'' \subset \mathcal{S}$. Then $P(\mathcal{S}') \subset \mathcal{S}''$ (i. e., for each $S \in \mathcal{S}'$, P(S) is defined and is in $\mathcal{S}'$) if and only if there exists a normal progress structure $\{\mathcal{J}_i\}$ for P with each $\mathcal{J}_i \subset \mathcal{J}'$, such that $(\mathcal{S}', P_1)$ is contained in the union of the $\mathcal{J}_i$, and in addition we have $\mathcal{S}' \subset \mathcal{S}_1$, where $\mathcal{S}_1$ is the precondition induced by the consistent universal condition $\mathcal{J}'$ at F_1, and $S'' \in \mathcal{S}''$ for each S" such that $(S'', F_j) \in \mathcal{J}'$ for some F_j for which the corresponding $N_j(S'')$ is not defined.

PROOF. This now follows immediately from Floyd's Theorem and the Normal Progress Lemma given above. We note that in both of the subdivisions of the correctness problem for algorithms, the given techniques -- consistency in the one case, and normal progress in the other -- are not only sufficient, but necessary as well.

In applying the Verification Theorem, we need one further construction which we have not discussed as yet: the idea of factoring a program into sections.

Factor Graphs, Factor Programs, and Sections

Let G be any graph and let $\mathcal{D}$ be a decomposition of G. There is then an induced graph structure on $\mathcal{D}$, in which, if $D_1 \in \mathcal{D}$, $D_2 \in \mathcal{D}$, then $D_1 \rightarrow D_2$ if and only if there exist $G_1 \in D_1$ and $G_2 \in D_2$ with $G_1 \rightarrow G_2$. We call this graph the factor graph of G with respect to $\mathcal{D}$.

Let P be a program on $\mathcal{T} \subset \mathcal{S} \times P$, and let $\mathcal{D}$ be a decomposition of P. Let $D \in \mathcal{D}$; then we may make D into a program on $\mathcal{T}_D \subset \mathcal{T}$ as follows. The elements $F_i \in D$ are associated with pairs (P_i, Q_i), where P_i is as before while $Q_i(S) = N_i(S)$ whenever $N_i(S) \in D$; we take $Q_i(S)$ to be undefined whenever $N_i(S) \notin D$. Such a program is called a section of P. If $F_i \in D$, then F_i is an entry (or an entry point) of D if and only if there exists $F_j \in P$, $F_j \notin D$, with $F_j \rightarrow F_i$ in P. Now suppose that $\mathcal{D}$ is such that each section $D \in \mathcal{D}$ has at most one entry. Let $D_1 \in \mathcal{D}$ have no entries and let $F_1 \in D_1$. For each $D \in \mathcal{D}$ let P_D be the computation of the section D with respect to its entry, if it has one; otherwise P_D is the partial function defined nowhere, unless $D = D_1$, in which case we take P_D to be the computation of D_1 with respect to F_1. Let $\mathcal{S}_i$ be the precondition associated with F_i by $\mathcal{T}$. If $S \in \mathcal{S}_i$ and F_D is the entry point of D, we define $N_D : \mathcal{S} \rightarrow \mathcal{D}$ by $N_D(S) = D'$, where the computation sequence $T_0, \ldots, T_m$ of P_D starting with $T_0 = (S, F_D)$ ends with (S', F') for some $F' \in D'$ (clearly, of course, then $F' = F_{D'}$). Then the factor program of P on $\mathcal{D}$ with entry F_1 is defined to be the program whose statements are the pairs (P_D, N_D) for all $D \in \mathcal{D}$. The graph of this factor program is the factor graph of P with respect to $\mathcal{D}$.

Sections of a program are somewhat analogous to cosets of a group with respect to a subgroup, except that they do not all have to be the same size. However, just as a factor group is a set of cosets under an induced group structure, so a factor program is a set of sections under an induced program structure.

THEOREM. Let Q be the factor program of P on $\mathfrak{D}$ with entry point F_1. Then the computation of Q with respect to that element $D \in \mathfrak{D}$ which contains F_1 is the same as the computation of P with respect to F_1.

PROOF. Let the two computations mentioned in the theorem be denoted by Q' and P' respectively, and let $T_0, \ldots, T_n$ be any computation sequence for Q, with $T_i = (S_i, D'_i)$, in which D'_1 contains F_1; by definition, F_1 is the entry for D'_1. We have $Q(T_i) = (S_{i+1}, D'_{i+1})$, $0 \leq i \leq n$, and $Q'(S_0) = P'_n(S_n)$, where $D'_i = (P'_i, N'_i)$; we must show $P'(S_0) = P'_n(S_n)$. Here P'_i is the computation of the section D'_i with respect to its entry F'_i, and this means that there is a computation sequence T_{ij} for this value of i, $0 \leq j \leq m_i$, such that $T_{i0} = (S_i, F'_i)$. We have $T_{im_i} = (S'', F'')$ for some $F'' \in D'_i$, and $N''(S'')$ is not defined in the section D'_i, where $F'' = (P'', N'')$; this means that either $N''(S'')$ is not defined in the program P, or else $N''(S'') = F'$ in P for some $F' \notin D'_i$ and hence $F' \in N'_i(S_i) = D'_{i+1}$. In this latter case, we have $P''(S'') = P'_i(S_i) = S_{i+1}$, and also $F'' \rightarrow F'$ in P, which implies that F' is the entry of D'_{i+1}; hence $T_{i,n_i+1} = (P''(S''), N''(S'')) = (S_{i+1}, D'_{i+1}) = T_{i+1,0}$. Hence the sequence $T_{00}, \ldots, T_{0m_0}, T_{10}, \ldots, \ldots, T_{n0}, \ldots, T_{nm_n}$ is the computation sequence of P starting with $T_{00} = (S_0, F'_0) = (S_0, P_1)$. If $N''(S'')$ as above is not defined in P, then $i = n$ and $P''(S'') = P'_n(S_n)$; since T_{nm_n} is the end of the given computation sequence of P, we also have $P''(S'') = P'(S_0)$, and thus $P'(S_0) = P'_n(S_n)$.

Conversely, let $T'_0, \ldots, T'_z$ be any computation sequence of P starting with $T'_0 = (S'_0, F_1)$; we renumber the T'_i as follows. Set $T_{00} = T'_0$. If $T'_i = (S'_i, F'_i)$ with $F'_i \in D_x$ has been renumbered as T_{jk}, we renumber $T'_{i+1} = (S'_{i+1}, F'_{i+1})$, with $F'_{i+1} \in D_y$, as $T_{j,k+1}$ if $D_x = D_y$, or as $T_{j+1,0}$, setting $m_j = k$, if $D_x \neq D_y$. If T'_z has been renumbered as T_{jk}, we set $n = j$ and $m_n = k$. Thus the sequence T'_i has been renumbered as $T_{00}, \ldots, T_{0m_0}, T_{10}, \ldots, \ldots, T_{n0}, \ldots, T_{nm_n}$, and, setting each $T_{ij} = (S_{ij}, F_{ij})$, all F_{ij} for each fixed i, $0 \leq j \leq m_i$, belong to the same member D_i' of the decomposition. Setting $F_{im_i} = (P'', N'')$, we have $N''(S_{im_i}) \notin D_i'$ in P, for $i \neq n$, and hence $N''(S_{im_i})$ is undefined in D_i'; if $i = n$, then $N''(S_{nm_n})$ is undefined in P and hence also in D_n'. Thus in either case $T_{i0}, \ldots, T_{im_i}$ is the computation sequence of S_{i0} in D_i', and $P_i'(S_{i0}) = P''(S_{im_i}) = S_{i+1,0}$ for $i \neq n$, where P_i' is the computation of D_i', while if $i = n$, $P_n'(S_{n0}) = P''(S_{nm_n}) = P'(S_0)$. If $T_i'' = (S_i'', F_i'')$ is the computation sequence of S_0 in Q, we now show inductively that $T_i'' = T_{i0}$; this is clear for $i = 0$. If $T_j'' = T_{j0}$, then, writing $F_j'' = (P_j'', N_j'')$ and using the definition of a factor program, we have $P_j''(S_{j0}) = S_{j+1,0}$, as above, while $N_j''(S_{j0}) = D_{j+1}'$, completing the induction. In particular, $T_n'' = T_{n0}$ and $Q'(S_0) = P_n'(S_{n0}) = P'(S_0)$. This completes the proof.

<u>THEOREM</u>. Let P be a program on $\mathfrak{J}' \subset \mathcal{S} \times P$, and let Q be the factor program of P on $\mathfrak{D}$ with the entry $F_1 \in D_1$, $D_1 \in \mathfrak{D}$. Suppose that D_1 is an entry for a normal progress structure on Q, and likewise, for each section $D_i \in \mathfrak{D}$, its entry F_i is an entry for a normal progress structure on D_i. Then F_1 is an entry for a normal progress structure on P.

<u>PROOF</u>. Let $T_0 = (S, F_1) \in \mathfrak{J}'$. We seek to prove that the computation sequence beginning with T_0 must terminate; under these conditions, F_1 will be an entry for a normal progress structure on P. Since F_1 is the entry of D_1, there is a computation sequence $T_0 = T_{00}, T_{01}, \ldots, T_{0m_0}$ in D_1; if T_{0m_0} is not the end of the original computation sequence, the next element of this sequence is $T_{10} = (S_{10}, F_{10})$ where F_{10} is the entry of a new section of the program. Since this is an entry for a normal progress structure for that section, there is a computation sequence $T_{10}, T_{11}, \ldots, T_{1m_1}$ in that section. The original computation sequence may thus be

written $T_0 = T_{00}, \ldots, T_{0m_0}, T_{10}, \ldots, \ldots$, and this corresponds, as in the proof of the preceding theorem, to a computation sequence $T_{00}, T_{10}, \ldots$, in the factor program Q, where $T_{00} = (S, D_1)$. Since D_1 is an entry for a normal progress structure on Q, this sequence must terminate, and this means, again as in the preceding proof, that the original sequence must terminate. This completes the proof.

This theorem may be used in analyzing programs having sections or factor programs which have ordered graphs. As we have seen, any program with an ordered graph automatically has a universal normal progress structure. In a typical case, a program might have three sections, such that the first and third are ordered but the second is not, while the factor program is likewise ordered. Thus normal progress of the entire program is equivalent to normal progress of the second section alone.

Controlled Expressions

Controlled expressions in programs are the natural generalizations of controlled variables in FORTRAN and ALGOL -- i. e., variables which appear in DO statements and *for* statements. The generalization is necessary because in some programs control is exercised not by a single variable but by several, which combine in some way to form a single controlled expression.

DEFINITION. Let Y be any set with addition (and therefore also with multiplication by positive integers), ordered in such a way that if a, b, c $\in$ Y with c $>$ 0 there exists a positive integer k such that a+kc $>$ b. Let P be a program on $\mathcal{T}' \subset \mathcal{S} \times P$, and let e: $\mathcal{S} \rightarrow Y$. Suppose that for any statement $F_i = (P_i, N_i)$ of P, if $(S, F_i) \in \mathcal{T}' - \{\underline{\text{exit}}\}$, then $e(P_i(S)) \geq e(S)$. Suppose also that there exists c $\in$ Y, c $>$ 0, such that an acyclic graph is obtained by removing from the graph of the program all links $F_i \rightarrow F_j$ such that $(S, F_i) \in \mathcal{T}'$ and $N_i(S) = F_j$ implies $e(P_i(S)) \geq e(S) + c$. Then e is a *controlled expression* for P. If x $\in$ M and the expression e defined by e(S) = S(x) for all S $\in \mathcal{S}$ is a controlled expression for P, then x is a *controlled variable* for P.

The axiom given for the set Y is such that the integers, the reals, or the rationals will satisfy it. Another possible set Y is the set of all floating point numbers on a given computer in which the fraction part of the number is strictly bounded (i. e., fits into a given maximum number of bits) but the exponent part is unlimited in size, under the usual rules of floating point addition. As we have mentioned before, size limitations on numbers in actual computers may be handled by specifying, as a precondition for every statement in a program, that every variable involved in that statement lies within the limits specified by the given computer.

The two axioms for controlled expressions are generalizations of the usual properties of controlled variables. The first axiom says that a controlled expression must be monotonically increasing. Note that it is not necessary for us to develop a dual theory of monotonically decreasing expressions, because if e is a decreasing expression then $-e$ (i. e., the expression $f: \mathcal{S} \rightarrow Y$ defined by $f(S) = -e(S)$ for all $S \in \mathcal{S}$) is an increasing expression. Although the monotonic condition is the usual one, it is by no means necessary; we may have, for example, a controlled expression which proceeds erratically, or "two steps forward, one step back" toward its goal; such an expression is not covered by this definition. The second axiom says roughly that the values of e do not converge; clearly, if the successive values of e were $y_1, y_2, \ldots,$ where the y_i constitute a monotonic sequence converging to $y \in Y$, and the only test condition in the program on e asked whether its value was greater than y, for example, then e would not satisfy this axiom, and, in fact, our program could enter an endless loop. Note that if Y is the integers then we may always assume $c = 1$, replacing the condition $e(P_i(S)) \geq e(S) + c$ by $e(P_i(S)) > e(S)$.

The following three theorems state that a program with a controlled expression always terminates provided that this expression satisfies some additional condition. It should be noted that a controlled expression for a program P is one which satisfies the conditions given in the definition over the entire range of statements of P. Many programs will have DO loops, for loops, or the equivalent, and each of these will have a controlled expression, but these will not be controlled expressions for the entire program; each one will only be a controlled expression for its own loop. It is necessary in such a case to factor the given program into sections, in which the loops are (some of) the sections; each loop can now be proved terminating, and thus the entire program terminates.

THEOREM. Let P be a program on $\mathfrak{J}' \subseteq \mathfrak{S} \times P$, let $e: \mathfrak{S} \rightarrow Y$ be a controlled expression for P, and for each $F_i \in P$ let $\mathfrak{S}_i$ be the precondition associated with F_i by $\mathfrak{J}'$. Let $e': \mathfrak{S} \rightarrow Y$ be such that each $\mathfrak{S}_i$ is contained in $\{e \leq e'\}$ and such that $\Delta(e', \mathfrak{S}_i) \cap \rho(P_i, \mathfrak{S}_i) = \phi$ for each $\mathfrak{S}_i$. Then P terminates when started in $\mathfrak{J}'$.

This condition generalizes the most natural condition for a controlled variable: that it be bounded above, preferably by a constant. For any actual computer, where Y is the set of possible contents of a computer word, such a constant always exists, namely the largest element of Y. If the upper bound is not constant, we hypothesize here that it be "effectively constant" in the sense that it can never change during the course of a program. It is clearly possible for a loop from I = 1 through N to be an endless loop if inside it we keep jacking up the value of N.

PROOF. Let $i \geq 0$ be an integer, let $b \in Y$, and consider the set $\mathfrak{J}_{ib} = \{(S, F_k) \in \mathfrak{J}': e'(S) = b \text{ and } b-ic \leq e(S) < b-(i-1)c\}$, where c is the constant appearing in the definition of the controlled expression e. We order the $\mathfrak{J}_{ib}$ by writing $\mathfrak{J}_{ib} > \mathfrak{J}_{jd}$ if $i < j$ (note the reversal of order) and $b = d$. Under this ordering we shall now show that the $\mathfrak{J}_{ib}$ constitute a universal normal progress structure for P. The structure is certainly universal, because if $e(S) = a$ and $e'(S) = b$, for any $(S, F_j) \in \mathfrak{J}'$, then by hypothesis $a \leq b$, but there exists a positive integer k with $a+kc \geq b$; we then have $(S, F_j) \in \mathfrak{J}_{ib}$ where i is equal to the least integer k having this property. The structure also clearly has locally bounded chains, because from any $\mathfrak{J}_{ib}$ the only chain goes from $\mathfrak{J}_{ib} \rightarrow \mathfrak{J}_{i-1,b} \rightarrow \ldots \rightarrow \mathfrak{J}_{2b} \rightarrow \mathfrak{J}_{1b}$, and hence i is the associated local bound. Let $(S, F_j) \in \mathfrak{J}_{ib}$, and let

$P(S, F_j) = (S', F_j')$; it is sufficient to show $(S', F_j') \in \mathfrak{J}_{mb}$ for $m \leq i$. Setting $F_j = (P_j, N_j)$, we have $S' = P_j(S)$, and if $\mathcal{S}_j$ is the precondition associated with F_j by $\mathfrak{J}'$, we have $S \in \mathcal{S}_j$. Thus $\Delta(e', \mathcal{S}_j) \cap \rho(P_j, \mathcal{S}_j) = \phi$ implies $e'(S) = e'(P_j(S)) = e'(S') = b$; also, by the definition of a controlled expression, $e(P_j(S)) \geq e(S)$, and thus, by the definition of the $\mathfrak{J}_{mb}$, we have $m \leq i$. This completes the proof.

<u>THEOREM</u>. Let P be a program on $\mathfrak{J}' \subset \mathcal{S} \times P$, let $e: \mathcal{S} \to Y$ be a controlled expression for P, and for each $F_i \in P$ let $\mathcal{S}_i$ be the precondition associated with F_i by $\mathfrak{J}'$. Let $e': \mathcal{S} \to Y$ be such that $\Delta(e', \mathcal{S}_i) \cap \rho(P_i, \mathcal{S}_i) = \phi$ for each $\mathcal{S}_i$, and suppose that an acyclic graph is obtained by removing from the graph of P all links involving nodes $F_i = (P_i, N_i) \in P$ such that $(S, F_i) \in \mathfrak{J}'$ and $e(S) > e'(S)$ implies that $N_i(S)$ is not defined. Then P terminates when started in $\mathfrak{J}'$.

This condition is sometimes easier to spot than the previous one. The definition of a controlled expression implies that in any controlled loop it should be impossible to escape <u>incrementation</u> of the controlled expression by some minimum increment. The present theorem says that if in addition it is impossible to escape <u>comparison</u> between the controlled expression and some upper bound, and transfer out of the loop if it is greater than that upper bound, then the loop must terminate.

<u>PROOF</u>. Let $\{\mathfrak{J}_{ib}\}$ be the universal normal progress structure of the preceding theorem. Let Q be the set of all links in P which are removed from the graph of P by the hypothesized construction, and corresponding to each $F_k \notin Q$ we consider $\mathfrak{J}_k' = \{(S, F_k) \in \mathfrak{J}': e(S) \geq e'(S)\}$. We write $\mathfrak{J}_k' > \mathfrak{J}_{ib}$ for all $\mathfrak{J}_k'$ and all $\mathfrak{J}_{ib}$, while $\mathfrak{J}_k' > \mathfrak{J}_m'$ if and only if $F_k > F_m$ in the order imposed on the graph of P - Q by the fact that it is acyclic. Finally, let $\mathfrak{J}_0' = \{(S, F_k) \in \mathfrak{J}': e(S) \geq e'(S)\}$ and $F_k \in Q\}$, and write $\mathfrak{J}_0' > \mathfrak{J}_k'$ for each $\mathfrak{J}_k'$.

We shall show that the expanded ordered structure comprising the $\mathfrak{J}_{ib}$, the $\mathfrak{J}'_k$, and $\mathfrak{J}'_0$ constitutes a universal normal progress structure for P. The structure is certainly universal, since if $(S, F_k) \in \mathfrak{J}'$ and $e(S) < e'(S)$ then (S, F_k) is in some $\mathfrak{J}_{ib}$, while if $e(S) \geq e'(S)$ then (S, F_k) is in $\mathfrak{J}'_0$ if $F_k \in Q$ and is otherwise in some $\mathfrak{J}'_k$. The structure has locally bounded chains because any chain starting with $\mathfrak{J}_{ib}$ can have no more than i elements before it gets into the $\mathfrak{J}'_k$ or $\mathfrak{J}'_0$, and any chain in the $\mathfrak{J}'_k$ can have no more elements in it than there are statements in the given program. If $(S, F_j) \in \mathfrak{J}_{ib}$, then either $P(S, F_j) \in \mathfrak{J}_{mb} > \mathfrak{J}_{ib}$ as before, or else $P(S, F_j) \in \mathfrak{J}'_k > \mathfrak{J}_{ib}$ or $\mathfrak{J}'_0 > \mathfrak{J}_{ib}$. If $(S, F_j) \in \mathfrak{J}'_j$, then, setting $F_j = (P_j, N_j)$, we have $e(P_j(S)) \geq e(S) \geq e'(S)$, and so $P(S, F_j)$ cannot be in $\mathfrak{J}_{ib}$; therefore, either it is in $\mathfrak{J}'_0 \geq \mathfrak{J}'_j$, or else it is in $\mathfrak{J}'_k$, where there is a directed path (of length 1) from F_j to F_k in the graph P - Q and therefore $\mathfrak{J}'_k > \mathfrak{J}'_j$. Finally, if $(S, F_j) \in \mathfrak{J}'_0$, where $F_j = (P_j, N_j)$, then $N_j(S)$ is undefined. This completes the proof.

COROLLARY. Let P be a program on $\mathfrak{J}' \subset \mathcal{S} \times P$, let $e: \mathcal{S} \to Y$ be a controlled expression for P, and for each $F_i \in P$ let $\mathcal{S}_i$ be the precondition associated with F_i by $\mathfrak{J}'$. Let $e': \mathcal{S} \to Y$ be such that $\Delta(e', \mathcal{S}_i) \cap \rho(P_i, \mathcal{S}_i) = \phi$ for each $\mathcal{S}_i$, and suppose that an acyclic graph is obtained by removing from the graph of P all links involving nodes $F_i \in P$ such that the corresponding $\mathcal{S}_i$ is contained in $\{e < e'\}$. Then P terminates when started in $\mathfrak{J}'$.

PROOF. The condition that $\mathcal{S}_i$ is contained in $\{e < e'\}$ clearly implies, vacuously, that $N_i(S)$ is not defined for each S for which $(S, F_i) \in \mathfrak{J}'$ and $e(S) \geq e'(S)$, because there are no such S. The corollary thus reduces to the theorem.

We may use the corollary, rather than the preceding theorem, when the test or tests within a loop are of type "equal" or "unequal," rather than "greater than" or "less than," or when they involve auxiliary quantities not directly related to the controlled variable. It is more closely related in spirit to the first of the above theorems than to the second, and in fact is much easier to use than this first theorem because the condition $e < e'$ only needs to be verified on a sufficient substructure. This can be done at the same time, and using the same machinery, as when consistency is checked. As an example, consider the following algorithm for summing the real numbers $A(1), \ldots, A(\underline{n})$: $I \leftarrow 0$; $SUM \leftarrow 0$; A: $I \leftarrow I+1$; $SUM \leftarrow SUM+A(I)$; <u>if</u> $I \neq \underline{n}$ <u>then</u> <u>go to</u> A; STOP. We can then associate with the label A the precondition $I < \underline{n}$ and prove this consistent; since an acyclic graph is obtained by removing the statement $I \leftarrow I+1$, the corollary may be applied directly.

<u>THEOREM</u>. Let P be a program on $\mathfrak{J}' \subset \mathcal{S} \times P$, let $e: \mathcal{S} \rightarrow Y$ be a controlled expression for P, and for each $P_i \in P$ let $\mathcal{S}_i$ be the precondition associated with P_i by $\mathfrak{J}'$. Let $e': \mathcal{S} \rightarrow Y$ be such that each $\mathcal{S}_i$ is contained in $\{e < e'\}$ and such that $e'(P_i(S)) \leq e'(S)$ for each P_i. Then P terminates when started in $\mathfrak{J}'$.

This theorem is occasionally necessary when working with two indices which are moving toward each other -- one always increasing and the other always decreasing. Note that it is necessary for only one of the two to be a controlled expression.

<u>PROOF</u>. This follows immediately from the first theorem above by realizing that, under the given conditions, $e-e'$ is a controlled expression which is bounded by the constant zero. We could, of course, have formulated the second theorem above in this way as well.

We note, on the general subject of controlled expressions, that the condition $e(P_i(S)) \geq e(S)$ for a controlled expression e is immediate for any P_i for which $\rho(P_i, \mathcal{S}_i) \cap \Delta(e) = \phi$, where $\mathcal{S}_i$ is the precondition associated with P_i. In fact, in this case we have $e(P_i(S)) = e(S)$ for all $S \in \mathcal{S}_i$.

The Skeleton of a Program

DEFINITION. Let P be an arbitrary program, and let F_j be a statement of P such that there does not exist any statement $F_i \in P$ with $F_i \rightarrow F_j$ in the graph of P. Then F_j is an initial statement of P.

Any initial statement F_j of a program P may be removed to form a new program Q in which the functions P_i and N_i are the same as before. In particular, we cannot have $N_i(S, F_k) = F_j$ for any (S, F_k), for that would imply $F_k \rightarrow F_j$ in the graph of the program. Thus each N_i may be defined in Q exactly as it was in P.

DEFINITION. The skeleton of a program P with no initial statements is defined to be P; the skeleton of a program P containing the initial statement F_j is defined, recursively, to be the skeleton of the program Q obtained by removing F_j from P as above.

The process of finding the skeleton of a program consists, therefore, in successively removing initial statements from the program until there are none left. The removed statements, however, need not all be initial statements of the original program. In particular, the skeleton of a program whose graph is ordered is the null set, because the smallest element of such a program will always be an initial statement, and when this is removed the result, if non-null, will still be ordered and will thus have a smallest statement of its own.

THEOREM. Any program always terminates if and only if its skeleton always terminates; [illegible].

This theorem is used mainly in connection with controlled expressions. Any controlled expression must be monotonic; it is therefore unlikely that any expression will ever be a controlled expression for a program in which it is initialized, because its value before initialization is presumably unknown, and might be larger than its value after initialization. By restricting our attention to the skeleton of the program, we avoid this problem because the initializations in a program are normally not part of its skeleton.

PROOF. Let T_0, T_1, ..., be an arbitrary computation sequence of the program P whose skeleton Q always terminates; we show that the given computation sequence terminates. If any $T_i = (S_i, F'_i)$ for $F'_i \notin Q$, then the sequence beginning with T_i terminates, so we may assume this is not the case. If T_i does not terminate, the program statements appearing in this sequence repeat themselves; thus there is some smallest i, j, $i < j$, such that $T_k = (S_k, F'_k)$ for $i \leq k \leq j$ and $F'_i = F'_j$. Let F'_m be the first of the F'_k, $i \leq k \leq j$, that is removed from P in the course of forming the skeleton; if $F'_k = F'_i$, we may take $F'_k = F'_j$, and so in all cases we may assume $k > i$. But now we obtain a contradiction: F'_k should not have been removed, because F'_{k-1} is in all cases defined and $F'_{k-1} \rightarrow F'_k$ in the graph of P. This completes the proof.

<u>FORTRAN</u> <u>Iteration</u>

We shall now apply our general theorems on controlled expressions to the specific case of DO statements.

<u>DEFINITION</u>. Let P be an ordered program on $\mathcal{S} = \prod_{x \in M} V_x$, with statements $\mathbf{F}_1, \ldots, \mathbf{F}_n$. Let I denote the set of integers between $-p$ and q, inclusive, and if $i, j \in I$, we define $i \oplus j$ to be that member of I which is congruent to $i+j$ modulo $p+q$. Let $x \in M$ be such that $V_x = I$, and let $n_1: \mathcal{S} \rightarrow I$, $n_2: \mathcal{S} \rightarrow I$, $n_3: \mathcal{S} \rightarrow I$. Then the <u>FORTRAN</u> <u>iteration</u> $\{\text{DO } \mathbf{F}_n\, x = n_1, n_2, n_3\}$ (or $\{\text{DO } \mathbf{F}_n\, x = n_1, n_2\}$ whenever n_3 is the constant function $n_3(S) \equiv 1$) is the ordered program with statements $\mathbf{F}_0, \ldots, \mathbf{F}_{n+2}$, where $\mathbf{F}_1, \ldots, \mathbf{F}_n$ are the statements of P, whereas $\mathbf{F}_0$ is the assignment $\{x \leftarrow n_1\}$, $\mathbf{F}_{n+1}$ is the assignment $\{x \leftarrow x \oplus n_3\}$, and $\mathbf{F}_{n+2}$ is the conditional $\{$<u>if</u> $x \leq n_2$ <u>then</u> <u>go to</u> $P_1\}$.

More explicitly, $P_0(S) = S'$ where $S'(x) = n_1(S)$ and $S'(z) = S(z)$ for $z \neq x$, with $N_0(S) \equiv \mathbf{F}_1$; $N_n(S) = \mathbf{F}_{n+1}$ in the FORTRAN iteration whenever $N_n(S)$ was undefined in P; $P_{n+1}(S) = S'$ where $S'(x) = S(x) \oplus n_3(S)$ and $S'(z) = S(z)$ for $z \neq x$, with $N_{n+1}(S) \equiv \mathbf{F}_{n+2}$; and P_{n+2} is the identity, while $N_{n+2}(S) = \mathbf{F}_1$ whenever $S(x) \leq n_2(S)$ and $N_{n+2}(S)$ is otherwise undefined.

The following theorem on termination of FORTRAN iterations is sufficient, but is far from being necessary; it embodies, however, most of the usual restrictions on FORTRAN iterations made by actual FORTRAN systems.

<u>THEOREM</u>. The FORTRAN iteration $\{\text{DO } \mathbf{F}_n\; x = n_1, n_2, n_3\}$ will always terminate, provided that the following conditions are all satisfied:

(1) The original ordered program P always terminates.

(2) The function n_3 satisfies $n_3(S) > 0$ for all S.

(3) For any S, we have $n_1(S) + n_3(S) < q$.

(4) For any S, we have $n_2(S) + n_3(S) < q$.

(5) For any i, $1 \leq i \leq n$, if $P_i(S) = S'$, then $S'(x) = S(x)$.

(6) For any i, $1 \leq i \leq n$, we have $n_2(P_i(S)) = n_2(S)$.

Roughly, a FORTRAN iteration terminates if it contains no non-terminating inner loop; if the increment is always positive (a well-known special FORTRAN condition); if there can be no arithmetic overflow when the controlled variable is incremented; and if neither the controlled variable nor its maximum value is ever changed in any of the statements of the loop. It is not necessary for the initial value of the controlled variable to be less than or equal to its "maximum" value; and it is not even necessary for the increment to be a constant, as long as it remains strictly positive.

PROOF. Consider the decomposition of the given FORTRAN iteration into $\{F_0\}$, $\{F_1, \ldots, F_n\}$, $\{F_{n+1}\}$, and $\{F_{n+2}\}$. Since none of F_0, F_{n+1}, F_{n+2} ever branch to themselves, and since, by hypothesis, the original ordered program always terminates, it suffices to consider the factor program. The graph of this factor program is

$$\begin{tikzcd}
F_0 \arrow[r] & P \arrow[r] & F_{n+1} \arrow[r] & F_{n+2} \arrow[ll, bend left]
\end{tikzcd}$$

We may place the precondition $\{x < q-n_3\}$ at the statement P in this program. The consistency of this precondition follows from the fact that F_0 sets x to n_1, and $n_1 < q-n_3$ by condition (3) above; whereas, if we go from F_{n+2} to P, we have $x \leq n_2$, and $n_2 < q-n_3$ by condition (4) above. We have $e_x(P'(S)) = e_x(S)$ for all S by condition (5) above, where P' is the computation of the original ordered program, and thus $\{x < q-n_3\}$ is likewise a precondition for the statement F_{n+1}, which sets x to $x \bullet n_3$. Since $n_3 > 0$, we have $-p \leq x < x+n_3 < q$, and thus $x \bullet n_3 = x + n_3$.

For termination, it suffices to consider the skeleton of the given program, so we may eliminate F_0 from consideration. We shall show that a controlled expression for the remaining section is $e_x: \mathcal{S} \rightarrow Y$, where Y is the set of <u>all</u> integers and $e_x(S) = S(x)$; then we may apply the first of the above theorems, since we have shown that $x < q$ everywhere within the loop. The set Y of all integers certainly satisfies the conditions on the set Y given in the definition of a controlled expression. Of the conditions on e_x, we may verify the first for the three remaining statements; in fact, we have shown above that $e_x(P'(S)) = e_x(S)$ for all S, and $e_x(P_{n+1}(S)) > e_x(S)$ since the $\oplus$ addition is equivalent in this case to the $+$ addition; and $e_x(P_{n+2}(S)) = e_x(S)$ follows from the fact that P_{n+2} is the identity. The minimum increment c may be taken to be 1, and if we eliminate P_{n+1} from the graph of the program, it becomes acyclic; here P_{n+1} may be eliminated because the two forms of addition are the same here and because of condition (6) above. This completes the proof.

There are a number of generalizations which may be made. Condition (3) is actually superfluous, although if we eliminate it we add a rather uninteresting case: the initial value is extremely large (q, for instance), and the first time through the loop this becomes negative by overflow; but eventually the loop terminates anyway, as this negative number is repeatedly increased. Condition (6) could be replaced by $n_2(P_1(S)) \leq n_2(S)$, and the proof goes through much as before, except that the third of the above theorems is used rather than the first. Condition (5) could also be replaced by $S'(x) \geq S(x)$, but this would complicate the discussion of integer overflow, and condition (4) would have to be changed to make up for this.

Storage Allocation

The assignment of elements of a particular set M underlying a p-set $\mathcal{S} = \prod_{x \in M} V_x$ to variables in a program is known as storage allocation. Computers generally impose size and accessibility limitations on M; storage allocation provides compatibility in this regard between a program and the computer it runs on. Typically we have a particular subset $M' \subset M$ of "allocatable memory," where $M' = \{c_1, \ldots, c_m\}$, and each c_i is a cell with address or index i, $1 \leq i \leq m$. Storage allocation then consists in the assignment of r_v cells to each variable v in the program, where r_v is the storage requirement of v, and is 1 for integers and real numbers, 2 for complex and double precision numbers, n for single real or integer arrays of dimension n, and so on. The address of the first of these cells (almost always the first, by convention) then becomes the value of the address function associated with that variable. The address function in turn is the fundamental function associated with any variable, and is used to produce expressions and assignments involving that variable.

Three types of storage allocation have been identified (in PL-I, for example): static, automatic, and controlled. The address function associated with any variable under static allocation is a constant function. Allocation of such variables may proceed by assigning to each variable v of this type the cells with addresses x through $x+r_v-1$, where x is the value of a counter k which is initialized to zero, and then adding r_v to the value of k so that it continuously contains the address of the first available cell. If a program uses static allocation and also non-static allocation, then the statically allocated cells may, if desired, be removed from what we specify to be our allocatable memory.

Controlled allocation involves the use of explicit executable statements which perform allocation. With each variable of this type there is associated an <u>address cell</u> which, at all times, contains the address of the variable -- that is, the address of the first of the r_v cells allocated to the variable v. If the address cell of v is a_v, and $r_v = 1$, the address function for v is then $a(S) = S(a_v)$. For a single real or integer array v of dimension <u>n</u>, where $r_v = n$, a subscripted variable use involves v and an integer p-function e, and the associated address expression is $f(S) = S(x) + e(S)$. The address cell is itself subject to static allocation, at least if the name of the variable is constant, as it is in algebraic languages (but not, for example, in SNOBOL, where new variable names may be invented dynamically by programs). The contents of the address cell are changed by allocation statements, which are executable. There are also de-allocation statements, which do not change the contents of the address cells directly, but keep track of the available space for future allocation.

Automatic allocation is a concept that was used in ALGOL and has been carried over to PL-I and various other languages. It involves the interpretation of declarations in a block of the given program as executable allocation statements, and the process of exit from that block as an executable de-allocation statement. It allows some of the r_v to be variable, and with each v such that r_v is variable (in a given block, other than the first such) there is associated an address cell; but if r_v is constant, or if v is the first variable in its block such that r_v is variable, no address cell need be assigned and access to variables is therefore faster than it is using controlled allocation. Instead of an address cell for each variable, we have an address cell for each block level.

We shall denote the cell for block level i by B_i, $i \geq 0$; in addition, we have another special cell L which gives the current block level plus one. The B_i thus form a stack; this stack is "potentially infinite," although usually in verifying an algorithm in a particular language we find that either an arbitrary maximum number of B_i has been given, or else that the B_i are themselves allocated, using linked allocation.

Initially, when the program is started, L and B_0 are set to zero. Every time we enter a block, L is increased by one, to the new value $\underline{i}$, and B_i is set to B_{i-1} plus the sum of all the r_v for variables v declared in this block. Every time we leave a block, L is decreased by one. Thus the "topmost" B_i (where i is the current value of L) always gives us the address of the first available cell. The address function associated with a variable v such that r_v is constant, where v is defined in a block at level i (i. e., with L having the value i), is then $f(S) = S(B_i) + k_v$, where the constants k_v are chosen for the variables v in much the same way as in static allocation. This is true whether v is being referenced in the given block at level $\underline{i}$ or in a subblock (or sub-subblock, etc.) at a level $\underline{j} \geq \underline{i}$. The variable name v may be associated with the block, allowing different variables in the same program to have the same name provided that each of them is defined in a different block.

A variable r_v must depend on the values of variables defined in blocks $\underline{j}$ for $\underline{j} < \underline{i}$. The address cell for the corresponding v is set in the following way. Initially, when the block is entered at level i, B_i is set to B_{i-1} plus the sum $\underline{s}$ of all <u>constant</u> r_v for variables v declared in this block. We now proceed to treat those v for which r_v is variable. The first such v has an associated index

k_v = s which is constant, and therefore it does not require an address cell. Having treated this v, we now increase B_i by r_v. If there are any more remaining v with corresponding variable r_v, we treat each of these in turn by setting the corresponding address cell to the current value of B_i and then increasing B_i by the corresponding r_v. When this process is finished, B_i will contain the address of the first available cell.

<u>Standard Subroutines</u>

The line between a section of a program and an internal subroutine of that program is not sharply defined. Often a program will have "subroutines" which look more like sections than subroutines. We now define a type of subroutine which is entered, and which exits, only in the "normal" way.

Let P be a program on $\mathcal{J}' \subset \mathcal{S} \times P$, where $\mathcal{S} = \prod_{x \in M} V_x$, and let $x \in M$ be such that $V_x = P$. A statement $F_i = (P_i, N_i)$ of P such that $N_i(S) = S(x)$ is an <u>indirect transfer to x</u>, and <u>x</u> is an <u>indirect transfer variable</u>. In this case, if P is taken as a program on $\mathcal{S}$, there will be a link $F_i \rightarrow F_j$ for <u>every</u> $F_j \in P$. Normally, a program with an indirect transfer to <u>x</u> will be equipped with a global condition restricting the values of <u>x</u> to a certain subset of P, namely the set U of all possible return addresses. The graph of P as a program on the universal condition $\mathcal{J}'$ including this global condition will contain only those links $F_i \rightarrow F_j$ (for this F_i) where $F_j \in U$.

Let $\mathcal{D}$ be a decomposition of P, let $D_i \in \mathcal{D}$, and let $x_i \in M$. Suppose that for each $F_{ij} = (P_{ij}, N_{ij}) \in D_i$ we have $N_{ij}(S) \notin D_i$ implies $N_{ij}(S) = S(x_i)$. Then D_i is a <u>standard subroutine of P at level 0</u> and x_i is the <u>return address variable</u> of D_i. If D_i is complete, its exit is an indirect transfer to its return address variable. If D_i is not complete, it may have various "conditional indirect transfers." An instruction $F_k = (P_k, N_k) \notin D_i$ <u>calls D_i at $F_i \in D_i$ with return to $F_m \notin D_i$</u> if $N_k(S) \equiv F_i$ and $P_k(S) = S'$ where $S'(x_i) \equiv F_m$. More generally, D_i is a <u>standard subroutine of P at level k</u> if for each $F_{ij} = (P_{ij}, N_{ij}) \in D_i$, we have $N_{ij}(S) \notin D_i$ implies either $N_{ij}(S) = S(x_i)$ or F_{ij} calls a standard subroutine $D_j \in \mathcal{D}$ at level $k' < k$ at its entry with return to some other statement of D_i; and, again, x_i is the return address variable of D_i.

Standard subroutines in this sense are non-recursive and cannot have error exits. If a subroutine which does have error exits is used in a program, it can be treated as a standard subroutine by imposing a consistent universal condition which guarantees that the error exits will never be taken; the links to these may then be eliminated from the graph of the given program. If $F_i = (P_i, N_i)$ calls D_j at F_j and $\rho(P_i) = \{x_j\}$, then F_i is thereby completely determined, and we write $F_i = \{\text{CALL } F_j\}$. Such a call statement, of course, does not pass any parameters.

The simplest way of handling parameters to a subroutine is by assigning them values when the subroutine is called. Using this method, we associate certain variables $x_{i1}, \ldots, x_{im_i} \in M$ with D_i, and call them the <u>assigned</u> <u>parameters</u> of D_i. If $e_{ij}: \mathcal{S} \rightarrow V_{x_{ij}}$ for $1 \leq j \leq m_i$, the <u>assigning</u> <u>call</u> <u>statement</u> $\{\text{CALL } F_i(e_{i1}, \ldots, e_{im_i})\}$ consists of the composition, in order, of the assignments $\{x_{ij} \leftarrow e_{ij}\}$, $1 \leq j \leq m_i$, followed by $\{\text{CALL } F_i\}$ is defined above. This method of passing parameters is always sufficient whenever the parameters are called by value, as is the case, for example, in SNOBOL 4 (although we have not, as mentioned earlier, taken recursion into account). Calling by value does not allow us to return values of parameters, but we can easily modify our scheme to allow this by introducing <u>returned</u> <u>parameters</u> $y_{i1}, \ldots, y_{in_i}$ associated with D_i. If a_{ij}: $\mathcal{S} \rightarrow M$ for $1 \leq j \leq n_i$, and $a_{ij}(S) = z$ implies $V_z \subset V_{y_{ij}}$ for all S and all j, $1 \leq j \leq n_i$, then the <u>assigning-returning</u> <u>call</u> <u>statement</u> $\{\text{CALL } F_i(e_{i1}, \ldots, e_{im_i}; a_{i1}, \ldots, a_{in_i})\}$ consists of the composition of $\{\text{CALL } F_i(e_{i1}, \ldots, e_{im_i})\}$, as above, with the assignments $\{a_{ij} \leftarrow y_{ij}\}$ in order, for $1 \leq j \leq n_i$. It is quite permissible for the same variable to occur both as x_{ia} and as y_{ib}, $1 \leq a \leq n_i$, $1 \leq b \leq n_i$, so that a variable may be either used or returned or both. This type of call is essentially that which is used in JOVIAL.

It is always immediately allowable for a parameter of a subroutine to be a <u>label</u> <u>parameter</u> -- i. e., an element $x \in M$ with $V_x = P$. If $S(x) = F_i \in P$ is the entry of a subroutine D_i, and if we have uniquely associated a (possibly null) set of formal parameters with D_i, our label parameters can indicate subroutines; if they are restricted to indicating subroutines, they can then be used in this way, so that, for example, we can have a call statement which calls $S(x)$. A parameter of this type may be called a <u>functional</u> <u>parameter</u>. In some languages, such as SNOBOL, it is perfectly permissible for more than one function (with its own set of parameters) to be associated with the same starting instruction $F_i \in P$; in such a case we will have a set X of all possible subroutines, where a subroutine consists of a starting location together with a set of formal parameters; a functional parameter in such a system is an element $x \in M$ with $V_x = X$. It has long been known that any system capable of handling functional parameters is automatically capable of handling parameters called by name in the ALGOL sense. In place of a parameter that is to be called by name, we simply substitute a functional parameter which references a subroutine that computes it (this is sometimes known as Jensen's device). Thus assigning-returning call statements may be extended to cover all of the normal modes of calling subroutine parameters.

Most languages, however, use <u>calling</u> <u>sequences</u>, rather than assigning and returning. Let D_i be a standard subroutine of P, with <u>n</u> arguments $a_{i1}, \ldots, a_{im_i}$. A <u>calling</u> <u>sequence</u> for D_i is an m_i-tuple $(x_{i1}, \ldots, x_{im_i})$ of actual parameters, and an element $x \in M$ such that V_x is the set H_i of all calling sequences for D_i is a <u>calling</u> <u>sequence</u> <u>indicator</u> for D_i. The sequence call statement $\{\text{CALL } F_i(h_\lambda)\}$, for $h_\lambda \in H_i$, is the function $F_j = (P_j, N_j)$ where $N_j(\lambda) \equiv F_i$ and $P_j =$

$\{F_m \leftarrow x_i\} \circ \{h_k \leftarrow q_i\}$, where $\{\text{CALL } F_i(h_k)\}$ returns to $F_m \notin D_i$, x_i is the return address variable of D_i, and q_i is the calling sequence indicator for D_i.

The main advantage of calling sequences over assigning and returning is that the calling statement using a calling sequence does not involve so many assignments (and can therefore, presumably, be processed by the computer in a shorter time). The disadvantage of calling sequences is that the formal parameters are more complex. An assigned formal value parameter, for example, is simply an element $x_{ij} \in M$ with the usual associated program function p such that $p(S) = S(x_{ij})$; a <u>sequence parameter</u>, on the other hand, if it is a value parameter x_{ij}, has a program function p given by $p(S) = h_k(j)$ where $h_k = S(q_i)$. That is, we take the value $S(q_i)$, which is an m_i-tuple specifying a calling sequence; the j-th element of this calling sequence is then the desired value. The elements x_{ij} of a calling sequence $h_k = (x_{i1}, \ldots, x_{im_i})$ do not all have to be values; in fact, their form is determined by the specification given with the subroutine definition as to whether its various parameters are to be called by value, by location, or by name. For a call by value, x_{ij} is simply a member of the corresponding type set. For a call by location, x_{ij} is an address function, and the address function <u>a</u> of the corresponding parameter is given by $a(S) = f(S)$, where $f = h_k(j)$ (where, as before, $h_k = S(q_i)$). For a call by name, x_{ij} is a procedure, as before.

Verification of FORTRAN Programs

We shall now proceed to verify a number of examples of FORTRAN programs. We shall actually be verifying *sections* of FORTRAN programs; these may then be included in complete FORTRAN programs. Each exit statement will be denoted by CONTINUE; if the given section is included in a larger program, CONTINUE will ordinarily be replaced by the first statement of the next section, or by RETURN, STOP, or CALL EXIT.

Each of the given sections may be factored into subsections; these subsections may then be further factored. The subsections are defined by comment cards which contain C*jx* beginning in column 1, where *j* is an integer denoting the level of subsection and *x* is an optional identifier. The main section being verified is taken as level zero. Each section at level *n* may be broken up into subsections at level *n*+1. Each C*j* card defines a section at level *j* which is made up of the contiguous executable statements ranging from this card to the next C*k* or C*kx* card for $k \leq j$. Each C*jx* card for constant *j* and *x* defines a section at level *j* which is not made up of contiguous executable statements, but is the union of several collections of contiguous statements each of which ranges from a particular C*jx* card to the next C*k* or C*kx* card for $k \leq j$.

For any section which has a controlled expression *e*, this expression appears after C*j* or C*jx*, separated from this by a blank. Whether or not there is a controlled expression, a comma may follow; after this comma, there is either the word GLOBAL, followed by a global condition or conditions for this section separated by commas, or the word COND, followed by preconditions for the first executable statement of this section separated by commas, or both. On any such card, if the last non-blank character is a comma, a continuation card is to follow, which may be any comment card and which is scanned following its first blank. Preconditions for statements which are not the first statement of a section or a contiguous piece of a section may be given on a comment card with the word COND in columns 1-4.

In the proofs of consistency we make use of a path list. All paths relevant to junction consistency are enumerated in order of their initial statements, and, within this, in order of the results of their branch statements, successively. For each path we give its starting statement, the assignments and conditions along the path, and its final statement. Each junction statement, which must be numbered, is given by its statement number in parentheses; for the initial and terminal statements, which may not be numbered, we give expressions of the form (i)+j or (i)-j, which refer to the statement obtained by counting forward or backward, respectively, j executable statements from the statement numbered i. Assignments are given as they stand; for a conditional statement, we give the condition which is satisfied if the particular branch appearing within the path is taken. This condition appears in parentheses to identify it as a condition. After the path list, we give the proof of consistency; in this proof, the value which a variable assumes at the beginning of any path is denoted by an underlined lower case representation. Thus the initial value along a path of I is i, of KAPPA is kappa, etc.

The proof of normal progress of an algorithm consists in verifying, for each section, either that its graph is ordered or that the given controlled expression is actually a controlled expression according to the definition given earlier.

For the purposes of the proofs, we define a junction structure to be the sufficient substructure consisting of all junction statements of the given program, together with initial and terminal statements. A junction path is a control path of this structure.

__EXAMPLE 1__ -- __Inner Product__. The following routine sets S equal to the inner product of the vectors A and B of dimension __m__. In the comments, we use SUM(F(K), K, A, B) to denote $\sum_{K=A}^{B}$ F(K); thus we have SUM(F(K), K, A, A-1) = 0 and SUM(F(K), K, A, B) = F(B) + SUM(F(K), K, A, B-1) for B $\geq$ A.

```
CO, GLOBAL N .GT. 0, N .LE. m
      REAL A(m), B(m)
C1
      I = 1
      S = 0
C1 I, COND S = SUM(A(K)*B(K), K, 1, I-1), I .GT. 0, I .LE. N
3     S = S + A(I)*B(I)
      I = I + 1
      IF (I .LE. N) GO TO 3
C1, COND S = SUM(A(K)*B(K), K, 1, N)
      CONTINUE
```

Under the global conditions N $>$ 0 and N $\leq$ __m__, the single terminal condition is that S has been properly calculated as the inner product. There is one junction statement, namely the statement 3, and thus a junction structure has been completely specified. The semantics of FORTRAN tell us that I and N are integer variables and that A(1), ..., A(N), B(1), ..., B(N) are real variables. Statement 3, which references A(I) and B(I), is valid only under the conditions I .GT. 0 and I .LE. __m__, and these are given by the preconditions at this statement and by the global conditions.

__Proof of Consistency__. There are three junction paths: (3)-2, (3)-1, (3); (3), (3)+1, (3)+2, (3); and (3), (3)+1, (3)+2, (3)+3. The path list is as follows:

1. (3)-2; I = 1; S = 0; (3).

2. (3); S = S + A(I)*B(I); I = I + 1; (I .LE. N); (3).

3. (3); S = S + A(I)*B(I); I = I + 1; (I .GT. N); (3)+3.

For path 1, we have by definition S = SUM(A(K)*B(K), K, 1, 0) = 0; I .GT. 0 because I = 1; and I .LE. N because N .GT. 0, by the global condition, and I = 1 (and the fact that N is an integer). For path 2, the final condition I .LE. N follows from the conditional (I. LE. N) after I has been modified. We have I = $\underline{i}$+1 $>$ 0 because $\underline{i} > 0$, and S = $\underline{s}$ + A($\underline{i}$)*B($\underline{i}$) = SUM(A(K)*B(K), K, 1, $\underline{i}$-1) + A($\underline{i}$)*B($\underline{i}$) = SUM(A(K)*B(K), K, 1, $\underline{i}$) = SUM(A(K)*B(K), K, 1, I-1). For path 3, we have I = $\underline{i}$+1, $\underline{i}+1 > \underline{n}$ but $\underline{i} \leq \underline{n}$, which, since $\underline{i}$ is an integer, implies $\underline{i} = \underline{n}$; and, just as in path 2, we end with S = SUM(A(K)*B(K), K, 1, $\underline{i}$), which is thus SUM(A(K)*B(K), K, 1, N).

The global conditions N .GT. 0 and N .LE. $\underline{m}$ involve only N, which is not changed by any statement of the computation; they are therefore truly global. This completes the proof of consistency.

<u>Proof of Normal Progress</u>. The CO factor program and the first and third C1 sections have ordered graphs. The second C1 section has the controlled variable I; we verify that it is actually a controlled variable by noting that the assignment I = I + 1 is the only statement of this section for which I is in the effective range, and for this statement P_i we have S'(I) $>$ S(I) where S' = P_i(S); and that by removing this statement from the graph of the section we obtain a graph with no directed cycles. We may now apply the second controlled expression theorem, because by removing the IF statement from the graph of the program we also obtain an acyclic graph. This completes the verification.

For an extremely simple program such as this one, there is a well-known alternative method of verification, which may be called "the enumeration method." It consists of specifying the computation sequences explicitly. In this case, the length of any computation sequence is 3N+3, and its elements may be given as

S_1: I = 1

S_2: I = 1, S = 0

S_{3i}: I = i, S = $\sum_{k=1}^{i}$(A(k)*B(k)) $(1 \leq i \leq N)$

S_{3i+1}: I = i+1, S = $\sum_{k=1}^{i}$(A(k)*B(k)) $(1 \leq i \leq N)$

$S_{3i+2} = S_{3i+1}$ $(1 \leq i \leq N)$

S_{3N+3}: S = $\sum_{k=1}^{N}$(A(k)*B(k))

<u>EXAMPLE 2</u> -- <u>Euclid's Algorithm</u>. The following routine uses Euclid's algorithm to find the greatest common divisor (GCD) of the positive integers M and N.

```
CO, GLOBAL M .GT. 0, N .GT. 0
C1
          I = M
          J = N
C1 -(I+J), COND GCD(M, N) = GCD(I, J),
C                    0 .LT. I, I .LE. M, 0 .LT. J, J .LE. N
1         IF (I - J) 2, 4, 3
2         J = J - I
          GO TO 1
3         I = I - J
          GO TO 1
C1, COND I = GCD(M, N)
4         CONTINUE
```

Under the global conditions $M > 0$ and $N > 0$, the single terminal condition is that I has been properly calculated as the GCD. There is one junction statement, namely the statement 1 (notice, in particular, that statements 2 and 3 are not junction statements), and thus a junction structure has been completely specified. It is assumed that I, J, M, and N are integer variables. In the proof we need certain elementary facts about the GCD function.

<u>LEMMA</u>. Let GCD(M, N) be the greatest common divisor of the positive integers M and N -- i. e., the greatest integer I such that I divides M and I divides N. Then:

(a) If $a > 0$, then $GCD(a, a) = a$.

(b) If $a > 0$, $b > 0$, and $b-a > 0$, then $GCD(a,b) = GCD(a,b-a)$.

(c) If $a > 0$, $b > 0$, and $a-b > 0$, then $GCD(a,b) = GCD(a-b,b)$.

PROOF. (a) a divides a, and nothing larger than a may divide a. For (b), we notice that GCD(a, b) divides a and b and therefore divides b-a; likewise GCD(a, b-a) divides a and b-a and therefore divides (b-a)+a = b. Thus GCD(a, b) $\leq$ GCD(a, b-a) $\leq$ GCD(a, b), and this means that GCD(a, b) = GCD(a, b-a). The proof of (c) is similar to that of (b).

Proof of Consistency. There are four junction paths: (1)-2, (1)-1, (1); (1), (2), (2)+1, (1); (1), (3), (3)+1, (1); (1), (4). The path list is as follows:

1. (1)-2; I = M; J = N; (1).
2. (1); (I .LT. J); J = J - I; (1). (1).
3. (1); (I .GT. J); I = I - J; (1).
4. (1); (I=J); (4).

The first path may be immediately verified. The conditions 0 .LT. I and I .LE. M in path 2 and the conditions 0 .LT. J and J .LE. N in path 3 follow because the respective variables are not changed within the given paths. The condition J .LE. N in path 2 follows from J = $\underline{j}$-$\underline{i}$ < $\underline{j}$ (from 0 < I = $\underline{i}$) $\leq$ $\underline{n}$ = N; the condition I .LE. M in path 3 follows from I = $\underline{i}$-$\underline{j}$ < $\underline{i}$ (from 0 < J = $\underline{j}$) $\leq$ $\underline{m}$ = M. The condition 0 .LT. J in path 2 follows from 0 < $\underline{j}$-$\underline{i}$ (from I-J .LT. 0) = J; the condition 0 .LT. I in path 3 follows from 0 < $\underline{i}$-$\underline{j}$ (from I-J .GT. 0) = I. The condition GCD(M, N) = GCD(I, J) follows, in path 2, from GCD(M, N) = GCD($\underline{i}$, $\underline{j}$) = GCD($\underline{i}$, $\underline{j}$-$\underline{i}$) (by (b) of the lemma) = GCD(I, J), and in path 3 from GCD(M, N) = GCD($\underline{i}$, $\underline{j}$) = GCD($\underline{i}$-$\underline{j}$, $\underline{j}$) (by (c) of the lemma) = GCD(I, J). The condition I = GCD(M, N) in path 4 follows from GCD(M, N) = GCD($\underline{i}$, $\underline{j}$) = GCD($\underline{i}$, $\underline{i}$) (from (I=J)) = $\underline{i}$ (by (a) of the lemma) = I. The global conditions M .GT. 0 and N .GT. 0 involve only M and N, which are not changed by any statement of the computation; they are therefore truly global. This completes the proof of consistency.

We remark that the conditions I .LE. M and J .LE. N at step 1 were not necessary in order to set up a consistent precondition structure. They will, however, be used to verify that the algorithm terminates.

<u>Proof of Normal Progress</u>. The C0 factor program and the first and third C1 sections have ordered graphs. The second C1 section has the controlled expression -(I+J); we verify that it is actually a controlled expression by noting that the assignments I = I - J and J = J - I are the only statements of this section whose effective ranges have non-empty intersection with the effective domain {I, J} of the expression -(I+J), and that each of these statements increases the value of -(I+J); and that by removing these statements from the graph of the program we obtain a graph with no directed cycles. We may now apply the corollary to the second controlled expression theorem, because at statement number 1 we have -(I+J) < 0, and removing this statement leaves a graph with no directed cycles. This completes the verification.

The enumeration method of the preceding example does not work in this example. Indeed there is no simple formula which gives, for arbitrary M and N, the number of steps this algorithm will take. (There is a recurrence relation, of course, which gives this number.) We note that there exists a verification of Euclid's algorithm in [6]; the version given there is much more complex than the one given here. In this connection it must be remembered that an algorithm may be programmed in different ways for different computers and in different languages, and must be verified separately each time. The size of the proof given here is smaller than that of [6], [illegible] algorithm is much simpler; also, it is actually faster in some cases, as when calculating the GCD of two adjacent Fibonacci numbers. It would, however, give rise to certain difficulties in practice, the most obvious of which is that it would take <u>n</u> steps to calculate the GCD of the numbers <u>n</u> and 1.

EXAMPLE 3 -- Merging. The following routine merges the sorted arrays A and B, of lengths M and N respectively, into a new sorted array C of length M+N. All arrays are assumed to be sorted in ascending order. In the comments, we use ASC(P, Q) for the state condition in which the first Q elements of the array P are sorted in ascending order; thus ASC(P, 1) is always true, ASC(P, Q+1) = (ASC(P,Q) and P(Q+1) $\geq$ P(Q)), for Q $\geq$ 1, and ASC(P, Q) implies ASC(P, R) for 1 $\leq$ R $<$ Q. We use PERM(X, Y) to denote the state condition that there is a one-to-one correspondence **f**, such that S(**x**) = S(**y**) where S belongs to the given state condition and **y** = f(**x**), between A(1), ..., A(X), B(1), ..., B(Y), on the one hand, and C(1), ..., C(X+Y), on the other (this is the condition that the elements of the array C are the elements of A and B in some new order). We use MERGE(X, Y) to denote the state condition that the first X elements of A and the first Y elements of B have been merged into the first X+Y elements of C; thus MERGE(X, Y) = (PERM(X, Y) and ASC(C, X+Y)).

```
CO, GLOBAL ASC(A, M), ASC(A, N), M .GT. 0, N .GT. 0
C1
         I = 1
         J = 1
         K = 0
C1 K, COND PERM(I-1, J-1), K=I+J-2, I .LE. M, J .LE. N,
C K=0 OR (ASC(C, K), C(K) .LE. B(J), C(K) .LE. A(I), K .GT. 0)
1        K = K + 1
         IF (A(I) .LT. B(J)) GO TO 3
         C(K) = B(J)
         J = J + 1
         IF (J .LE. N) GO TO 1
```

```
COND PERM(I-1, J-1), K=I+J-2, I .LE. M, J = N + 1,
C ASC(C, K), C(K) .LE. A(I), K .GT. 0
2       K = K + 1
        C(K) = A(I)
        I = I + 1
        IF (I .LE. M) GO TO 2
        GO TO 5
3       C(K) = A(I)
        I = I + 1
        IF (I .LE. M) GO TO 1
COND PERM(I-1, J-1), K=I+J-2, I = M+1, J .LE. N,
C ASC(C, K), C(K) .LE. B(J), K .GT. 0
4       K = K + 1
        C(K) = B(J)
        J = J + 1
        IF (J .LE. N) GO TO 4
C1, COND PERM(I-1,J-1), K=I+J-2, I = M+1, J = N-1,
C ASC(C, K), MERGE(M, N)
5       CONTINUE
```

Under the global conditions M .GT. 0, N .GT. 0, ASC(A, M), and ASC(B, N) -- i. e., assuming that the arrays A and B of positive length are initially in ascending order -- there are several terminal conditions, among which is MERGE(M, N), which, by our definition of MERGE, implies that the arrays A and B have been properly merged into the array C. Statements 1, 2, and 4 are junction statements (and statement 3 is not); therefore, we have a junction structure. It is assumed that I, J, K, M, and N are integer variables and that A(1), ..., A(M), B(1), ..., B(N), C(1), ..., C(M+N) are real variables (this assumption depends on the values of M and N and the proper dimensioning of A, B, and C).

<u>Proof of Consistency</u>. The conditions M .GT. 0, N .GT. 0, ASC(A, M), and ASC(B, N) are global because they involve only M and N and the elements of the arrays A and B, and none of these are changed by any statement in the program. There are nine junction paths: (1)-3, (1)-2, (1)-1, (1); (1), (1)+1, (1)+2, (1)+3, (1)+4, (1); (1), (1)+1, (1)+2, (1)+3, (1)+4, (2); (1), (1)+1, (3), (3)+1, (3)+2, (1); (1), (1)+1, (3), (3)+1, (3)+2, (4); (2), (2)+1, (2)+2, (2)+3, (2); (2), (2)+1, (2)+2, (2)+3, (2)+4, (5); (4), (4)+1, (4)+2, (4)+3, (4); and (4), (4)+1, (4)+2, (4)+3, (5). The path list is as follows:

1. (1)-3; I = 1; J = 1; K = 0; (1).
2. (1); K = K+1; (A(I) .GE. B(J)); C(K) = B(J); J = J+1; (J .LE. N); (1).
3. (1); K = K+1; (A(I) .GE. B(J)); C(K) = B(J); J = J+1; (J .GT. N); (2).
4. (1); K = K+1; (A(I) .LT. B(J)); C(K) = A(I); I = I+1; (I .LE. M); (1).
5. (1); K = K+1; (A(I) .LT. B(J)); C(K) = A(I); I = I+1; (I .GT. M); (4).
6. (2); K = K+1; C(K) = A(I); I = I+1; (I .LE. M); (2).
7. (2); K = K+1; C(K) = A(I); I = I+1; (I .GT. M); (5).
8. (4); K = K+1; C(K) = B(J); J = J+1; (J .LE. N); (4).
9. (4); K = K+1; C(K) = B(J); J = J+1; (J .GT. N); (5).

The first path may be immediately verified, and, in particular, K=0, so that the second comment line before statement number 1 is satisfied. The following conditions hold at the beginning and end of the same path because the variables in them are not changed within that path: I .LE. M, paths 2 and 3; J .LE. N, paths 4 and 5;

<u>Proof of Consistency</u>. The conditions M .GT. 0, N .GT. 0, ASC(A, M), and ASC(B, N) are global because they involve only M and N and the elements of the arrays A and B, and none of these are changed by any statement in the program. There are nine junction paths: (1)-3, (1)-2, (1)-1, (1); (1), (1)+1, (1)+2, (1)+3, (1)+4, (1); (1), (1)+1, (1)+2, (1)+3, (1)+4, (2); (1), (1)+1, (3), (3)+1, (3)+2, (1); (1), (1)+1, (3), (3)+1, (3)+2, (4); (2), (2)+1, (2)+2, (2)+3, (2); (2), (2)+1, (2)+2, (2)+3, (2)+4, (5); (4), (4)+1, (4)+2, (4)+3, (4); and (4), (4)+1, (4)+2, (4)+3, (5). The path list is as follows:

1. (1)-3; I = 1; J = 1; K = 0; (1).

2. (1); K = K+1; (A(I) .GE. B(J)); C(K) = B(J); J = J+1; (J .LE. N); (1).

3. (1); K = K+1; (A(I) .GE. B(J)); C(K) = B(J); J = J+1; (J .GT. N); (2).

4. (1); K = K+1; (A(I) .LT. B(J)); C(K) = A(I); I = I+1; (I .LE. M); (1).

5. (1); K = K+1; (A(I) .LT. B(J)); C(K) = A(I); I = I+1; (I .GT. M); (4).

6. (2); K = K+1; C(K) = A(I); I = I+1; (I .LE. M); (2).

7. (2); K = K+1; C(K) = A(I); I = I+1; (I .GT. M); (5).

8. (4); K = K+1; C(K) = B(J); J = J+1; (J .LE. N); (4).

9. (4); K = K+1; C(K) = B(J); J = J+1; (J .GT. N); (5).

The first path may be immediately verified, and, in particular, K=0, so that the second comment line before statement number 1 is satisfied. The following conditions hold at the beginning and end of the same path because the variables in them are not changed within that path: I .LE. M, paths 2 and 3; J .LE. N, paths 4 and 5;

J = N+1, paths 6 and 7; I = M+1, paths 8 and 9; C(K) .LE. A(I), path 2; and C(K) .LE. B(J), path 4. The condition K .GT. 0 in paths 2 through 9 follows from the initial condition K = 0 or K .GT. 0 in each case and the fact that each of these paths contains the statement K = K+1. The condition J .LE. N in paths 2 and 8 and the condition I .LE. M in paths 4 and 6 follow from the corresponding conditions (J .LE. N and I .LE. M) appearing within these paths. The condition J = N+1 in paths 3 and 9 and the condition I = M+1 in paths 5 and 7 follow from the corresponding conditions (J .GT. N and I .GT. M) appearing in these paths and the conditions J = $\underline{j}$+1, $\underline{j} \leq$ N and I = $\underline{i}$+1, $\underline{i} \leq$ M respectively. The condition K=I+J-2 follows from I = $\underline{i}$+1, J = $\underline{j}$, K = $\underline{k}$+1 = ($\underline{i}$+$\underline{j}$-2)+1 = ($\underline{i}$+1)+j-2 = I+J-2 in paths 4, 5, 6, and 7 and from I = $\underline{i}$, J = $\underline{j}$+1, K = $\underline{k}$+1 = ($\underline{i}$+$\underline{j}$-2)+1 = $\underline{i}$+($\underline{j}$+1)-2 = I+J-2 in paths 2, 3, 8, and 9. The proof of ASC(C, K) in all paths separates into two cases. If $\underline{k}$ = 0, then K = $\underline{k}$+1 = 1 and ASC(C, K) is always true. Otherwise, we must derive ASC(C, $\underline{k}$+1) from ASC(C, $\underline{k}$). This follows from C($\underline{k}$) .LE. B($\underline{j}$) = C($\underline{k}$+1) (by the statement C(K) = B(J) in its given position) in paths 2, 3, 8, and 9, and from C($\underline{k}$) .LE. A($\underline{i}$) = C($\underline{k}$+1) (by the statement C(K) = A(I) in its given position) in paths 4, 5, 6, and 7. The condition C(K) .LE. A(I) in paths 4 and 6 follows from C(K) = C($\underline{k}$+1) = B($\underline{j}$) .LE. B($\underline{j}$+1) (from the global condition ASC(B, N) with $\underline{j} \leq$ N) = B(J). The condition C(K) .LE. B(J) in paths 2 and 8 follows from C(K) = C($\underline{k}$=1) = A($\underline{i}$) .LE. A($\underline{i}$+1) (from the global condition ASC(A, M) with $\underline{i} \leq$ M) = A(I). The condition C(K) .LE. A(I) in path 3 follows from C(K) = C($\underline{k}$+1) = B($\underline{j}$) .LE. A($\underline{i}$) (from the condition (A(I) .GE. B(J)) or (A($\underline{i}$) $\geq$ B($\underline{j}$))) = A(I); the condition C(K) .LE. B(J) in path 5 follows from C(K) = C($\underline{k}$+1) = A($\underline{i}$) $<$ B($\underline{j}$) (from the condition (A(I) .LT. B(J)) or (A($\underline{i}$) $<$ B($\underline{j}$))) = B(J).

It remains to verify the condition PERM(I-1, J-1). This will be done for paths 2, 3, 8, and 9; a similar argument holds for paths 4, 5, 6, and 7, with A and B reversed, I and J reversed, and M and N reversed. We are given PERM($\underline{i}$-1, $\underline{j}$-1), and, therefore, a one-to-one correspondence $\underline{f}$ whose domain includes A(1), ..., A($\underline{i}$-1) and B(1), ..., B($\underline{j}$-1) and whose range includes C(1) through C($\underline{k}$). We must show PERM($\underline{i}$-1, $\underline{j}$), and to do this we construct a new function $\underline{f}'$ by setting $\underline{f}'(\underline{v}) = \underline{f}(\underline{v})$ for $\underline{v}$ in the domain of $\underline{f}$, and $\underline{f}'(B(\underline{j})) = C(\underline{k}+1)$. Thus $\underline{f}'$ is a one-to-one correspondence whose domain is A(1) through A($\underline{i}$-1) and B(1) through B($\underline{j}$), and whose range is C(1) through C($\underline{k}$+1); it satisfies $S(\underline{x}) = S(\underline{y})$ where $\underline{y}$ is the image of $\underline{x}$, because $\underline{f}$ does so and because $S(B(\underline{j})) = S(C(\underline{k}+1))$ from the statement C(K) = B(J) carried out when K = $\underline{k}$+1 and J = $\underline{j}$. This completes the proof of consistency.

<u>Proof of Normal Progress</u>. The C0 factor program and the first and third C1 sections have ordered graphs. The second C1 section has the controlled variable K; we verify that it is actually a controlled variable by noting that the various assignments of the form K = K+1 are the only statements in this section for which K is in the effective range, and if P_k is any of these statements we have $S'(K) > S(K)$ where $S' = P_k(S)$; and that by removing all these statements from the graph of the section we obtain a graph with no directed cycles. We may now apply the corollary to the second controlled expression theorem, because at each statement of our junction structure the given preconditions imply $K \leq M+N$, and M+N involves only variables which are not changed in the program. This completes the verification.

R E F E R E N C E S

1. d'Imperio, M. E., Data structures and their representation in storage, Annual Review in Automatic Programming 5 (1969), pp. 1-75.

2. Elgot, C. C., and Abraham Robinson, Random access stored program computers -- an approach to programming languages, Journal of the ACM, 11, 4 (October 1964), pp. 365-399.

3. Floyd, R. W., Assigning meanings to programs, Proc. Symp. Applied Math. 19 (1967), American Mathematical Society, Providence, R. I., pp. 19-32.

4. Good, D. I., Toward a man-machine system for proving program correctness, Computation Center, University of Texas, Austin, Texas, June 1970.

5. King, J. C., A program verifier, Ph. D. Thesis, Carnegie-Mellon University, September 1969.

6. Knuth, D. E., The art of computer programming, Vol. I: Fundamental algorithms, Addison-Wesley, 1968.

7. Knuth, D. E., Semantics of context-free languages, Mathematical Systems Theory 2, 2 (June 1968), pp. 127-146.

8. London, R. L., Proving programs correct: some techniques and examples, BIT, 10, 2 (1970), pp. 168-182.

9. Lucas, P., and K. Walk, On the formal description of PL-I, Annual Review in Automatic Programming 6, 3 (1969).

10. Manna, Z., Termination of programs represented as interpreted graphs, Proc. 1970 Spring Joint Computer Conference, pp. 83-89.

11. Maurer, W. D., A theory of computer instructions, Journal of the ACM, 13, 2 (April 1966), pp. 226-235.

12. Maurer, W. D., Examples of algorithm verification, Memorandum M-291, Electronics Research Laboratory, University of California, Berkeley, January 1971.

13. McCarthy, J., Towards a mathematical science of computation, Information Processing 1962, Proc. of IFIP Congress 62, C. M. Popplewell, ed., North-Holland, Amsterdam, 1963, pp. 21-28.

A C K N O W L E D G M E N T

This work was partially supported by National Science Foundation Grant GJ-821.