Generic commands—a tool for partial correctness formalisms

J. Schwarz

Department of Artificial Intelligence, University of Edinburgh, Hope Park Square, Meadow Lane, Edinburgh EH8 9NW

A semantic framework for logical formalisms based on partial correctness assertions of the form $P\{S\}Q$ is given. $P\{S\}Q$ asserts that if P holds before execution of S then Q holds after. Generic commands are introduced. The generic command $\{P => Q\}$ is in a sense the least specified command with precondition P and postcondition Q. Proof rules and a noneffective semantics are given for generic commands. A proof rule for recursive procedures is given using generic commands.

(Received October 1975)

1. Introduction

Hoare (1969) introduced the first of a family of proof systems for partial correctness which are based on the linear representations of programs. The basic notation of these systems is what I will call a partial correctness assertion (or assertion for short). It has the form $P\{S\}Q$ where P and Q are the precondition and postcondition respectively, and S is the command of the assertion. P and Q are expressions in some logical language which has usually been taken to be the predicate calculus, and S is a program or a program fragment in some programming language. Moreover, the programming language and the logical language must have some common structure, in particular they normally share the atomic expressions commonly called 'variables' in logic or 'identifiers' in programming languages. I will call these identifiers. I use this word to emphasise that whether these items should be considered as fixed or varying is a question of context.

One advantage of an approach using partial correctness assertions over other methods of relating logical conditions to programs (such as attaching them to nodes in a flowchart) is that the proof systems have a form similar to that of conventional logics. They consist of axioms, and rules of inference with a conventional notion of proof as a sequence of lines (or assertions) each of which follows from earlier ones via some inference rule.

The first step in the analysis of a logic in this form is often the development of a semantics for the lines of proofs (in this case the assertions). That is, a criteria for when these assertions can be said to hold. Based on such a semantics, we can ask questions such as 'Does every provable assertion always hold?' or 'Is every assertion which always holds provable?', etc.

A development along these lines is not difficult, and can be found for example in Igarashi, London and Luckham (1975), Schwarz (1975) or Oppen and Cook (1975).

Hoare (1971) introduced a set of inference rules for programs containing call statements. The rules for nonrecursive procedures fit into the logical framework, but recursive procedures require the introduction of a new element, namely hypothetical deduction. One of the hypotheses of the recursive invocation rule is that a certain assertion is provable from another assertion, which is written $A \vdash B$. There are other logics (e.g. Gentzen Sequent calculus) which permit lines of proofs to contain '$\vdash$', and it is often possible to give this symbol a semantic interpretation. However, in the case of recursive invocation as presented by Hoare, one is forced to read '$\vdash$' as 'provable' and give the syntactic definition indicated above. This means that the correctness of the rule must be shown by an induction on the structure of possible proofs of the hypothesis $A \vdash B$. (Such a proof can be found in Igarashi, London and Luckham (1975)).

This report presents a construction which leads to an inference rule for recursive procedures which eliminates the need for a hypothetical deduction as an antecedent. The technique essentially consists of introducing the 'generic' command which can be used as a place holder for some other command when we do not care about the detailed workings of the latter command. This is a formal device (similar to the possible introduction of variables standing for commands) for introducing hypotheses. It does not greatly affect the effort needed to prove particular programs, but does simplify the study of the proof systems. It should be understood that it is not the elimination of '$\vdash$' which creates this simplification, but the provision of a semantic interpretation for every line in a proof.

In addition to their use in the rules for recursive procedures, generic program statements are interesting in their own right because they demonstrate a technique for using proof rules to determine the semantics of a programming language. That is, given a programming language and a proof system for it, one can determine a semantics for that language which is 'defined' by the proof system. This should agree with the (formal or informal) semantics which was in mind when the proof system was designed. This process is dealt with in more detail in Schwarz (1975).

In this report I will assume that the reader is familiar with partial correctness assertions for imperative programming language features, such as assignment, conditional expressions, etc. and with the following proof rules. I consider an axiom to be equivalent to a proof rule with zero antecedents.

Assignment: $P\langle t/X\rangle\{X := t\}P$

(where $P\langle t/X\rangle$ is the result of substituting t for free occurrences of X in P)

Consequence: $\dfrac{P\{S\}Q}{P'\{S\}Q}$ (providing $P' \supset P$ is a theorem)

$\dfrac{P\{S\}Q}{P\{S\}Q'}$ (providing $Q \supset Q'$ is a theorem)

Conjunction: $\dfrac{P\{S\}Q\ P'\{S\}Q'}{P \wedge P'\{S\}Q \wedge Q'}$

Disjunction: $\dfrac{P\{S\}Q\ P'\{S\}Q'}{P \vee P'\{S\}Q \vee Q'}$

2. Generic commands

To avoid confusion among different kinds of syntactic objects, I adopt the following convention. A 'condition' is a formula of first order predicate calculus with function symbols and equality; a 'command' is a program or program fragment in some unspecified programming language; an 'assertion' is a partial correctness assertion with a precondition, a postcondition and a command. P and Q will denote conditions; S a command; and $P\{S\}Q$ the assertion with precondition P, postcondition Q, and command S.

In reasoning about programs it is often convenient to make some assumptions about the behaviour of a part of a program without specifying either its complete behaviour or giving a description of its inner workings. By postponing the specification of details this can be a great aid in the design of algorithms. One technique for achieving this is the introduction of nondeterministic features into a programming language, for example the guarded commands of Dijkstra (1975). Generic commands can be seen as a direct embodiment of this idea. If we need a command which satisfies a given pre and postcondition we simply construct it from those two conditions. That is, we form a syntactic object from those two conditions and call it a command. The semantics of this command will be nondeterministic. Any effect which conforms to the specified pre and postconditions will be permissible.

For example, suppose we need a command which computes the square root of an integer. More specifically, if X is a positive integer before the command is executed then after execution Y must be its square root. The corresponding generic command would be written

$$\{X > 0 => Y^2 \leq X < (Y+1)^2\} .$$

This is called a generic command because it represents the class of all possible commands S for which

$$X > 0\{S\}Y^2 \leq X < (Y+1)^2$$

holds.

Actually this includes too many commands, for example $\{X := 0; Y := 0\}$. A further fact is needed about the command S, namely that it does not modify X. Therefore in addition to the pre and postconditions, a generic command must indicate certain identifiers which are involved actively in the computation. For the general form of a generic command we have

$$\{P => Q[A]\}$$

where A is a list of the active identifiers, P is the precondition and Q is the postcondition. All identifiers other than A are called inactive identifiers of S.

These identifiers will be inactive not only in the sense that their value is not modified but in the stronger sense that their value is not examined in the course of the computation.

The square root example might now be written as

$$S \equiv \{X = x \wedge X > 0 => Y^2 \leq x < (Y+1)^2[X, Y]\}$$

What does this mean? There are three cases. 1. The value of X is the same as the value of x and greater than 0. Execution of S must assign to Y the square root of this value. It may meanwhile arbitrarily (i.e. nondeterministically) assign a value to X. All other identifiers must remain untouched. 2. The value of X is not greater than 0. The precondition fails and S may arbitrarily assign values to X and Y while leaving other identifiers untouched. 3. The value of X is greater than 0 but different from the value of x. In this case since x is inactive S cannot detect this inequality and must behave as though they were in fact equal, i.e. as case 1.

As a further example consider a generic command which exchanges X and Y

$$\{X = x \wedge Y = y => X = y \wedge Y = x[X, Y]\} .$$

In fact a very useful form of the generic command is

$$\{X_1 = x_1 \wedge X_2 = x_2 \ldots => Q[X_1, X_2 \ldots]\}$$

because Q can relate the values of the active identifiers after execution to their values before execution (as given by the inactive variables).

If S is $\{P => Q[A]\}$, the following new inference rules are relevant. If A is a set of identifiers I will write $\tilde{A}$ for a list of the identifiers not in A.

Generic Axiom: $P\{S\}Q$

Stability Axiom: $P'\{S\}P'$ (providing all free identifiers of P' are in $\tilde{A}$)

Existential Generalisation: $\dfrac{P'\{S\}Q'}{(\exists x)P'\{S\}(\exists x)Q'}$

(providing $x \in \tilde{A}$) .

Universal Generalisation: $\dfrac{P'\{S\}Q'}{(\forall x)P'\{S\}(\forall x)Q'}$

(providing $x \in \tilde{A}$) .

A precise definition of the semantics of generic commands will be given below. For now the reader can note that the generic axiom is the essential characterisation of the command; the stability axiom is a consequence of the fact that S does not modify the values of inactive identifiers; and the two generalisation rules are consequences of the aloofness imposed on inactive identifiers.

Application of the rules of stability and generalisation need not be restricted to generic commands. Indeed if S is a more conventional command (such as an assignment) they hold if we take as the active identifiers the set of identifiers occuring in S. In general associated with a statement S there is a set of identifiers V for which stability and generalisation are given or can be derived. I will say the identifiers in V are active in S, and that all others are inactive in S.

The point of the generic and stability axioms is clear. To see the need for the generalisation rules consider the square root example,

$$S \equiv \{x = X => Y^2 \leq x < (Y+1)^2 \wedge x = X[X, Y]\} .$$

We want to prove *true* $\{S\}Y^2 \leq X < (Y+1)^2$. The generic axiom gives $x = X\{S\}Y^2 \leq x < (Y+1)^2 \wedge x = X$, and an application of existential generalisation gives

$$(\exists x)(x = X)\{S\}(\exists x)(Y^2 \leq x < (Y+1)^2 \wedge x = X) .$$

The desired conclusion will now follow by applications of the consequence rule of inference, since the pre and postconditions are logically equivalent to the desired ones.

The pattern of the above proof is quite common in working with generic commands. First remove inactive identifiers from the postcondition using consequence, and then from the precondition using existential generalisation and consequence.

It is possible to extend the results of this paper to generic commands with multiple condition pairs. The form of the command is then $\{P_1 => Q_1; \ldots; P_n => Q_n[V]\}$, and the generic axiom becomes:

$$P_i\{P_1 => Q_1; \ldots; P_n => Q_n[V]\}Q_i \text{ (for } 1 \leq i \leq n).$$

This extension is carried out in Schwarz (1975).

3. Recursive procedures

The previous section introduced generic commands and stated some inference rules for them. In this section I will present a rule of inference for recursive procedures which utilises generic commands. Without giving the details I assume that there is a nonrecursive call command of the form $\{call\ f(A)\}$ where A is a list of actual arguments and f is a procedure. f can either be a procedure identifier declared separately or (a theoretically more tractable alternative) a lambda expression. Further, I assume there are rules of inference which handle the argument passing and other aspects of these calls, and will give inference rules for recursive procedures which can be used to reduce reasoning about recursive procedures to applications of rules for nonrecursive procedures.

Suppose we have the recursive declaration, $F \equiv [f(X) = S]$; where f is a procedure identifier which can occur inside S and X is a list of identifiers (the formal arguments). Consider a call command $\{call\ F(A)\}$. To make the meaning of a call command selfcontained and not dependent on external declarations I am assuming that the actual declaration occurs in the command, but of course various abbreviations are conceivable.

Hoare's rule for recursive invocation is:

$$\frac{F \quad P\{call\ f(X)\}Q \vdash P\{S\}Q}{P\{call\ f(X)\}Q} .$$

Note that the validity of this rule depends on the identity of the actual and formal arguments. This rule represents an induction on the depth of nesting of recursive calls, or alternatively on the length of the computation. In this form it is difficult to give a meaning to the antecedents of this rule. In particular, F is not an assertion. If F is not made an antecedent but attached by a proviso it is not clear how to handle the case of $P\{call\ F(X)\}Q$ failing to hold. A slightly modified version of the rule would be

$$\frac{P\{call\ f(X)\}Q \vdash P\{S\}Q}{P\{call\ F(X)\}Q} .$$

It is now possible to give a meaning to the antecedent of this rule which involves variation of 'f' over all possible procedures (as is done in Donahue (1975)), but I will not pursue that approach. Instead I present an alternative version of recursive invocation which uses generic commands.

As above let $F \equiv [f(X) = S]$ and also let

$$F^* \equiv [\lambda(X)(P = > Q[V])] .$$

And assume that V includes all active identifiers of S. My version of recursive invocation is

$$\frac{P_1\{call\ F^*(A)\}Q_1 \quad P\{S\langle F^*/f\rangle\}Q}{P_1\{call\ F(A)\}Q_1}$$

where $S\langle F^*/f\rangle$ is the result of replacing free occurrences of the procedure identifier f (i.e. not bound by declarations of recursive functions) by F^* in S, and P_1, Q_1 are any conditions. The rule is another form of induction on the depth of nesting of recursive calls. The actual manipulations in the proof of $P\{S\langle F^*/f\rangle\}Q$ will be essentially the same as those in the hypothetical deduction of $P\{S\}Q$ from $P\{call\ f(X)\}Q$. Some steps may be simpler in my version because P and Q are directly pre and postconditions of the body of F^* and not of a call command. This could simplify arguments relating to parameter passing. My treatment also eliminates the need for Hoare's rule of adaptation which was required to take into account the fact that certain identifiers do not change values during the calls to f which occur inside S. Because F^* will actually be used in $S\langle F^*/f\rangle$ the role of adaptation can be played by the rules (such as stability) which are applicable to nonrecursive procedures.

As an example of the recursion rule, consider factorial

Let $S \equiv \{$**if** $X = 0$ **then** $Y := 1$ **else** $call\ fact(X - 1, Y)$;
$Y := X \times Y$ **fi**$\}$
$F \equiv [fact(X) = S]$
$G \equiv \{X = x = > Y = x! \wedge X = x[X, Y]\}$
$F^* \equiv [\lambda(X, Y)G]$.

(Note that the form of G names X as an active identifier in addition to Y because although during execution of S, X's value does not change, execution of S does require examining X).

A relevant instance of recursive invocation is

$$\frac{true\{call\ F^*(A, Y)\}\,Y = A! \quad X = x\ \{\textbf{if } X = 0 \textbf{ then } Y := 1 \textbf{ else } call\ F^*(X - 1, Y); Y := X \times Y \textbf{ fi}\}\,Y = x! \wedge X = x}{true\{call\ F(A, Y)\}\,Y = A!}$$

4. Semantic framework

This section presents a version of relational semantics as a general mechanism for dealing with partial correctness assertions. (The reader is referred to Hitchcock and Park (1973) for a more detailed treatment of similar semantics). I assume that the vocabulary (i.e. identifier, function, predicate and constant symbols) but not its interpretation has been selected and fixed for the remainder of this paper. A structure M consists of a set $|M|$ and interpretations for function, predicate and constant symbols as functions from $|M|$ to $|M|$, relations over $|M|$ and elements of $|M|$ respectively. An environment e (for M) is a list of elements of $|M|$ corresponding in some fixed order to the identifiers. For a condition P (i.e. a formula of predicate calculus) there is a standard notion of P holding in M for an environment e. I will write $[\![P]\!]_M$ for the set of environments in which P holds. For M the meaning of a command S will be a relation $[\![S]\!]_M$ between environments (i.e. a set of pairs of environments). In general if R is a relation I will write eRe' for $\langle e, e'\rangle \in R$. The intuitive content of $[\![S]\!]_M$ is that $e\,[\![S]\!]_M\,e'$ precisely when e' is a possible result of having executed S starting in the environment e. If for some e there is no e' such that $e\,[\![S]\!]_M\,e'$ then S does not terminate; if there is a unique e' then S is determinate; in the general case where there is more than one e' we are dealing with nondeterminism.

It will be convenient to have some notation for dealing with environments. If V is a list of variables I will write $\tilde{V}$ for the list variables not in V and $e|V$ (e restricted to V) for the sublist of e which corresponds to the members of V. If in addition m is a list of elements of $|M|$ I will write $e\langle m/V\rangle$ for the unique environment e' such that $e'|V = m$ and $e'|\tilde{V} = e|\tilde{V}$. I define the relation $=_V$ to be $\{\langle e, e'\rangle : e|V = e'|V\}$. It is convenient to assume that the active identifiers of all commands are selected from a fixed set A, and the inactive ones from $\tilde{A}$. Then another convenient notation is that $e \cdot e'$ denotes the unique d such that $d =_A e'$ and $d =_{\tilde{A}} e'$. (A useful mnemonic is that 'active' lexicographically precedes 'inactive'). Note that

$$(e \cdot e') \cdot e^* = e \cdot e^* = e \cdot (e' \cdot e^*) .$$

If P and Q are conditions let

$$[\![P \rightarrow Q]\!]_M = \{\langle e, e'\rangle : e \notin [\![P]\!]_M \text{ or } e' \in [\![Q]\!]_M\} .$$

A partial correctness statement $P\{S\}Q$ holds in M if $[\![S]\!]_M \subset [\![P \rightarrow Q]\!]_M$. An equivalent characterisation of $P\{S\}Q$ holding is that if $e\,[\![S]\!]_M\,e'$ and $e \in [\![P]\!]_M$ then $e' \notin [\![Q]\!]_M$.

An important property of the meanings of commands is the fashion in which inactive identifiers are handled. I define two properties of relations between environments which will hold for all meanings. Let R be a relation

stability: if eRe' then $e =_{\tilde{A}} e'$.
aloofness: if eRe' then for all e^*, $(e \cdot e^*)\ R(e' \cdot e^*)$.

Informally stability is the requirement that 'inactive' identifiers are not modified by commands and aloofness is the requirement that inactive identifiers are not examined.

At this point I assume an arbitrary structure has been chosen, and that all definitions and proofs are assumed to refer to this structure. Therefore the structure will in general be omitted from the above notations.

5. Semantics of generic commands

Within the framework of the previous section I will give a definition of the meaning of generic commands. Since there is a fixed set A of active identifiers I will drop the indication of active identifiers from generic commands. Let G be the generic command $\{P = > Q\}$. One might think that $[\![P \rightarrow Q]\!]$ could serve as the meaning of G. However it does not in general satisfy either stability or aloofness. Consider for example $\{x = X = > x = X\}$ where $X \in A$ and $x \in \tilde{A}$. Suppose e is an environment such that $e \notin [\![x = X]\!]$. Then $e\,[\![x = X \rightarrow x = X]\!]\,e'$ for any e', and clearly this violates stability. Let R be $[\![x = X \rightarrow x = X]\!] \cap =_{\tilde{A}}$. Then R satisfies stability, but it still does not satisfy aloofness. To see this, suppose $e \notin [\![x = X]\!]$ then $eR(e\langle m/X\rangle)$ for any m. Let $m^* = e|X$, and choose $m \neq m^*$. If R satisfied aloofness we would have $(e\langle m^*/x\rangle)\ R(e\langle m/X\rangle\langle m^*/x\rangle)$. But $e\langle m^*/x\rangle \in [\![x = X]\!]$ and $(e\langle m/X\rangle\langle m^*/X\rangle) \notin [\![x = X]\!]$ which contradicts

$$R \subset [\![x = X \rightarrow x = X]\!] .$$

A definition which does work is

$$[\![P = > Q]\!] = \bigcap_{e^*} \{\langle e, e' \cdot e\rangle : (e \cdot e^*)\, [\![P \rightarrow Q]\!] e' \text{ and } e' =_{\tilde{A}} e^*\}$$

To see how this works consider $\{x = X = > x = X\}$ where $X \in A$ and $x \in \tilde{A}$. I will show that $[\![x = X = > x = X]\!] = \{\langle e, d\rangle : e|X = d|X \text{ and } e =_{\tilde{A}} d\}$. Suppose $e|X = d|X$ and $e =_{\tilde{A}} d$. For any e^* let $e' = d \cdot e^*$ then $(e \cdot e^*)[\![x = X \rightarrow x = X]\!] e'$, $e' =_{\tilde{A}} e^*$ and $e' \cdot e = (d \cdot e^*) \cdot e = d \cdot e = d$. Thus $e[\![x = X = > x = X]\!] d$. On the other hand suppose $e[\![X = x = > X = x]\!] d$. That $e =_{\tilde{A}} d$ is immediate. To see that $e|X = d|X$ let $m = e|X$, and $e^* = e\langle m/x\rangle$. There must be an e' such that $e \cdot e^* [\![P \rightarrow Q]\!] e'$, $e'|x = e^*|x$ and $e' \cdot e = d$. Since $e \cdot e^* \in [\![X = x]\!]$ this implies $e' \in [\![X = x]\!]$ and so $d|X = e'|X = e'|x = e^*|x = m = e|X$.

I now show that the rules of inference of Section 2 are valid for generic commands.

Theorem 1:
Any instance of the generic rule holds.

Proof:
An instance of the rule is $P\{P = > Q\}Q$.
Letting $e^* = e$ in the definition gives
$[\![P \rightarrow Q]\!] \subset \{\langle e, e' \cdot e\rangle : (e \cdot e)[\![P \rightarrow Q]\!] e' \text{ and } e'|A = (e \cdot e)|A\}$
Since $e \cdot e = e$ this implies $[\![P = > Q]\!] \subset [\![P \rightarrow Q]\!]$ which is the desired conclusion.

Theorem 2:
The meaning of a generic command satisfies stability and aloofness.

Proof:
Stability is immediate. To see that it satisfies aloofness replace e in the definition by $e \cdot d$ (which has the same range of variation). Then

$$\begin{aligned}[\![P = > Q]\!] &= \bigcap_{e^*} \{\langle e \cdot d, e' \cdot (e \cdot d) : (e \cdot d) \cdot e^* [\![P \rightarrow Q]\!] e' \\ &\qquad \text{and } e' =_{\tilde{A}} e^*\} \\ &= \bigcap_{e^*} \{\langle e \cdot d, e' \cdot d\rangle : e \cdot e^* [\![P - > Q]\!] e' \text{ and } e' =_{\tilde{A}} e^*\}\end{aligned}$$

Thus $e[\![P = > Q]\!] e' \cdot e$ implies $e \cdot d [\![P = > Q]\!] e' \cdot d$.

Theorem 3:
If S satisfies stability and aloofness then the rules of stability and generalisation are valid for S.

Proof:

(*a*) Stability. Recall that this rule allows inferring $P\{S\}P$ provided all free variables of P are in $\tilde{A}$. Then if $e[\![S]\!]e'$, e and e' agree on the free variables of P by the stability condition. So $e \in [\![P]\!]$ implies $e' \in [\![P]\!]$.

(*b*) Existential generalisation. This says that if $x \in \tilde{A}$ we can infer $(\exists x)P\{S\}(\exists x)Q$ from $P\{S\}Q$. Suppose $P\{S\}Q$ holds, $e[\![S]\!]e'$ and $e \in [\![(\exists x)P]\!]$. Then there is an m such that $e\langle m/x\rangle \in [\![P]\!]$ and by aloofness $e'\langle m/x\rangle \in [\![Q]\!]$. Hence $e' \in [\![(\exists x)Q]\!]$.

(*c*) Universal generalisation. This says that if $x \in \tilde{A}$ we can infer $(\forall x)P\{S\}(\forall x)Q$ from $P\{S\}Q$. Suppose $P\{S\}Q$ holds, $e[\![S]\!]e'$ and $e \in [\![(\forall x)P]\!]$. Then for all m, $e\langle m/x\rangle \in [\![P]\!]$ and by aloofness $e'\langle m/x\rangle \in [\![Q]\!]$. Thus $e' \in [\![(\forall x)Q]\!]$.

Corollary 4:
The rules of stability and generalisation are valid for generic commands.

In Schwarz (1975) I prove that if the rules of consequence, conjunction and disjunction are added to the generic, stability and generalisation rules then the system is complete in the sense that any assertion with a generic command which holds in all structures is provable. The essential step of that proof is a construction which gives for $\{P = > Q\}$, P' and Q', formula R such that in any structure M,

$$[\![R]\!]_M = [\![true]\!]_M \textit{ iff } [\![P = > Q]\!]_M \subset [\![P' \rightarrow Q']\!]_M .$$

A significant fact about generic command $\{P = > Q\}$ is that it is the largest relation compatible with the requirement that P and Q are good pre and postconditions and which also satisfies stability and aloofness. More formally we have

Theorem 5:
If S is a command satisfying stability and aloofness such that $P\{S\}Q$ holds, then $[\![S]\!] \subset [\![P = > Q]\!]$.

Proof:

$$\begin{aligned}[\![S]\!] &= \{\langle e, e'\rangle : e[\![S]\!]e'\} \\ &= \{\langle e, e' \cdot e\rangle : e[\![S]\!]e'\} && \text{stability} \\ &= \bigcap_{e^*} \{\langle e, e' \cdot e\rangle : e \cdot e^* [\![S]\!] e' \cdot e^*\} && \text{aloofness} \\ &= \bigcap_{e^*} \{\langle e, e' \cdot e\rangle : e \cdot e^* [\![S]\!] e'' \text{ and} \\ &\qquad e'' =_{\tilde{A}} e^*\} && \text{stability} \\ &\subset \bigcap_{e^*} \{\langle e, e' \cdot e\rangle : e \cdot e^* [\![P - > Q]\!] e'' \\ &\qquad \text{and } e'' =_{\tilde{A}} e^*\} && \text{hypothesis} \\ &= [\![P = > Q]\!]\end{aligned}$$

On the basis of Theorem 5 we may say that the meaning of generic commands is determined from the proof rules for it in the sense that it is the largest relation which is compatible with the assertions which can be derived from those rules and which satisfies stability and aloofness. In general given any proof system for partial correctness assertion we can find such maximal meanings for the commands of the system which are compatible with the proof rules and satisfies stability and aloofness. A general construction for this can be found in Schwarz (1975) where I also argue that a criteria for the adequacy of a proof system should be that the relational semantics given by this construction should agree with the 'intended' semantics.

6. Semantics of recursive procedures

There are several methods available for defining the meanings of calls to recursive procedure. I will adopt one which gives the meaning of recursive calls in terms of a sequence of nonrecursive 'expansions' of the given procedure. This has the advantage of not requiring a detailed description of the calling mechanism.

In the rest of this section let F be the recursive function declared by $[f(X) = S]$; let $S_0 \equiv \{true = > false\}$; let F_0 be the nonrecursive procedure $\lambda(X)S_0$; for $n \geq 0$ define S_{n+1} and F_{n+1} inductively by $S_{n+1} \equiv S\langle S_n/f\rangle$ and $F_{n+1} \equiv \lambda(X)S_{n+1}$. Then I define

$$[\![call\ F(A)]\!] = \bigcup_{n=0} [\![call\ F_n(A)]\!] .$$

Because S_n contains fewer occurrences of calls to recursive procedures than S does we may assume that each component of this union has been previously defined. In order to verify the validity of the recursive invocation rule it will be necessary to make certain assumptions about the nonrecursive calls in terms of which recursive calls are defined. The requirements would be met by any reasonable definition of calls to nonrecursive procedures.

Requirements:

1. All commands satisfy stability and aloofness.
2. If $G_1 \equiv \lambda(X)S^*$, $G_2 \equiv \lambda(X)S^{**}$ and $[\![S^*]\!] \subset [\![S^{**}]\!]$ then $[\![S\langle G_1/f\rangle]\!] \subset [\![S\langle G_2/f\rangle]\!]$.

Theorem 6:
If a relational semantics for a language with recursive and nonrecursive calls satisfies requirements 1. and 2. then recursive invocation is a valid inference rule.

Proof:
Let $G \equiv \{P => Q\}$ and $F^* \equiv \lambda(X)G$. Then an instance of the recursive invocation rule for F is

$$\frac{P_1\{call\ F^*(A)\}Q_1 \quad P\{S\langle F^*/f\rangle\}Q}{P_1\{call\ F(A)\}Q_1}$$

Suppose the antecedents hold. I will show that for all n $[\![S_n]\!] \subset [\![G]\!]$ and hence by 2. applied to $\{call\ f(A)\}$, $[\![call\ F_n(A)]\!] \subset [\![call\ F^*(A)]\!]$. By hypothesis $[\![call\ F^*(A)]\!] \subset [\![P_1 \rightarrow Q_1]\!]$. Thus taking a union over n gives $[\![call\ F(A)]\!] \subset [\![P_1 \rightarrow Q_1]\!]$.

The proof of $[\![S_n]\!] \subset [\![G]\!]$ is by induction on n. $[\![S_0]\!] = \emptyset$ and hence the base case is immediate. If $[\![S_n]\!] \subset [\![G]\!]$ then by 2. $[\![S_{n+1}]\!] \subset [\![S\langle F^*/f\rangle]\!] \subset [\![P \rightarrow Q]\!]$.

By Theorem 5 $[\![G]\!]$ is the largest relation smaller than $[\![P \rightarrow Q]\!]$ which also satisfies stability and aloofness. Since according to 1. $[\![S_{n+1}]\!]$ satisfies stability and aloofness $[\![S_{n+1}]\!] \subset [\![G]\!]$.

Acknowledgements
The results of this paper are taken from my dissertation (1975) for which John Reynolds was a helpful adviser. Work on that dissertation was partially supported by grants from the US National Science Foundation GJ-41540 and ARPA 30602-72-C-0003. Rod Burstall and David MacQueen read drafts and made suggestions on presentation. Eleanor Kerse typed those many drafts. The preparation of this paper was supported by the UK Science Research Council.

References

DIJKSTRA, E. W. (1975). Guarded commands, non-determinacy and formal derivation of programs, *CACM*, Vol. 18, No. 8, pp. 433-457.
DONAHUE, J. E. (1975). The mathematical semantics of axiomatically defined programming language constructs, *Proceedings of Symposium on Proving and Improving Programs*, Arc-et-Senans, France, pp. 353-370.
HITCHCOCK, P., and PARK, D. (1973). Induction rules and termination proofs, *Automata, Languages and Programming*, (ed. M. Nivat) pp. 225-251.
HOARE, C. A. R. (1969). An axiomatic basis for computer programming, *CACM*, Vol. 12, No. 10, pp. 576-580, 583.
HOARE, C. A. R. (1971). Procedures and parameters: an axiomatic approach, *Symposium on Semantics of Algorithmic Languages*, (ed. E. Engeler) Springer Verlag, pp. 102-106.
IGARASHI, S., LONDON, R., and LUCKHAM, D. (1975). Automatic program verification I: a logical basis and its implementation, *Acta Informatica*, Vol. 4, pp. 145-176.
OPPEN, D., and COOK, S. (1975). Proving assertions about programs that manipulate data structures, *Proceedings of 7th Annual ACM Symposium on Theory of Computing*.
SCHWARZ, J. S. (1975). Partial correctness formalisms, *Ph.D. dissertation*, Syracuse University, School of Systems and Information Science.