Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

Introduction. This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs, including proofs of correctness, equivalence, and termination. The basis of our approach is the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever that connection is taken. To prevent an interpretation from being chosen arbitrarily, a condition is imposed on each command of the program. This condition guarantees that whenever a command is reached by way of a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. Then by induction on the number of commands executed, one sees that if a program is entered by a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. By this means, we may prove certain properties of programs, particularly properties of the form: "If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2." Proofs of termination are dealt with by showing that each step of a program decreases some entity which cannot decrease indefinitely.

These modes of proof of correctness and termination are not original; they are based on ideas of Perlis and Gorn, and may have made their earliest appearance in an unpublished paper by Gorn. The establishment of formal standards for proofs about programs in languages which admit assignments, transfer of control, etc., and the proposal that the semantics of a programming language may be defined independently of all processors for that language, by establishing standards of rigor for proofs about

[1] This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146).

programs in the language, appear to be novel, although McCarthy [1, 2] has done similar work for programming languages based on evaluation of recursive functions.

A semantic definition of a programming language, in our approach, is founded on a syntactic definition. It must specify which of the phrases in a syntactically correct program represent commands, and what conditions must be imposed on an interpretation in the neighborhood of each command.

We will demonstrate these notions, first on a flowchart language, then on fragments of ALGOL.

DEFINITIONS. A *flowchart* will be loosely defined as a directed graph with a command at each vertex, connected by *edges* (arrows) representing the possible passages of control between the commands. An edge is said to be an *entrance* to (or an *exit* from) the command c at vertex v if its destination (or origin) is v. An *interpretation* I of a flowchart is a mapping of its edges on propositions. Some, but not necessarily all, of the free variables of these propositions may be variables manipulated by the

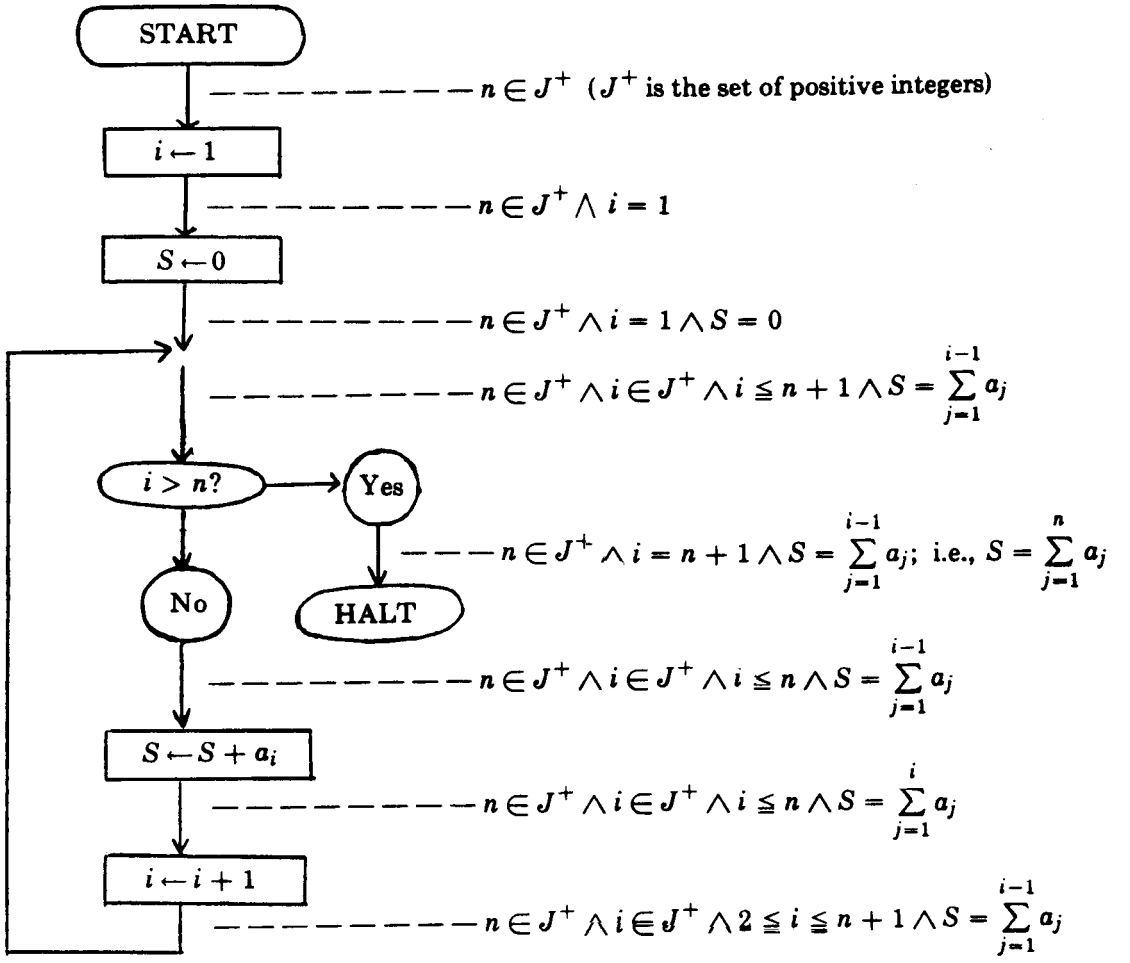

FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ $(n \geqq 0)$

program. Figure 1 gives an example of an interpretation. For any edge e, the associated proposition $I(e)$ will be called the *tag* of e. If e is an entrance (or an exit) of a command c, $I(e)$ is said to be an *antecedent* (or a *consequent*) of c.

For any command c with k entrances and l exits, we will designate the entrances to c by $a_1, a_2, \cdots, a_k$, and the exits by $b_1, b_2, \cdots, b_l$. We will designate the tag of a_i by P_i $(1 \leqq i \leqq k)$, and that of b_i by Q_i $(1 \leqq i \leqq l)$. Boldface letters will designate vectors formed in the natural way from the entities designated by the corresponding nonboldface letters: for example, $\mathbf{P}$ represents $(P_1, P_2, \cdots, P_k)$.

A *verification* of an interpretation of a flowchart is a proof that for every command c of the flowchart, if control should enter the command by an entrance a_i with P_i true, then control must leave the command, if at all, by an exit b_j with Q_j true. *A semantic definition* of a particular set of command types, then, is a rule for constructing, for any command c of one of these types, a *verification condition* $V_c(\mathbf{P}; \mathbf{Q})$ on the antecedents and consequents of c. This verification condition must be so constructed that a proof that the verification condition is satisfied for the antecedents and consequents of each command in a flowchart is a verification of the interpreted flowchart. That is, if the verification condition is satisfied, and if the tag of the entrance is true when the statement is entered, the tag of the exit selected will be true after execution of the statement.

A *counterexample* to a particular interpretation of a single command is an assignment of values (e.g., numbers in most programming languages) to the free variables of the interpretation, and a choice of entrance, such that on entry to the command, the tag of the entrance is true, but on exit, the tag of the exit is false for the (possibly altered) values of the free variables. A semantic definition is *consistent* if there is no counterexample to any interpretation of any command which satisfies its verification condition. A semantic definition is *complete* if there is a counterexample to any interpretation of any command which does not satisfy its verification condition. A semantic definition clearly must be consistent. Preferably, it should also be complete; this, however, is not always possible.

In what follows, we shall have in mind some particular deductive system D, which includes the axioms and rules of inference of the first-order predicate calculus, with equality. We shall write $\Phi_1, \Phi_2, \cdots, \Phi_n \vdash \Psi$ to mean that Ψ is a proposition deducible from $\Phi_1, \Phi_2, \cdots, \Phi_n$ and the axioms of D by the rules of inference of D. We shall designate by

$$S^{x_1, x_2, \cdots, x_n}_{f_1, f_2, \cdots, f_n}(\Phi) \text{ or, more briefly, } S^{\mathbf{x}}_{\mathbf{f}}(\Phi),$$

the result of simultaneously substituting f_i for each occurrence of x_i in Φ, after first systematically changing bound variables of Φ to avoid conflict with free variables of any f_i.

Connectives will be assumed to distribute over the components of vectors; for instance, $\mathbf{X} \wedge \mathbf{Y}$ means $(X_1 \wedge Y_1, X_2 \wedge Y_2, \cdots, X_n \wedge Y_n)$, and $\mathbf{X} \vdash \mathbf{Y}$ means $(X_1 \vdash Y_1) \wedge (X_2 \vdash Y_2) \wedge \cdots \wedge (X_n \vdash Y_n)$.

General axioms. In order for a semantic definition to be satisfactory, it must meet several requirements. These will be presented as axioms, although they may also be deduced from the assumptions of completeness and consistency, where these hold.

If $V_c(\mathbf{P}; \mathbf{Q})$ and $V_c(\mathbf{P}'; \mathbf{Q}')$, then:

AXIOM 1. $V_c(\mathbf{P} \wedge \mathbf{P}'; \mathbf{Q} \wedge \mathbf{Q}')$;

AXIOM 2. $V_c(\mathbf{P} \vee \mathbf{P}'; \mathbf{Q} \vee \mathbf{Q}')$;

AXIOM 3. $V_c((\exists x)(\mathbf{P}); (\exists x)(\mathbf{Q}))$.

Also,

AXIOM 4. If $V_c(\mathbf{P}; \mathbf{Q})$ and $\mathbf{R} \vdash \mathbf{P}$, $\mathbf{Q} \vdash \mathbf{S}$, then $V_c(\mathbf{R}; \mathbf{S})$.

COROLLARY 1. *If* $V_c(\mathbf{P}; \mathbf{Q})$ *and* $\vdash (\mathbf{P} \equiv \mathbf{R})$, $\vdash (\mathbf{Q} \equiv \mathbf{S})$, *then* $V_c(\mathbf{R}; \mathbf{S})$.

Axiom 1, for example, essentially asserts that if whenever P is true on entering command c, Q is true on exit, and whenever P' is true on entry, Q' is true on exit, then whenever both P and P' are true on entrance, both Q and Q' are true on exit. Thus Axiom 1 shows that if separate proofs exist that a program has certain properties, then these proofs may be combined into one by forming the conjunction of the several tags for each edge. Axiom 2 is useful for combining the results of a case analysis, for instance, treating several ranges of initial values of certain variables. Axiom 3 asserts that if knowing that the value of the variable x has property P before executing a command assures that the (possibly altered) value will have property Q after executing the command, then knowing that a value exists having property P before execution assures that a value exists having property Q after execution. Axiom 4 asserts that if P and Q are verifiable as antecedent and consequent for a command, then so are any stronger antecedent and weaker consequent.

To indicate how these axioms are deducible from the hypotheses of completeness and consistency for V_c, consider Axiom 1 as an example. Suppose $V_c(\mathbf{P}; \mathbf{Q})$ and $V_c(\mathbf{P}'; \mathbf{Q}')$. Consider any assignment of initial values $\mathbf{V}$ to the free variables $\mathbf{X}$ of the interpretation such that P_i is true (that is, $\vdash S^{\mathbf{X}}_{\mathbf{V}}(P_i)$) and P_i' is true. Then, if the statement is entered by a_i, the exit chosen will be some b_j such that Q_j is true at that time (that is, $\vdash S^{\mathbf{X}}_{\mathbf{W}}(Q_j)$, where $\mathbf{W}$ is the vector of final values of $\mathbf{X}$ after execution of c), and Q_j' is also true, by the assumption of consistency. Thus, there can be no counterexample to the interpretation $I(\mathbf{a}) = (\mathbf{P} \wedge \mathbf{P}')$, $I(\mathbf{b}) = (\mathbf{Q} \wedge \mathbf{Q}')$, and by the assumption of completeness, $V_c(\mathbf{P} \wedge \mathbf{P}'; \mathbf{Q} \wedge \mathbf{Q}')$.

A flowchart language. To make these notions more specific, consider a particular flowchart language with five statement types, represented pictorially as in Figure 2, having the usual interpretations as an assignment operation, a conditional branch, a join of control, a starting point for the program, and a halt for the program.

Take specifically the assignment operator $x \leftarrow f(x, \mathbf{y})$, where x is a variable and f is an expression which may contain occurrences of x and of the vector $\mathbf{y}$ of other program variables. Considering the effect of the command, it is clearly desirable that if P_1 is $(x = x_0 \wedge R)$, and Q_1 is $(x = f(x_0, \mathbf{y}) \wedge R)$, where R contains no free occurrences of x, then $V_c(P_1; Q_1)$. Applying the axioms, we shall establish a definition of $V_{x \leftarrow f(x,\mathbf{y})}$ which is complete and consistent if the underlying deductive system is, and which is, in that sense, the most general semantic definition of the assignment operator.

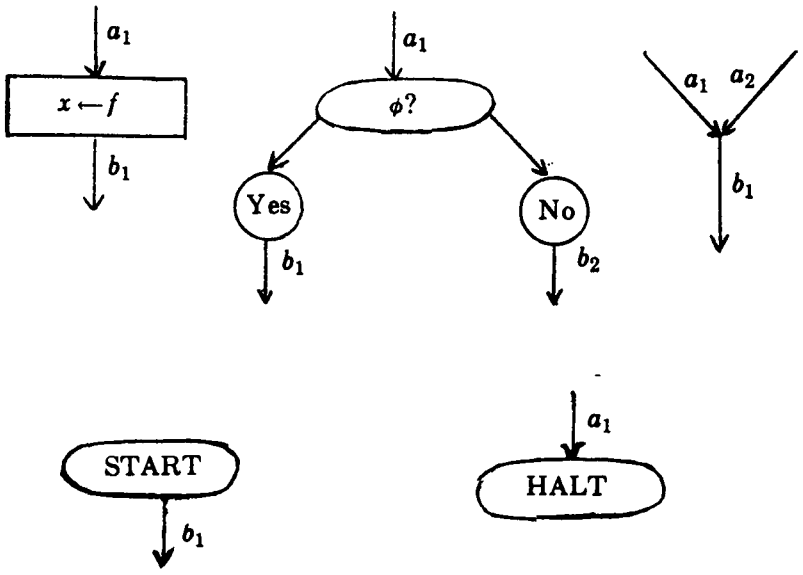

FIGURE 2

Designating the command $x \leftarrow f(x, \mathbf{y})$ by c, we apply Axiom 3 to $V_c(P_1, Q_1)$, to obtain

$$V_c((\exists x_0) P_1; (\exists x_0) Q_1).$$

Because $[(\exists x)(x = e \wedge P(x))] \equiv P(e)$, provided x does not occur free in e, we apply Corollary 1, to get $V_c(R(x, \mathbf{y}); (\exists x_0)(x = f(x_0, \mathbf{y}) \wedge R(x_0, \mathbf{y})))$.

Finally, by Corollary 1, we have

The verification condition for assignment operators.

(1) If P_1 has the form $R(x, \mathbf{y})$ and if $(\exists x_0)(x = f(x_0, \mathbf{y}) \wedge R(x_0, \mathbf{y})) \vdash Q_1$, then $V_{x \leftarrow f(x,\mathbf{y})}(P_1, Q_1)$.

Taking this as the semantic definition of $x \leftarrow f(x, \mathbf{y})$, and assuming the completeness and consistency of the deductive system D, we show that the semantic definition is complete and consistent.

To show consistency, assume that x_1 and $\mathbf{y}_1$ are initial values of x and $\mathbf{y}$ such that $\vdash R(x_1, \mathbf{y}_1)$. Then after execution of $x \leftarrow f(x, \mathbf{y})$, the values x_2 and $\mathbf{y}_2$ are such that $x = x_2 = f(x_1, \mathbf{y}_1)$, $\mathbf{y} = \mathbf{y}_2 = \mathbf{y}_1$; thus $x_2 = f(x_1, \mathbf{y}_2) \wedge R(x_1, \mathbf{y}_2)$, or $(\exists x_0)(x_2 = f(x_0, \mathbf{y}_2) \wedge R(x_0, \mathbf{y}_2))$. Designating $(\exists x_0)(x = f(x_0, \mathbf{y}) \wedge R(x_0, \mathbf{y}))$ as $T_c(R(x, \mathbf{y}))$, we have shown that upon exit from c, $S^{x\mathbf{y}}_{x_2\mathbf{y}_2}(T_c(R(x, \mathbf{y})))$ is true. Now since $T_c(R(x, \mathbf{y})) \vdash Q$, we find $\vdash S^{x\mathbf{y}}_{x_2\mathbf{y}_2}(Q)$, by the assumption of the consistency of D, so that V_c is consistent.

To show completeness, assume it false that $T_c(R(x, \mathbf{y})) \vdash Q$. Then, by the completeness of D, there is a set of values x_2 and $\mathbf{y}_2$ for x and $\mathbf{y}$ such that $S^{x\mathbf{y}}_{x_2\mathbf{y}_2}(T_c(R(x, \mathbf{y})))$ is true, but $S^{x\mathbf{y}}_{x_2\mathbf{y}_2}(Q)$ is false. Thus, $(\exists x_0)(x_2 = f(x_0, \mathbf{y}_2) \wedge R(x_0, \mathbf{y}_2))$. Let x_1 be a particular value of x_0 for which $x_2 = f(x_1, \mathbf{y}_2) \wedge R(x_1, \mathbf{y}_2)$. Now using x_1 and $\mathbf{y}_2$ as initial values for x and $\mathbf{y}$, we may generate a counterexample to the interpretation $I(a_1) = R(x, \mathbf{y})$, $I(b_1) = Q$.

Thus we have shown that V_c is complete (consistent) if D is complete (consistent). By consideration of vacuous statements such as $x \leftarrow x$, we could change each "if" to "if and only if." Thus, the semantic definition (1) we have given is the natural generalization of the original sufficient condition for verification; V_c is both necessary and sufficient.

The other command types of Figure 2 are more easily dealt with. For the branch command, $V_c(P_1; Q_1, Q_2)$ is $(P_1 \wedge \Phi \vdash Q_1) \wedge (P_1 \wedge \neg \Phi \vdash Q_2)$. For the join command, $V_c(P_1, P_2; Q_1)$ is $(P_1 \vee P_2 \vdash Q_1)$. For the start command the condition $V_c(Q_1)$, and for the halt command the condition $V_c(P_1)$ are identically true. All of these semantic definitions accord with the usual understanding of the meanings of these commands, and in each case V_c is complete and consistent if D is.

Using these semantic definitions, it is not hard to show that Figure 1 is a verifiable interpretation of its flowchart provided D contains a suitable set of axioms for the real numbers, summation, the integers, inequalities, and so forth. Thus, if the flowchart is entered with n a positive integer, the value of i on completion will be $n + 1$ (assuming that the program terminates) and the value of S will be $\sum_{j=1}^{n} a_j$. Presumably, the final value of i is of no interest, but the value of S is the desired result of the program, and the verification proves that the program does in fact compute the desired result if it terminates at all. Another section of this paper deals with proofs of termination.

Each of the given semantic definitions of the flowchart commands takes the form that $V_c(\mathbf{P}, \mathbf{Q})$ if and only if $(T_1(\mathbf{P}) \vdash Q_1) \wedge \cdots \wedge (T_l(\mathbf{P}) \vdash Q_l)$, where T_j is of the form $T_{j1}(P_1) \vee T_{j2}(P_2) \vee \cdots \vee T_{jk}(P_k)$. In particular there is the following:

(1) For an assignment operator $x \leftarrow f$

$$T_1(P_1) \quad \text{is} \quad (\exists x_0)(x = S^x_{x_0}(f) \wedge S^x_{x_0}(P_1)).$$

(2) For a branch command

$$T_1(P_1) \quad \text{is} \quad P_1 \wedge \Phi,$$
$$T_2(P_1) \quad \text{is} \quad P_1 \wedge \neg \Phi.$$

(3) For a join command

$$T_1(P_1, P_2) \quad \text{is} \quad P_1 \vee P_2; \quad \text{that is,}$$
$$T_{11}(P_1) \quad \text{is} \quad P_1, \qquad T_{12}(P_2) \quad \text{is} \quad P_2.$$

(4) For a start command, $T_1(\ )$ is **false**.
Thus, $V_c(Q_1)$ is identically true.

(5) For a halt command, the set of T_j's and Q_j's is empty.
Thus $V_c(P_1)$ is identically true.

For any set of semantic definitions such that

$$V_c(\mathbf{P}, \mathbf{Q}) \equiv (T_1(\mathbf{P}) \vdash Q_1) \wedge \cdots \wedge T_l(\mathbf{P}) \vdash Q_l),$$

in any verifiable interpretation, it is possible to substitute $T_j(\mathbf{P})$ for Q_j as a tag for any particular exit of a command without loss of verifiability. It is obvious that this substitution satisfies the semantic definition of the command whose exit is b_j; since $\vdash (T_j(\mathbf{P}) \supset Q_j)$, by Axiom 4 the substitution satisfies the semantic definition of the command whose entrance is b_j, and there are no other commands whose verification condition involves $I(b_j)$.

It is, therefore, possible to extend a partially specified interpretation to a complete interpretation, without loss of verifiability, provided that initially there is no closed loop in the flowchart all of whose edges are not tagged and that there is no entrance which is not tagged. This fact offers the possibility of automatic verification of programs, the programmer merely tagging entrances and one edge in each innermost loop; the verifying program would extend the interpretation and verify it, if possible, by mechanical theorem-proving techniques.

We shall refer to $\mathbf{T}_c(\mathbf{P})$ as the *strongest verifiable consequent* of the command c, given an antecedent $\mathbf{P}$. It seems likely that most semantic definitions in programming languages can be cast into the form $\mathbf{V}_c(\mathbf{P}, \mathbf{Q}) \equiv (\mathbf{T}_c(\mathbf{P}) \vdash \mathbf{Q})$, where $\mathbf{T}_c$ has several obvious properties:

(1) If $\mathbf{P} \supset \mathbf{P}_1$, $\mathbf{T}_c(\mathbf{P}) \supset \mathbf{T}_c(\mathbf{P}_1)$.

(2) If upon entry by entrance a_i with initial values $\mathbf{V}$, a command is executed and left by exit b_j with final values $\mathbf{W}$, then $\mathbf{T}_c(\mathbf{P}) \equiv \mathbf{Q}$, where P_α is defined as **false** for $\alpha \neq i$, $\mathbf{X} = \mathbf{V}$ for $\alpha = i$, and Q_β is defined as **false** for $\beta \neq j$, $\mathbf{X} = \mathbf{W}$ if $\beta = j$.

(3) If $\mathbf{P} = \mathbf{P}_1 \wedge \mathbf{P}_2$, $\mathbf{T}_c(\mathbf{P}) \equiv \mathbf{T}_c(\mathbf{P}_1) \wedge \mathbf{T}_c(\mathbf{P}_2)$.
If $\mathbf{P} = \mathbf{P}_1 \vee \mathbf{P}_2$, $\mathbf{T}_c(\mathbf{P}) \equiv \mathbf{T}_c(\mathbf{P}_1) \vee \mathbf{T}_c(\mathbf{P}_2)$.
If $\mathbf{P} = (\exists x)(\mathbf{P}_1)$, $\mathbf{T}_c(\mathbf{P}) \equiv (\exists x)(\mathbf{T}_c(\mathbf{P}_1))$.

That is, the transformation $\mathbf{T}_c$ distributes over conjunction, disjunction, and existential quantification. A semantic definition having these properties satisfies Axioms 1-4.

An ALGOL subset. To apply the same notions to a conventional programming language on the order of ALGOL, one might adopt a formal syntax for the language, such as the existing syntactic definition of ALGOL; designate certain phrase types as semantic units, such as the statements in ALGOL; and provide semantic definitions for these semantic units. Let us say that each statement Σ in an ALGOLic language is tagged with an antecedent and a consequent proposition (P_Σ and Q_Σ respectively), said to hold whenever control enters and leaves the statement in the normal sequential mode of control.

Now we may readily set up a verification condition for each common statement type.

(1) If Σ is an assignment statement, $x := f$, then

$$V_\Sigma(P_\Sigma; Q_\Sigma) \quad \text{is} \quad (\exists x_0)(S^x_{x_0}(P_\Sigma) \wedge x = S^x_{x_0}(f)) \vdash Q_\Sigma.$$

This assumes for simplicity that f is a true function of its free variables and has no side effects.

(2) If Σ is a conditional statement of the form **if** Φ **then** Σ_1 **else** Σ_2,

$$V_\Sigma(P_\Sigma, Q_{\Sigma 1}, Q_{\Sigma 2}; P_{\Sigma 1}, P_{\Sigma 2}, Q_\Sigma) \quad \text{is} \quad (P_\Sigma \wedge \Phi \vdash P_{\Sigma 1})$$
$$\wedge (P_\Sigma \wedge \neg \Phi \vdash P_{\Sigma 2}) \wedge (Q_{\Sigma 1} \vee Q_{\Sigma 2} \vdash Q_\Sigma).$$

Observe that here the exits of Σ_1 and Σ_2 become entrances to Σ, and so on.

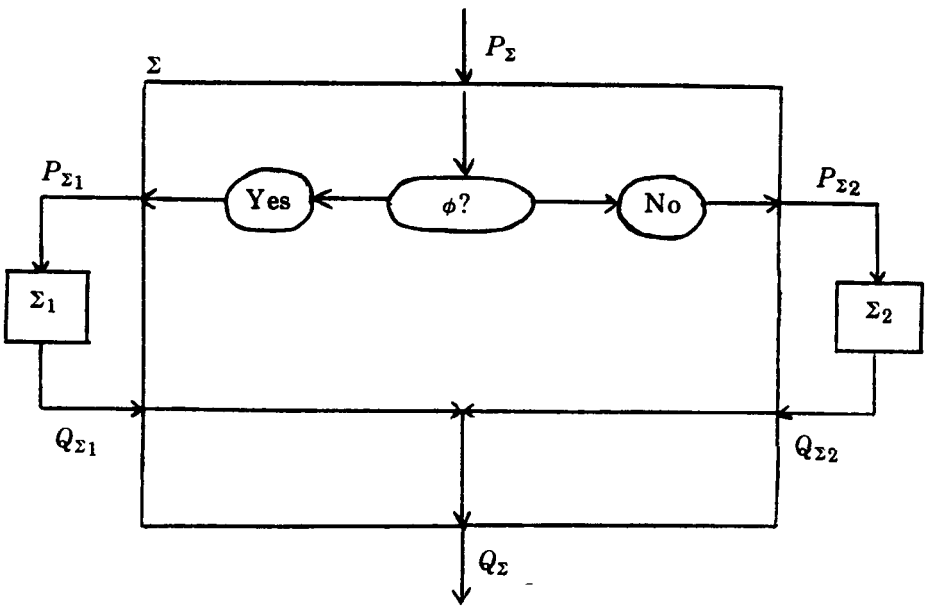

FIGURE 3

Consideration of the equivalent flowchart (Figure 3) indicates why this is true.

(3) If Σ_1 is a go-to statement of the form **go to** l, then $V_\Sigma(P_\Sigma; Q_\Sigma)$ is the identically true condition (**false** $\vdash Q_\Sigma$), because the sequential exit is never taken.

(4) If Σ is a labeled statement of the form $l: \Sigma_1$), then

$$V_\Sigma(P_\Sigma, P_l, Q_{\Sigma_1}; Q_\Sigma, P_{\Sigma_1})$$

is $(P_\Sigma \vee P_l \vdash P_{\Sigma_1}) \wedge (Q_{\Sigma_1} \vdash Q_\Sigma)$, where P_l is the disjunction of the antecedents of all statements of the form **go to** l.

(5) If Σ is a for-statement of the form **for** $x := a$ **step** b **until** c **do** Σ_1, where x is a variable and a, b, c are expressions, the verification rule is most easily seen by constructing the equivalent flowchart (Figure 4).

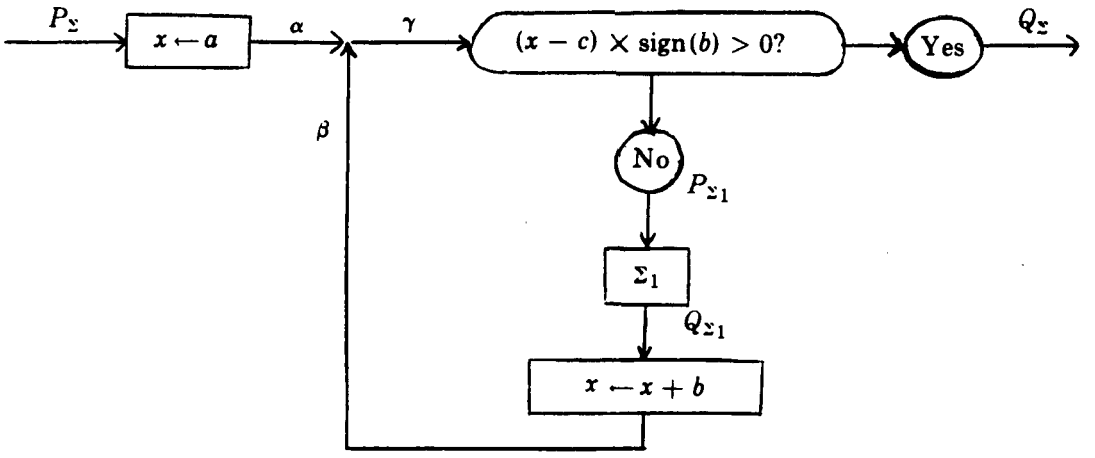

FIGURE 4

The strongest verifiable proposition P_α on edge α is

$$(\exists x_0)(S^x_{x_0}(P_\Sigma) \wedge x = S^x_{x_0}(a)).$$

The strongest verifiable proposition P_β on edge β is

$$(\exists x_0)(S^x_{x_0}(Q_{\Sigma_1}) \wedge x = x_0 + (S^x_{x_0}(b)),$$

which, if b contains no free occurrences of x, can be simplified to $S^x_{x-b}(Q_{\Sigma_1})$. The strongest verifiable proposition P_γ on edge γ is $P_\alpha \vee P_\beta$. Now the condition of verification is

$$(P_\gamma \wedge (x - c) \times \text{sign}(b) > 0 \vdash Q_\Sigma) \wedge (P_\gamma \wedge (x - c) \times \text{sign}(b) \leqq 0 \vdash P_{\Sigma_1}).$$

More precisely, since the definition of ALGOL 60 states that x is undefined after exhaustion of the for statement, the first half of the verification condition should be $(\exists x)(P_\gamma \wedge (x - c) \times \text{sign}(b) > 0) \vdash Q_\Sigma$. In typical cases, these conditions may be greatly simplified, since normally a, b, and c do not contain x, Σ_1 does not alter x, sign (b) is a constant, etc.

(6) Compound statements. A compound statement Σ is of the form **begin** Σ_1 **end**, where Σ_1 is a statement list. Then $V_\Sigma(P_\Sigma, Q_{\Sigma 1}; P_{\Sigma 1}, Q_\Sigma) \equiv (P_\Sigma \vdash P_{\Sigma 1}) \wedge (Q_{\Sigma 1} \vdash Q_\Sigma)$. Alternatively, one might identify P_Σ with $P_{\Sigma 1}$ and Q_Σ with $Q_{\Sigma 1}$. A statement list Σ is either a statement, or is of the form $\Sigma_1; \Sigma_2$ where Σ_1 is a statement list and Σ_2 is a statement. In the latter case,

$$V_\Sigma(P_\Sigma, Q_{\Sigma 1}, Q_{\Sigma 2}; P_{\Sigma 1}, P_{\Sigma 2}, Q_\Sigma) \quad \text{is} \quad (P_\Sigma \vdash P_{\Sigma 1}) \wedge (Q_{\Sigma 1} \vdash P_{\Sigma 2}) \wedge (Q_{\Sigma 2} \vdash Q_\Sigma).$$

Alternately, identify P_Σ with $P_{\Sigma 1}$, $Q_{\Sigma 1}$ with $P_{\Sigma 2}$, and $Q_{\Sigma 2}$ with Q_Σ.

(7) A null statement Σ is represented by the empty string. $V_\Sigma(P_\Sigma; Q_\Sigma)$ is $P_\Sigma \vdash Q_\Sigma$. Verification conditions are very similar to relations of deducibility, and in this case the verification condition reduces to precisely a relation of deducibility. One might say facetiously that the subject matter of formal logic is the study of the verifiable interpretations of the program consisting of the null statement.

Blocks (compound statements with bound local variables) cause some difficulties. If we treat occurrences of the same identifier within the scopes of distinct declarations as distinct, essentially renaming identifiers so that all variables have distinct names, the only effect of blocks is to cause their local variables to become undefined on exit. The effect of the undefining of a variable can be achieved by existential quantification of that variable. For instance, if a statement could have the form "**undefine** x," and the antecedent were $w < x \wedge x < y$, the strongest verifiable consequent would be $(\exists x)(w < x \wedge x < y)$, which is simplified to $w < y$.

One may then treat a block Σ as being of the form **begin** $\Sigma_1; \Sigma_2$ **end** where Σ_1 is a declaration list, where

$$V_\Sigma(P_\Sigma, Q_{\Sigma 1}, Q_{\Sigma 2}; P_{\Sigma 1}, P_{\Sigma 2}, Q_\Sigma) \quad \text{is} \quad (P_\Sigma \vdash P_{\Sigma 2}) \wedge (Q_{\Sigma 2} \vdash P_{\Sigma 1}) \wedge (Q_{\Sigma 1} \vdash Q_\Sigma).$$

A declaration Σ of the form (say) **real** x is treated as undefining x when executed at the end of the execution of a block. Thus $V_\Sigma(P_\Sigma; Q_\Sigma)$ is $(\exists x)\, P_\Sigma \vdash Q_\Sigma$.

A declaration list Σ is either a declaration, or is of the form $\Sigma_1; \Sigma_2$ where Σ_1 is a declaration list and Σ_2 is a declaration;

$$V_\Sigma(P_\Sigma, Q_{\Sigma 1}, Q_{\Sigma 2}; P_{\Sigma 1}, P_{\Sigma 2}, Q_\Sigma) \quad \text{is} \quad (P_\Sigma \vdash P_{\Sigma 1}) \wedge (Q_{\Sigma 1} \vdash P_{\Sigma 2}) \wedge (Q_{\Sigma 2} \vdash Q_\Sigma).$$

The above is a poor approximation to the actual complexities of ALGOL block structure; for example, it does not reflect the fact that transfers out of a block by go-to statements cause local variables of the block to become undefined. It may serve, however, to indicate how a complete treatment could be carried out. Note that it does not say that local variables lose their values upon leaving a block, but that preservation of their values may not be assumed in proofs of programs.

The ALGOL procedure statement offers even more complexities, with its several types of parameters, the dynamic-**own** feature, the possibility of recursive call, side effects, etc. We will not consider procedure statements in detail, but will illustrate the treatment of side effects by analyzing extended assignment statements allowing embedded assignments as subexpressions. For example, consider the statement $a := c + (c := c + 1) + c$, which has the effect of assigning $3c_0 + 2$ to a, where c_0 is the initial value of c, and assigning $c_0 + 1$ to c. Such a treatment requires saving the value of the leftmost c before executing the embedded assignment. Let us reluctantly postulate a processor, with a pushdown accumulator stack S. Introducing appropriate stacking and unstacking operators, we say that S_h (the *head* of S) is the contents of the top cell of S; that S_t (the *tail* of S) is the remainder of S, the value S would have if the stack were popped; and $x:S$ is the value S would have if x were stacked on S. These three operators are recognizable as the CAR, CDR, and CONS operators of LISP. The axioms governing them are $(x:S)_h = x$ and $(x:S)_t = S$. Now we may say that if an assignment statement has the form $x := f$, the processor should perform **begin** STACK (f); UNSTACK (x) **end**. If f is of the form $g + h$, STACK (f) is **begin** STACK (g); STACK (h); ADD **end**, where ADD pops the two top stack cells, adds their contents, and stacks the result; ADD is $S := (S_h + (S_t)_h) : ((S_t)_t)$. If x is a variable, STACK (x) is $S := x:S$. If f is of the form $x := g$, STACK (f) is **begin** STACK (g); STORE (x) **end**, where STORE (x) is $x := S_h$. UNSTACK (x) is **begin** $x := S_h$; $S := S_t$ **end**.

On this basis, any assignment statement is equivalent to a sequence of simple assignments without side effects; for instance,

$$a := c + (c := c + 1) + c$$

is equivalent to

begin $S := c:S$; $S := c:S$; $S := 1:S$; $S := ((S_t)_h + S_h) : ((S_t)_t)$;
$c := S_h$; $S := ((S_t)_h + S_h) : ((S_t)_t)$; $S := c:S$; $S := ((S_t)_h + S_h) : ((S_t)_t)$;
$a := S_h$; $S := S_t$ **end**.

If the antecedent of the original statement is $P(a,c,S)$, the strongest verifiable consequents of the successive statements in the equivalent compound statement are:

(1) $(S := c:S) : (\exists S')(S = c:S' \wedge P(a,c,S'))$.

(2) $(S := c:S) : (\exists S'')(\exists S')(S = c:S'' \wedge S'' = c:S' \wedge P(a,c,S'))$, or
$(\exists S')(S = c:(c:S') \wedge P(a,c,S'))$.

(3) $(S := 1:S) : (\exists S')(S = 1:(c:(c:S')) \wedge P(a,c,S'))$.

(4) $(S := ((S_t)_h + S_h) : ((S_t)_t)) : (\exists S'')(\exists S')(S = ((S''_t)_h + S''_h) : ((S''_t)_t) \wedge S'' = 1 : (c : (c : S')) \wedge P(a, c, S'))$

which simplifies, by application of the equation $S'' = 1 : (c : (c : S'))$, to $(\exists S')(S = (c + 1) : (c : S') \wedge P(a, c, S'))$.

(5) $(c := S_h) : (\exists c')(\exists S')(c = S_h \wedge S = (c' + 1) : (c' : S') \wedge P(a, c', S'))$.

Noting that $S_h = c' + 1$, or $c' = S_h - 1 = c - 1$, this becomes $(\exists S')(S = c : (c - 1 : S') \wedge P(a, c - 1, S'))$.

(6) $(S := ((S_t)_h + S_h) : ((S_t)_t)) : (\exists S')(S = 2c - 1 : S' \wedge P(a, c - 1, S'))$.

(7) $(S := c : S) : (\exists S')(S = c : (2c - 1 : S') \wedge P(a, c - 1, S'))$.

(8) $(S := ((S_t)_h + S_h) : ((S_t)_t)) : (\exists S')(S = 3c - 1 : S' \wedge P(a, c - 1, S'))$.

(9) $(a := S_h) : (\exists a')(\exists S')(a = S_h \wedge S = 3c - 1 : S' \wedge P(a', c - 1, S'))$, or
$(\exists a')(\exists S')(a = 3c - 1 \wedge S = 3c - 1 : S' \wedge P(a', c - 1, S'))$

(10) $(S \leftarrow S_t) : (\exists S'')(\exists a')(\exists S')(S = S''_t \wedge a = 3c - 1 \wedge S'' = 3c - 1 : S' \wedge P(a', c - 1, S'))$, or
$(\exists a')(\exists S')(S = S' \wedge a = 3c - 1 \wedge P(a', c - 1, S'))$, or
$(\exists a')(a = 3c - 1 \wedge P(a, c - 1, S))$.

For this statement, then, the condition of verification $V_\Sigma(P(a, c, S); Q)$ is $((\exists a')(a = 3c - 1 \wedge P(a', c - 1, S))) \vdash Q$, which is exactly the verification condition for either of

Begin $c := c + 1$; $\quad a := 3c - 1$ **end**

and

Begin $a := 3c + 2$; $\quad c := c + 1$ **end.**

Thus, the three statements are shown to be precisely equivalent, at least under the axioms (of exact arithmetic, etc.) used in the proof.

Proofs of termination. If a verified program is entered by a path whose tag is then true, then at every subsequent time that a path in the program is traversed, the corresponding proposition will be true, and in particular if the program ever halts, the proposition on the path leading to the selected exit will be true. Thus, we have a basis for proofs of relations between input and output in a program. The attentive reader, however, will have observed that we have not proved that an exit will ever be reached; the methods so far described offer no security against nonterminating loops. To some extent, this is intrinsic; such a program as, for example, a mechanical proof procedure, designed to recognize the elements of a recursively enumerable but not recursive set, cannot be guaranteed to terminate without a fundamental loss of power. Most correct programs, however, can be proved to terminate. The most general method appears to use the properties of well-ordered sets. A well-ordered set W is an ordered set in which each

nonempty subset has a *least* member; equivalently, in which there are no infinite decreasing sequences.

Suppose, for example, that an interpretation of a flowchart is supplemented by associating with each edge in the flowchart an expression for a function, which we shall call a W-function, of the free variables of the interpretation, taking its values in a well-ordered set W. If we can show that after each execution of a command the current value of the W-function associated with the exit is less than the prior value of the W-function associated with the entrance, the value of the function must steadily decrease. Because no infinite decreasing sequence is possible in a well-ordered set, the program must sooner or later terminate. Thus, we prove termination, a global property of a flowchart, by local arguments, just as we prove the correctness of an algorithm.

To set more precisely the standard for proofs of termination, let us introduce a new variable δ, not used otherwise in an interpreted program. Letting W designate the well-ordered set in which the W-functions are to be shown decreasing, and letting $\circledless$ be the ordering relation of W, it is necessary to prove for a command c whose entrance is tagged with proposition P and W-function ϕ, and whose exit is tagged with proposition Q and W-function ψ that

$$V_c(P \wedge \delta = \phi \wedge \phi \in W; Q \wedge \psi \circledless \delta \wedge \psi \in W).$$

Carrying out this proof for each command in the program, with obvious generalizations for commands having multiple entrances and exits, suffices not only to verify the interpretation, but also to show that the program must terminate, if entered with initial values satisfying the tag of the entrance.

The best-known well-ordered set is the set of positive integers, and the most obvious application of well-orderings to proofs of termination is to use as the W-function on each edge a formula for the number of program steps until termination, or some well-chosen upper bound on this number. Experience suggests, however, that it is sometimes much more convenient to use other well-orderings, and it may even be necessary in some cases. Frequently, an appropriate well-ordered set is the set of n-tuples of positive (or nonnegative) integers, for some fixed n, ordered by the assumption that $(i_1, i_2, \cdots, i_n) \circledless (j_1, j_2, \cdots, j_n)$ if, for some k, $i_1 = j_1$, $i_2 = j_2, \cdots, i_{k-1} = j_{k-1}$, $i_k < j_k$, $1 \leqq k \leqq n$. The flowchart of Figure 5 shows an interpretation using this well-ordering, for $n = 2$, to prove termination. It is assumed in the interpretation that the variables range over the integers; that is, the deductive system used in verifying the interpretation must include a set of axioms for the integers.

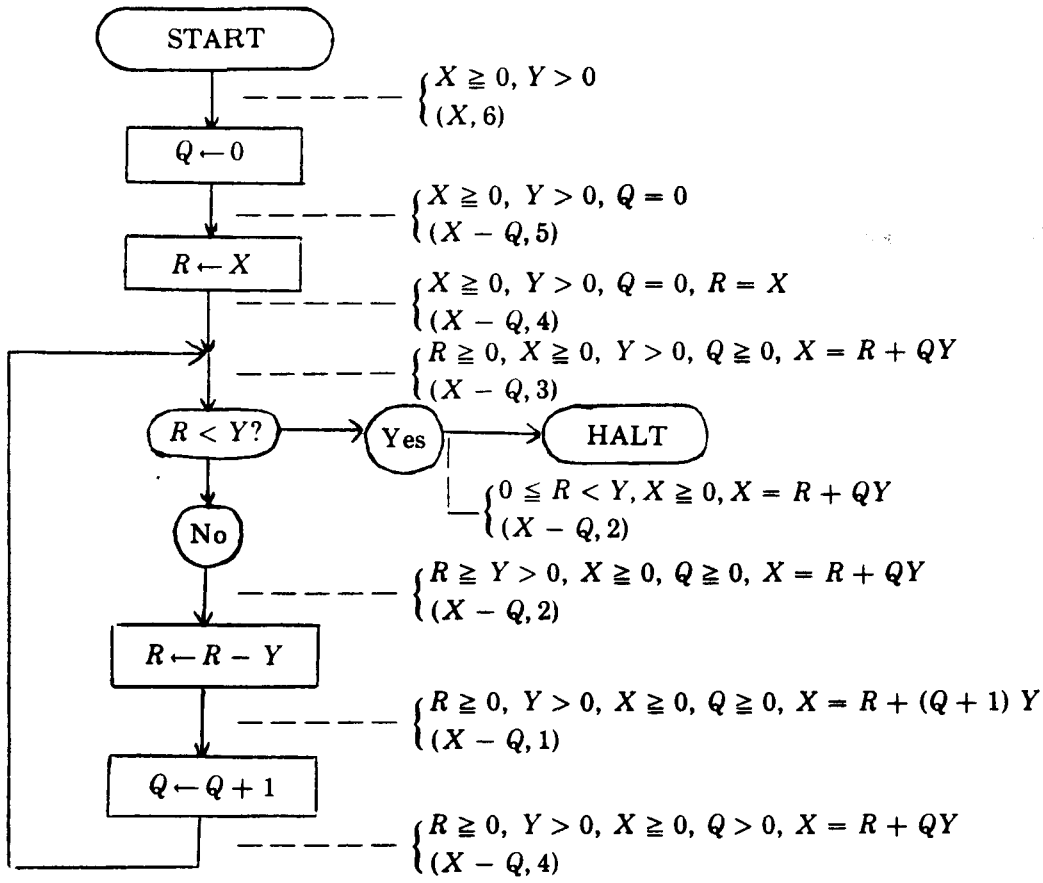

FIGURE 5. Algorithm to compute quotient Q and remainder R of $X \div Y$, for integers $X \geqq 0$, $Y > 0$

REFERENCES

1. J. McCarthy, "A basis for a mathematical theory of computation" in *Computer programming and formal systems*, North-Holland, Amsterdam, 1963, pp. 33-70.

2. _____, *Towards a mathematical science of computation*, Proc. IFIP Congr. 62, North-Holland, Amsterdam, 1962, pp. 21-28.

CARNEGIE INSTITUTE OF TECHNOLOGY
PITTSBURGH, PENNSYLVANIA