PROGRAMMING LANGUAGE CONSTRUCTS FOR WHICH IT IS IMPOSSIBLE TO OBTAIN GOOD HOARE-LIKE AXIOMS

Edmund Melson Clarke, Jr.

TR 76-287

August 1976

Department of Computer Science
Cornell University
Ithaca, New York 14853

PROGRAMMING LANGUAGE CONSTRUCTS FOR WHICH IT IS IMPOSSIBLE TO OBTAIN GOOD HOARE-LIKE AXIOMS

I. Introduction

Many different formalisms have been proposed for proving programs correct. Of these probably the most widely referenced is the axiomatic approach of C. A. R. Hoare [HO69]. Hoare gives a set of axioms and rules of inference for proving partial correctness of Algol-like programs. The formulas in Hoare's system are triples of the form $\{P\}\ A\ \{Q\}$ where A is a statement in the programming language and P and Q are predicates expressed in the language of the first order predicate calculus (the assertion language). The partial correctness assertion $\{P\}\ A\ \{Q\}$ is true iff whenever P holds for the initial values of the program variables and A is executed, then either A will fail to terminate or Q will be satisfied by the final values of the program variables.

Examples of axioms and rules of inference for programming constructs include:

(1) $\{P^{e}_{x}\}\ x:=e\{P\}$ **assignment statement**

(2) $\dfrac{\{P\}\ A_1\ \{S\},\ \{S\}\ A_2\ \{R\}}{\{P\}\ (A_1;A_2)\ \{R\}}$ composition of statements

(3) $\dfrac{\{P\wedge b\}\ A\ \{P\}}{\{P\}\ \text{while } b \text{ do } A\ \{P\wedge\neg b\}}$ while statement

The axioms are designed to capture the meanings of the basic statements of the programming language. Proofs of correctness for composite statements are constructed by using these axioms together with a proof system for the assertion language.

Modern programming languages use statements considerably more complicated than those described above. One might wonder how well Hoare's axiomatic approach can be extended to handle more complicated statements. In this paper we will be interested in the question of whether there are programming languages for which it is impossible to obtain good Hoare-like axioms. This question is of obvious importance in the design of programming languages whose programs can be naturally proved correct.

But what is a good Hoare-like axiom? One property a good axiom system should have is soundness ([HO74], [DO76]). A deduction system is sound iff every statement which is provable within the system is, indeed, true. Another property is completeness [CO75]. A deduction system is complete if every true statement is provable. One suspects from the Gödel incompleteness theorem that, if the deduction system for the assertion language is axiomatizable and if a sufficiently rich interpretation (such as number theory) is used for the assertion language, then for any (sound) system of Hoare-like axioms, there will be assertions {P} A {Q} which are true but not provable within the system. One might wonder, however, if this incompleteness of the Hoare-like axiom systems reflects some inherent complexity of the programming language constructs or whether it is due entirely to the incompleteness of the assertion language. If, for example, we are dealing with the integers, then for any consistent axiomatizable proof system there will be predicates which are true of the integers but not provable within the system. How can we talk about the completeness of a Hoare-like axiom system independently of its assertion language?

One way of answering this question is due to S. Cook [CO75]. He gives a Hoare-like axiom system for a subset of Algol including the while statement and non-recursive procedures. He then proves that if there is a complete proof system for the assertion language (e.g. all true statements of the assertion language) and if the assertion language satisfies a certain natural expressibility condition, then every true partial correctness assertion will be provable. Gorelick [GO75] extends Cook's work to handle recursive procedures. Other completeness results are given by deBakker and Meertens [DE73] and by Manna [MA70].

In this paper we extend the work of Cook and Gorelick by showing that there are natural control mechanisms for which it is impossible to obtain sound and complete sets of axioms in the sense described above. While such incompleteness is expected for data structures (e.g. the integers, stacks, queues, etc.), it is surprising that it should exist for control structures.

The first programming language feature considered is recursive procedures with procedure parameters (provided that static scope is used and global variables are allowed). This result is surprising for two reasons. First, it holds even if we disallow calls of the form "call P(. . . , P)".[1] Secondly, we show that it is possible to obtain a sound and complete system of Hoare-like axioms (using static scope and allowing global variables) if we either (a) allow recursive procedures with variable parameters (call by simple name) but disallow

---

[1] Calls of the form "Call P(. . . , P)" appear to be necessary if one wants to directly simulate the lambda calculus by passing procedure parameters.

procedure parameters or (b) allow procedure parameters but require that procedures be nonrecursive.

An independent source of incompleteness is the coroutine construct. If procedures are not recursive, there is a simple method for proving correctness of coroutines, based on the addition of auxiliary variables [OW76]. If, however, procedures are recursive, we show that no such simple method can give completeness. These observations generalize to languages with parallelism and recursion.

Incompleteness results can also be obtained for (a) call by name parameter passing with functions and global variables and (b) label variables with retention. All such features are too complicated for a simple axiomatic description of the type advocated by Hoare and thus in a sense inherently difficult to prove correct.

The development of these results is divided into two parts—the first dealing with procedures with procedure parameters and the second dealing with the coroutine construct. In section 2 of this paper a simple programming language is described which allows static scope, global variables and procedures with procedure parameters. A formal semantics for the language is given together with a precise definition of when $\{P\}$ A $\{Q\}$ is true. There follows a discussion of Cook's expressibility condition. In sections 3 and 4 Hoare-like axiom systems are given for this language. These axiom systems are shown to be both sound and complete, subject to the restrictions on recursion and procedures as parameters described above. In section 5 we prove that there can be no Hoare-like axiom system for the full language which is both sound and complete. Sections 6, 7, 8, and 9 are devoted to a discussion of completeness and incompleteness results for coroutines and essentially follow the same outline as was used in the first part of the paper. The paper concludes with a brief account of results obtained by the author but not reported elsewhere in the paper and a discussion of how the results described in the paper may be interpreted.

2. A Simple Programming Language and its Semantics

As in [CO75] we distinguish two logical systems involved in discussions of program correctness—the assertion language L_A in which predicates describing a program's behavior are described and the expression language L_E in which the terms forming the right hand sides of assignment statements and the quantifier free boolean expressions of conditionals and while statements are specified. Both L_A and L_E are assumed to be first order languages with equality and L_A is an extension of L_E. Thus, in particular, the variables of L_E will be a proper subset of the variables of L_A. The variables of L_E are called program identifiers (PROG_ID) and are assumed to be ordered by the positive integers. The variables of L_A are called variable identifiers (VAR_ID). An interpretation I for L_A consists of a set D (the domain of the interpretation) and an assignment of functions on D to the function symbols of L_A and predicates on D to the predicate symbols of L_A. We will use the notation $|I|$ to represent the cardinality of the domain of I. Once an interpretation I has been specified, meanings may be assigned to the variable free terms and closed formulas of $L_A(L_E)$ using the standard technique of first order model theory.

Let I be an interpretation with domain D. A <u>state</u> is a mapping from a <u>finite</u> <u>subset</u> of VAR_ID to D. If s is a state, i a variable identifier, and d an element of D, then $s[i \leftarrow d]$ is the function with domain $DOM(s) \cup \{i\}$ which is given by

$$s[i \leftarrow d]\ (v) = \begin{cases} d & \text{if } v = i \\ s(v) & \text{if } v \neq i,\ v \in \text{VAR\_ID} \end{cases}$$

Similarly, if $A \subseteq DOM(s)$ then $s|_A$ is the function with domain A which is given by

$$s|_A\ (v) = s(v) \text{ for } v \in A$$

In particular DEL(s,i) is the function with domain $DOM(s)-\{i\}$ given by

$$DEL(s,i) = s|_{DOM(s)-\{i\}}$$

i.e. DEL(s,i) is the function obtained from s by restricting its domain to exclude i. If P is a formula of the assertion language L_A with free variables $x_1,x_2,\ldots,x_n$ then we will use the notation P(s) to mean $P\,\dfrac{s(x_1),\ldots,s(x_n)}{x_1,\ldots,x_n}$.

An <u>environment</u> π is a mapping from a <u>finite</u> <u>subset</u> of PROC_ID to FORMAL_PARAMETER_LISTS X STMTS. Informally $\pi\ (q) = <(\bar{x};\bar{p}),\ K>$ means that the procedure with name q has declaration "q:proc $(\bar{x}:\bar{p})$; K end" where $\bar{x}$ are the <u>formal</u> <u>variable</u> <u>parameters</u> and $\bar{p}$ are the <u>formal</u> <u>procedure</u> <u>parameters</u>. The notation $\pi[q \leftarrow <(\bar{x}:\bar{p}),\ K>]$ is defined in a manner similar to that given above for $s[i \leftarrow a]$ and should be self-explanatory.

The meaning function $M = M_I$ associates with a statement A, state s and environment π a new state s′. Intuitively s′ is the state resulting if A is executed with initial state s and initial environment π. The definition of M is given operationally in a rather non-standard manner which makes extensive use of renaming. This type of definition has the following two advantages:

a) It is very close to the original statement of the <u>copy rule</u> in the Algol 60 report [NA63]—thus there should be no question that we are using <u>static scope</u>.

b) It simplifies the proof of soundness and completeness for the Hoare-like axioms given in later sections.

The definition of M[A] (s)(π) is by <u>cases on A</u>:

(1) A <u>is</u> "begin new x; B end" → DEL(M [begin B $\frac{x^i}{x}$ end] (s′) (π), x^i) where i is the index of the first program identifier not appearing in A, π, or DOM(s) and s′ = $s[x^i \leftarrow e_0]$. (e_0 is a special domain element which is used as the initial value of program identifiers.)

(2) A <u>is</u> "begin q: proc($\overline{x}$:$\overline{p}$); K end; B end" → M[begin B $\frac{q^i}{q}$ end] (e) (π′) where i is the index of the first procedure identifier not occurring in A or π and π′ = $\pi[q^i \leftarrow <(\overline{x}:\overline{p}),\ K\ \frac{q^i}{q}>]$. Note that we are assuming that the syntax of allowable programs has been restricted to require that procedures be declared before they are used.

(3) A <u>is</u> "begin B_1; B_2 end" → M[begin B_2 end](M[B_1] (s) (π)) (π)

(4) A <u>is</u> "begin end" → s

(5) A <u>is</u> "x := e" $\rightarrow$ s' where $s' = s[x \leftarrow I[e(s)]]$

(6) A <u>is</u> "$b \rightarrow A_1,A_2$" $\rightarrow$ $\begin{cases} M[A_1]\ (s)(\pi) & \text{if } I[\ b(s)] = \text{true} \\ M[A_2]\ (s)(\pi) & \text{o.w.} \end{cases}$

(7) A <u>is</u> "b*A" $\rightarrow$ $\begin{cases} M[b*A](M[A](s)(\pi))(\pi) & \text{if } I[b(s)] = \text{true} \\ s & \text{o.w.} \end{cases}$

("b*A" is our short-hand notation for the statement "while b do A")

(8) A <u>is</u> "call $q(\bar{a}\colon \bar{P})$" $\rightarrow M[K \frac{\bar{a}}{\bar{x}} \frac{\bar{P}}{\bar{p}}](s)(\pi)$ where $\pi(q) = \langle(\bar{x}\colon\bar{p}), K\rangle$. Here $\bar{a}$ is the list of <u>actual variable parameters</u> and $\bar{P}$ is the list of <u>actual procedure</u> parameters. To simplify the treatment of parameters we arbitrarily restrict the entries in $\bar{a}$ to be simple program identifiers.

If P is a formula of the assertion language L_A which has free variables $x_1,x_2,\ldots,x_n$, then we identify P with the set of all those states s with domain $\{x_1,x_2,\ldots,x_n\}$ such that if $s(x_i) = c_i$ for $1 \le i \le n$ then $P\frac{c_1,\ldots,c_n}{x_1,\ldots,x_n}$ is true with respect to I. By convention the predicate TRUE is identified with the set containing the "empty" function $s\colon \phi \rightarrow D$, while the predicate FALSE is identified with the empty state set. Under these conventions note that the implication $P \rightarrow Q$ will be true with respect to I if

$$\forall s[s \in P \Rightarrow s|_{free(Q)} \in Q]$$

where free(Q) is the set of program identifiers which are free in Q.

The partial correctness assertions which we will need are somewhat more complicated than those discussed in the introduction since we are attempting to handle procedures. We will use assertions of the form $\{P\}$ A $\{Q\}/E$ where A is a program statement, P, Q are formulas of L_A and E is the set of procedure declarations accessible to A.

2.1 Definition:

We say that $\{P\}$ A $\{Q\}/E$ is true with respect to I ($\models_I \{P\}$ A $\{Q\}/E$) iff

$$\forall s, s'[s \in P \wedge M[A]\ (s)(\pi) = s' \Rightarrow s'|_{free(Q)} \in Q] \quad \text{where}$$

π is the environment corresponding to E, i.e. $\pi(q) = \langle(\bar{x}:\bar{p}), K\rangle$ iff "q: proc $(\bar{x}:\bar{p})$; K end" $\in$ E.

This is the usual definition of partial correctness. Note that in order for $\models_I \{P\}$ A $\{Q\}/E$ we are implicitly requiring that free(Q) $\subseteq$ free(P). The program identifiers which are global to A or to some procedure in E must also be a subset of free(P).

2.2 Definition:

We say that the assertion language L_A is expressive with respect to expression language L_E and the interpretation I iff for all A, Q, π there is a formula of L_A which expresses R[A] (π) (Q) (i.e. F[A] (π)(Q)) where

$$R[A]\ (\pi)(Q) = \{s \mid DOM(s) \text{ consists of those variables which are free in Q or global to A or some procedure in } \pi \text{ and } M[A]\ (s)(\pi)\uparrow \text{ or } M[A]\ (s)(\pi)|_{free(Q)} \in Q\}$$

and

$$F[A]\ (\pi)(Q) = \{M[A]\ (\pi)(s)\ s \in Q\}.$$

We prove below that $R[A]\ (\pi)(Q)$ is the <u>weakest</u> <u>precondition</u> corresponding to A, π, and Q.

<u>2.3 Proposition</u>:

Suppose that L_A is expressive relative to L_E and I, let π be the environment corresponding to E, then

(a) $\models_I \{R[A]\ (\pi)(Q)\}\ A\ \{Q\}/E$

(b) $\models_I \{P\}\ A\ \{Q\}/E \Rightarrow \models_I P \rightarrow R[A]\ (\pi)(Q)$

<u>Proof</u>:

(a) $s \in R[A]\ (\pi)(Q)$ and $M[A]\ (s)(\pi) = s'$ implies that $s'|_{free(Q)} \in Q$.

(b) ($\Rightarrow$) Suppose that $\{P\}\ A\ \{Q\}/E$ is true, then $s \in P$ implies that either $M[A]\ (s)(\pi)\uparrow$ or $M[A]\ (s)(\pi)|_{free(Q)} \in Q$. Hence $s \in P$ implies $s|_{free(R[A]\ (\pi)(Q))} \in R[A]\ (\pi)(Q)$.

Thus $\models_I P \rightarrow R[A]\ (\pi)(Q)$.

($\Leftarrow$) $s \in P$ implies $s|_{free(R[A]\ (\pi)(Q))} \in R[A]\ (\pi)(Q)$.

Thus if $s \in P$ and $s' = M[A]\ (s)(\pi)$ then $s'|_{free(Q)} \in Q$.

Hence $\models_I \{P\}\ A\ \{Q\}/E$.

Similarly, we may prove that $F[A]\ (\pi)(Q)$ is the <u>strongest</u> <u>post</u> <u>condition</u> corresponding to A, π, and Q. It is possible to prove that the definition of "expressive" given above is not changed whether we use <u>strongest</u> <u>post</u> <u>condition</u> or <u>weakest</u> <u>precondition</u>. In section 4 we show that if the assertion language is expressive, then the invariants of while loops will be representable by formulas of the assertion language. Note that it is relatively easy to come up with examples of situations in which the assertion language L_A is <u>not</u> expressive with respect to the expression language L_E and I. Cook [CO75] gives the example in which the assertion language L_A and the expression language are both the language of Presburger Arithmetic. Note that since the programming language allows iteration, every partial recursive function can be computed by some program even if we only allow addition and subtraction (not multiplication) on the right hand sides of assignment statements.

Fortunately, more realistic choices for L_A, L_Σ, and I do give expressibility. If L_A and L_Σ are both the full language of number theory, and I is an interpretation in which the symbols of number theory receive their usual meanings, then L_A is expressive with respect to L_Σ and I. This is true because $F[A](P)$ can be viewed as the direct image of an arithmetical set by a partial recursive function and will therefore also be arithmetical. Also, if the domain of I is infinite, then we have expressibility.

<u>2.4 Proposition</u>:

If L_A, L_Σ are first order languages with equality and the domain of I is a finite set, then L_A will be expressive with respect to L_Σ and I.

<u>Proof</u>: Let D be the domain of I and suppose that $|D| < \infty$. Let A, Q, and π be given. Suppose that $x_1, \ldots, x_n$ are the variables which occur free in A, Q, π. Because of the finiteness of D there are only finitely many (say m) n-tuples $\langle a_1^j \ldots a_n^j \rangle$ such that if we define $s_j(x_i) = a_i^j$ then either $M[A](s_j)(\pi)\uparrow$ or $M[A](s_j)(\pi)\big|_{free(Q)} \in Q$. Let $R = \bigvee_{1 \le j \le m} x_1 = a_1^j \wedge x_2 = a_2^j \wedge \ldots \wedge x_n = a_n^j$, then it is not difficult to show that R expresses $R[A](\pi)(Q)$. We will see, however, that as a consequence of lemma 5.1 there cannot exist an algorithm for

computing R from A, π, and Q.

3. Hoare-like Axioms for Static Scope, Global Variables, Etc.

In this section we give a deductive system H for handling (a) static scope, (b) global variables, and (c) nonrecursive procedures with procedure parameters which is both sound and (in the sense of Cook) complete. Let T be a proof system for L_A, then a proof in the system (H,T) is defined in the usual way (see [GO75]) as a sequence of formulas of H or T each of which is either an axiom or follows from preceding formulas by a rule of inference. If F is a formula in such a sequence then we write $\vdash F$ and call F a theorem of (H,T). The axioms and rules of inference of H are given below.

(H1) $$\frac{\{U \wedge x^i = e_0\}\ \text{begin } A\,\frac{x^i}{x}\ \text{end}\ \{V\}/E}{\{U\}\ \text{begin new } x;\ A\ \text{end}\ \{V\}/E}$$

where i is the index of the first program identifier not appearing in A, E, or P.

(H2) $$\frac{\{U\}\ \text{begin } A\,\frac{q^i}{q}\ \text{end}\ \{V\}/E\ \cup\ \{q^i\text{: proc } (\bar{z}:\bar{p});\ K\,\frac{q^i}{q}\ \text{end}\}}{\{U\}\ \text{begin } q\text{: proc } (\bar{z}:\bar{p});\ K;\ \text{end};\ A;\ \text{end}\ \{V\}/E}$$

where i is the index of the first procedure identifier not appearing in K, A, or E.

(H3) $$\frac{\{U\}\ A\ \{V\}/E_1}{\{U\}\ A\ \{V\}/E_2}$$

provided that $E_1 \subseteq E_2$ and E_2 does not contain the declarations of two different procedures with the same name.

(H4) (a) $$\frac{\{U\}\ A\ \{V\}/E}{\{U\}\ \text{begin } A\ \text{end}\ \{V\}/E}$$

(b) $$\frac{\{U\}\ A_1\ \{V\}/E,\ \{V\}\ \text{begin } A_2\ \text{end}\ \{W\}/E}{\{U\}\ \text{begin } A_1;\ A_2\ \text{end}\ \{W\}/E}$$

(H5)-(H8) Usual axioms for assignment, conditional, while, and consequence (see HO71]). Note of course that each of these axioms must be modified to make explicit the set E of procedure declarations.

(H9)
$$\frac{\{U\}\ K\frac{\bar{a}}{\bar{x}}\frac{\bar{p}}{p}\ \{V\}/E \text{ which includes } \bar{P}}{\{U\}\ \text{call } h(\bar{a}:\bar{p})\ \{V\}/E \cup \{h\text{: proc } (\bar{x}:\bar{p});\ K \text{ end}\}}$$

provided that E does not already contain a procedure declaration "h: $(x':\bar{p}')$; K′ end" different from "h: proc $(\bar{x}:\bar{p})$; K end".

Note that axiom H9 is extremely simple; it merely states that if we can prove the body of the procedure correct when we substitute the actual parameters for the formal parameters then we may conclude that the procedure call is correct. We illustrate how these axioms may be used to handle static scope by considering a simple example involving nonrecursive procedures with procedure parameters. The example is designed so that the procedure g is called in an environment which is different from that in which it was defined.

<u>Example</u> {true}

```
begin
     h: proc(:p); begin new x; x := 1; call p(z); end
     f: proc( );
        begin new x;
             g: proc(y); y := x; end
             x := 2;
             call h(:g);
             end;
```

```
      end;
    z := 3;
    call f( );
    end;
{z = 2}/ φ
```

The reader should verify that this precondition—post condition assertion is correct with respect to the semantics given in section 2 (static scope—the standard Algol 60 scope rule. Procedure calls are expanded in the environment of the procedure's declaration.) Note also that if dynamic scope is used (i.e. if when an identifier is referenced the most recently declared, active copy of the identifier is used), the correct post-condition is z = 1.

Proof:

(1) {x′ = 2} z := x′ {z = 2}/ φ — H5

(2) {x′ = 2} begin call g′(z) end {z = 2}/ {g′: proc(y); y := x′; end} — H9, H4a

(3) $\{x' = 2 \wedge x'' = e_0\}$ begin x″ := 1; call g′(z); end {z = 2}/ {g: proc;...x′...end} — H5, H4b

(4) {x′ = 2} begin new x; x := 1; call g′(z); end {z = 2}/ {g: proc...x′...end} — H1

(5) {x′ = 2} call h(:g) {z = 2}/ {g: proc...x′...end, h: proc...end} — H9

(6) $\{x' = e_0\}$ begin x′ := 2; call h(:g) end {z = 2}/ {g: proc...x′...end, h: proc...end} — H5, E4a,b

```
(7)  {x' = e_0} begin
          g: proc(y); y := x'; end;
          x' := 2;
          call h(:g);
          end;                                          H2
               {z = 2}/{h: proc...end}

(8)  {true}  begin
             new x;
             g: proc(y); y := x; end
             x := 2;
             call h(:g);
             end;                                       H1
                  {z = 2}/{h: proc...end}

(9)  {true}  call f( ) {z = 2}/{h: proc...end, f: proc...end}    H8

(10) {true}  begin z := 3; call f( ); end  {z = 2}/
     {h: proc...end, f: proc...end}                     H5,H4a,b

(11) {true} begin
             h: proc (:p);...end;
             f: proc (  );...end;
             z := 3;                      By two applications
                                          of H2
             call f(  );
             end;
                {z = 3}/ ϕ
```

This completes the proof. In the next section we will show that axioms H1-H8 above are sound and, in a certain sense, complete.

4. Soundness and Completeness

We consider the programming language described in section 2 with the restriction that procedures be nonrecursive. (Thus we are allowing (a) static scope, (b) global variables, and (c) nonrecursive procedures with procedure parameters.)

4.1 Theorem:

For all L_A, L_E, and I if (a) T is a complete proof system for L_A relative to I and if (b) L_A is expressive relative to L_E and I, then for all partial correctness assertions of the form {P} A {Q}/E we have

$$\models_I \{P\}\ A\ \{Q\}/E \Leftrightarrow \underset{H,T}{\vdash} \{P\}\ A\ \{Q\}/E$$

Proof: The proof will be divided into two parts. In the first part we prove soundness, i.e. if $\underset{H,T}{\vdash}$ {P} A {Q}/E then $\models_I$ {P} A {Q}/ E. In the second part we prove completeness—if $\models_I$ {P} A {Q}/E then $\underset{H,T}{\vdash}$ {P} A {Q}/E.

Soundness: We must show that for each of the rules of inference H1-H9 that if all of the hypotheses of the rule are true (with respect to I) then the conclusion will be true also. Here we examine H1, H2, and H9—the remaining ones are left to the reader.

First H1. Assume that $\{U \wedge x^i = e_0\}$ begin $A \frac{x^i}{x}$ end $\{V\}/E$ is true with respect to I. Let π be the environment corresponding to E. Thus by the definition of partial correctness we have

$$\forall\ s, s'\ [s \in (U \wedge x^i = e_0) \wedge s' = M[\text{begin } A \tfrac{x^i}{x} \text{ end}](s)(\pi) \Rightarrow s'\big|_{free(V)} \in V]$$

Since x^i does not occur free in V, if we let $s'' = DEL(s', x^i)$ then

$$s''\big|_{free(V)} \in V \text{ iff } s'\big|_{free(V)} \in V.$$

Thus

$$\forall s, s'' [s \in (U \wedge x^i = e_0) \wedge s'' = DEL(M[\text{begin } A \tfrac{x^i}{x} \text{ end}]\ (s)(\pi),\ x^i)$$

$$\Rightarrow s''\big|_{free(V)} \in V]$$

Let $s^* = DEL(s, x^i)$. Then

$$M[\text{begin new } x; A \text{ end}]\ (s^*)(\pi) = DEL(M[\text{begin } A \tfrac{x^i}{x} \text{ end}]\ (s)(\pi),\ x^i)$$

Hence it follows that

$$\forall s^*, s'' [s^* \in U \wedge s'' = M[\text{begin new } x; A \text{ end}](s^*)(\pi) \Rightarrow s''\big|_{free(V)} \in V]$$

or $\models_I \{U\}$ begin new x; A end $\{V\}/E$.

Next we consider H2. Assume that

$$\models_I \{U\} \text{ begin } A \tfrac{q^i}{q} \text{ end } \{V\}/E \cup \{q^i: proc(\overline{x}:\overline{p});\ K \tfrac{q^i}{q} \text{ end}\}$$

is true with respect to I. Let π be the environment corresponding to E and let $\pi' = \pi[q^i \leftarrow <(\overline{x}:\overline{p})\ K, \tfrac{q^i}{q}>]$.

By the definition of partial correctness

$$\forall s, s' [s \in U \wedge s' = M[\text{begin } A \tfrac{q^i}{q} \text{ end}]\ (s)(\pi') \rightarrow s'\big|_{free(V)} \in V]$$

But $M[\text{begin } q: proc(\overline{x}:\overline{p});\ K \text{ end};\ A \text{ end}]\ (s)(\pi) = M[\text{begin } A \tfrac{q^i}{q} \text{ end}](s)(\pi')$

Thus

$$\forall s, s' [s \in U \wedge s' = M[\text{begin } q: proc(\overline{x}:\overline{p});\ K \text{ end};\ A \text{ end}]\ (s)(\pi)$$

$$\Rightarrow s'\big|_{free(V)} \in V]$$

or

$\models_I \{U\}$ begin q: proc($\bar{x}$:$\bar{p}$); K end; A end $\{V\}/E$ as required.

Finally we consider H9. Assume that $\{U\}\ K\ \frac{\bar{a}}{\bar{x}}\ \frac{\bar{P}}{\bar{p}}\ \{V\}/(E$ which includes $\bar{P}$) is true in I. Let π be the environment map corresponding to E, then we have by the definition of partial correctness that

$$\forall\ s, s'[s \in U \wedge s' = M[K\ \frac{\bar{a}}{\bar{x}}\ \frac{\bar{P}}{\bar{p}}]\ (s)(\pi) \Rightarrow s'\Big|_{free(V)} \in V]$$

But $M[\text{call } q(\bar{a}:\bar{P})](s)(\pi') = M[K\ \frac{\bar{a}}{\bar{x}}\ \frac{\bar{P}}{\bar{p}}]\ (s)(\pi')$

where $\pi' = \pi[q \leftarrow <(\bar{x}:\bar{p}),\ K>]$. Since recursion has been disallowed, we have

$$M[K\ \frac{\bar{a}}{\bar{x}}\ \frac{\bar{P}}{\bar{p}}]\ (s)(\pi') = M[K\ \frac{\bar{a}}{\bar{x}}\ \frac{\bar{P}}{\bar{p}}]\ (s)(\pi)$$

thus

$$M[\text{call } q\ (\bar{a}:\bar{p})]\ (s)(\pi') = M[K\ \frac{\bar{a}}{\bar{x}}\ \frac{\bar{P}}{\bar{p}}]\ (s)(\pi)$$

It follows that

$$\forall\ s, s'[s \in U \wedge s' = M[\text{call } q(\bar{a}:\bar{P})]\ (s)(\pi') \Rightarrow s'\Big|_{free(V)} \in V]$$

or that $\models_I \{U\}$ call q($\bar{a}$:$\bar{p}$)$\{V\}/E \cup \{$q: proc($\bar{x}$:$\bar{p}$); K end$\}$ which is the desired conclusion. The other cases H3-H8 are similar to the ones considered above.

<u>Completeness</u>: We show that if T is a complete proof system for L_A relative to I and if L_A is expressive relative to L_E and I, then for all partial correctness assertions of the form $\{U\}\ A\ \{V\}/E$,

$$\models_I \{U\}\ A\ \{V\}/E \Rightarrow\ \vdash_{H,T} \{U\}\ A\ \{V\}/E\ .$$

The proof will be by induction on the structure of A. We will show that $\vdash_{H,T} \{U\}\ A\ \{V\}/E$ if A is an <u>atomic</u> statement of the programming language (e.g. an assignment statement). We will also show that if A is a composite statement (e.g. "begin new x; B end" or "while b do A") then to establish $\{U\}$ A $\{V\}/E$ it is sufficient to first establish $\{U'\}$ A' $\{V'\}/E$ where A' is either shorter than A or involves fewer procedure calls.

Case 1: Suppose that A is "begin new x; B end" and that $\{U\}$ begin new x; B end $\{V\}/E$ is true with respect to I. Then by the definition of partial correctness

$$\forall\ s, s'[s \in U \wedge s' = M[\text{begin new x; B end}]\,(s)(\pi) \Rightarrow s'\big|_{free(V)} \in V]$$

where π is the environment corresponding to E. But

$$M[\text{begin new x; B end}]\,(s)(\pi) = DEL(M[\text{begin B } \tfrac{x^i}{x} \text{ end}]\,(s^*)(\pi),\ x^i)$$

where i is the index of the first program identifier not appearing in A, π, or DOM(s) and $s^* = s[x^i \leftarrow e_0]$. Since x^i does not occur free in U or V, we have

(a) if $s^* = s[x^i \leftarrow e_0]$ then $s \in U$ iff $s^* \in U \wedge x^i = e_0$, and

(b) if $s' = DEL(s'', x^i)$ then $s'\big|_{free(V)} \in V$ iff $s''\big|_{free(V)} \in V$.

It follows that if i is the index of the first program identifier not appearing free in A, E, or free(U) then

$$\forall\ s^*, s''[s^* \in (U \wedge x^i = e_0) \wedge s'' = M[\text{begin B } \tfrac{x^i}{x} \text{ end}]\,(s^*)(\pi) \Rightarrow s''\big|_{free(V)} \in V]$$

Applying the definition of partial correctness again we have that

$\{U \wedge x^i = e_0\}$ begin B $\frac{x^i}{x}$ end $\{V\}/E$.

Since "begin B $\frac{x^i}{x}$ end" is strictly shorter than "begin new x; B end", we have by the induction hypothesis that $\vdash_{H,T} \{U \wedge x^i = e_0\}$ begin $B \frac{x^i}{x}$ end $\{V\}/E$. Using the fact that i is the index of the first program identifier not appearing in A, E, or U, we may use H1 to conclude $\vdash_{H,T} \{U\}\ A\ \{V\}/E$ as required.

Case (2): Suppose that A is "begin q: proc$(\overline{x}:\overline{p})$; K end; B end" and that $\{U\}$ begin q: proc$(\overline{x}:\overline{p})$; K end; B end $\{V\}/E$ is true with respect to I. By the definition of partial correctness

$$\forall\ s, s'[s \in U \wedge s' = M[\text{begin q: proc}(\overline{x}:\overline{p});\ K\ \text{end};\ B\ \text{end}]\ (s)(\pi) \Rightarrow s'\big|_{free(V)} \in V]$$

where π is the environment corresponding to E. But M[begin q: proc $(\overline{x}:\overline{p})$; K end; B end] $(s)(\pi)$ = M[begin $B \frac{q^i}{q}$ end] $(s)(\pi')$ where i is the index of the first procedure identifier not occurring in A or π and $\pi' = \pi[q^i \leftarrow <(\overline{x}:\overline{p}), K \frac{q^i}{q}>]$

thus

$$\forall\ s, s'[s \in U \wedge s' = M[\text{begin } B \frac{q^i}{q} \text{ end}]\ (s)(\pi') \Rightarrow s'\big|_{free(V)} \in V]$$

If we let E' be $E \cup \{q^i$: proc$(\overline{x}:\overline{p})$; $K \frac{q^i}{q}$ end$\}$ then π' is the environment corresponding to E' and by another application of the definition of partial correctness, $\models_I \{U\}$ begin $B \frac{q^i}{q}$ end $\{V\}/E'$. Since "begin $B \frac{q^i}{q}$ end" is strictly shorter than "begin q: proc$(\overline{x}:\overline{p})$; K end; B end" we conclude that $\vdash_{H,T} \{U\}$ begin $B \frac{q^i}{q}$ end $\{V\}/E$. It follows by axiom ?? that $\vdash_{H,T} \{U\}\ A\ \{V\}/E$ also.

Case (3): Suppose that A is "b^*A_1" and that $\{U\}\ b^*A_1\ \{V\}/E$ is true with respect to I. Since the assertion language L_A is expressive with respect to L_P and I, we know that there is a formula in L_A which represents $R[b*A_1](\pi)(V)$ where π is the environment corresponding to E. The reader may verify that each of the following assertions is correct.

(i) $\models_I U \to R[b*A_1](\pi)(V)$ and hence $\vdash_T U \to R[b*A_1](\pi)(V)$ since T is assumed to be a complete proof system for L_A.

(ii) $\models_I \{R[b*A_1](\pi)(V) \wedge b\}\ A_1\ \{R[b*A_1](\pi)(V)\}/E$ —so $\vdash_{H,T} \{R[b*A_1](\pi)(V) \wedge b\}\ A_1\ \{R[b*A_1](\pi)(V)\}/E$ follows by the induction hypothesis since A_1 is shorter than $b*A_1$.

(iii) $\models_I R[b*A_1](\pi)(V) \wedge \sim b \to V$ and hence $\vdash_{H,T} R[b*A_1](\pi)(V) \wedge \sim b \to V$ since T is a complete proof system for L_A.

The desired result $\vdash_{H,T} \{U\}\ A\ \{V\}/E$ follows by an application of the while axiom and the rule of consequence.

<u>Case (4)</u>: Suppose that A is "call $q(\bar{a}:\bar{P})$", that $\{U\}$ call $q(\bar{a}:\bar{P})$ $\{V\}/E$ is true with respect to I, that π is the environment corresponding to E, and that $\pi(q) = \langle(\bar{x}:\bar{p}), K\rangle$. By the definition of partial correctness:

$$\forall s, s'[s \in U \wedge s' = M[\text{call } q(\bar{a}:\bar{P})](s)(\pi) \Rightarrow s'|_{\text{free}(V)} \in V]$$

But $M[\text{call } q(\bar{a}:\bar{P})](s)(\pi) = M[K\frac{\bar{a}}{\bar{x}}\frac{\bar{P}}{\bar{p}}](s)(\pi)$

Hence

$$\forall s, s'[s \in U \wedge s' = M[K\frac{\bar{a}}{\bar{x}}\frac{\bar{P}}{\bar{p}}](s)(\pi) \Rightarrow s'|_{\text{free}(V)} \in V]$$

and we have

$\models_I \{U\}\ K\frac{\bar{a}}{\bar{x}}\frac{\bar{P}}{\bar{p}}\ \{V\}/E$. Since recursion is not allowed, $K\frac{\bar{a}}{\bar{x}}\frac{\bar{P}}{\bar{p}}$ does not involve the procedure q. Thus by the inductive assumption

$\vdash_{H,T} \{U\}\ K\frac{\bar{a}}{\bar{x}}\frac{\bar{P}}{\bar{p}}\ \{V\}/E$.

By axiom H9 we get

$\vdash_{H,T} \{P\}$ call $q(\overline{a}:\overline{P})$ $\{Q\}/E \cup \{q:$ proc$(\overline{x}:\overline{p})$; K end$\}$

Since "Q: proc$(\overline{x}:\overline{p})$; K end" $\in$ E, we get the desired result that

$\vdash_{H,T} \{P\}$ call $q(\overline{a}:\overline{P})\{Q\}/E$.

Cases 5-8 correspond to A being one of

a) "begin B_1; B_2 end"

b) "begin end"

c) "x := e"

d) "b $\rightarrow A_1$, A_2"

Since these cases are handled in a manner similar to that used in cases 1-4, they will be left to the interested reader. This completes the proof of theorem 1.

If we disallow procedure parameters, then it is possible to obtain a complete set of Hoare-like axioms even if the procedures may be recursive. The axioms given in Gorelick [GO75] can be used almost verbatim in spite of the static scope requirement. We replace H9 by axioms R1-R5 below.

$$\text{R1}\quad \frac{\{P\}\ \text{call}\ r\ (\overline{x})\ \{Q\}/E \vdash \{P\}\ K(r)\ \{Q\}/E,\ r\ \text{a dummy procedure name}}{\{P\}\ \text{call}\ q(\overline{x})\ \{Q\}/E \cup \{q\text{: proc}(\overline{x});\ K(q);\ \text{end}\}}$$

provided that E does not contain a procedure "q: proc$(\overline{x})$; K$'$ end" which is different from "q: proc$(\overline{x})$; K(q) end".

4.2 Definition:

Let the procedure q have declaration "q: proc$(\overline{x})$; K end". A variable y is active with respect to "call $q(\overline{a})$" if y is either global to K or y is in $\overline{a}$ (i.e. is an actual parameter of the call). If y is not active with respect to "call $q(\bar{a})$" then y is said to be <u>inactive</u> (with respect to that particular call). Similarly, a term of the assertion language L_A is <u>inactive</u> if it contains only inactive variables. A substitution σ is <u>admissible</u> with respect to "call $q(\bar{a})$" provided that it is a substitution of inactive terms for inactive variables.

R2) $$\frac{\{P\}\ \text{call } q(\bar{a})\ \{Q\}/E}{\{P\ \sigma\}\ \text{call } q(\bar{a})\ \{Q\ \sigma\}/E}$$ provided σ is admissible with respect to "call $q(\bar{a})$"

R3) $$\frac{\{P(u_0)\}\ \text{call } q(\bar{a})\ \{Q(u_0)\}/E}{\{\exists u_0 P(u_0)\}\ \text{call } q(\bar{a})\ \{\exists u_0 Q(u_0)\}/E}$$ provided that u_0 is inactive in "call $q(\bar{a})$"

(Rule R3 above is not absolutely necessary but its inclusion simplifies the proof of completeness.)

R4) $$\frac{\{P\}\ \text{call } q(\bar{a})\ \{Q\}/E}{\{P \wedge T\}\ \text{call } q(\bar{a})\ \{Q \wedge T\}/E}$$ provided that no variables which occur free in T are active in "call $q(\bar{a})$"

R5) $$\frac{\{P\}\ \text{call } q(\bar{x})\ \{Q\}/E}{\{P\ \frac{\bar{a}}{\bar{x}}\}\ \text{call } q(\bar{a})\ \{Q\ \frac{\bar{a}}{\bar{x}}\}/E}$$ provided that no variable free in P or Q occurs in $\bar{a}$ but not in the corresponding position of $\bar{x}$. ($\bar{x}$ is the list of <u>formal parameters</u> of q.)

<u>4.3 Theorem:</u>

Let H′ be the set of axioms H1-H8 together with axioms R1-R5. Restrict the programming language of section 2 so that procedure parameters are disallowed. For all L_A, L_E and I, if (a) T is a complete proof system for L_A relative to I and (b) L_A is expressive relative to L_E and I then for all partial correctness assertions of the form

$$\{P\}\ A\ \{Q\}/E$$

we have

$$\models_I \{P\}\ A\ \{Q\}/E \Leftrightarrow \vdash_{H,T} \{P\}\ A\ \{Q\}/E\ .$$

The proof of theorem 2 involves modifying the argument of [GO75] to apply to static scope. Since static scope may be handled by the techniques described earlier in this section, the proof will be omitted.

5. Recursive Procedures with Procedure Parameters

In this section we prove that there cannot be a sound, complete set of Hoare-like axioms for a programming language with (a) global variables, (b) static scope, and (c) recursive procedures with procedure parameters (sound and complete in the sense of section 3. The effect of stronger notions of expressibility, etc., will be discussed later.) To make the theorem stronger, we disallow calls of the form Call P(...P), i.e. we require that actual procedure parameters be names of procedures with no procedure formal parameters. Calls of the form Call P(...,P) appear to be necessary in order to directly simulate the lambda calculus by parameter passing.

5.1 Lemma:

Consider a programming language allowing recursive procedures with procedure parameters (assuming static scope and allowing global variables) then the halting problem for such a language is undecidable for all finite interpretations I with $|I| \geq 2$.

The proof of the lemma uses techniques developed by Jones and Muchnick [JO75]. Before we outline the proof, note that the lemma is not true for flowchart schemes or while schemes since in each of these cases if $|I| < \infty$ the program may be viewed as a finite state machine and we may test for termination (at least theoretically) by watching the execution sequence of the program to see if any program state is repeated. In the case of recursion one might expect that the program could be viewed as a type of push down automaton (for which the halting problem is decidable). This is not the case if we allow procedures as parameters. The Algol 60 execution rule, which says that procedure calls are elaborated in the environment of the procedure's declaration rather than in the environment of the procedure call, allows the program to access values normally buried in the runtime stack without first "popping the top" of the stack.

As in Jones and Muchnick [JO75] we show that it is possible to simulate a queue machine which has three types of instruction, a) Enqueue x — add the value of x to the rear of the queue, b) Dequeue x — remove the front entry from the queue and place in x, and c) If x = y then L_1 else L_2 — conditional branch. (By using procedures with procedure parameters instead of call by name as in Jones and Muchnick, we are able to avoid introducing any non-Algol 60 constructs.)

The queue is represented by the successive activations of a recursive procedure "sim" with the queue entries being maintained as values of the variable "top" which is local to "sim". Thus an addition to the rear of the queue may be accomplished by having "sim" call itself recursively. Deletions from the front of the queue are more complicated. "Sim" also contains a local procedure "up" which is passed as a parameter during the recursive call which takes place when an entry is added to the rear of the queue. In deleting an entry from the front of the queue, this parameter is used to return control to previous activations of "sim" and inspect the values of "top" local to those activations. The first entry in the queue will be indicated by marking (e.g. negating) the appropriate copy of "top". Suppose that the Queue machine program to be simulated is given by

$$P = 1 : INST_1 ; \ldots K : INST_k$$

then the simulation program (in the language of section 2) has the form

```
begin
     new inst, dummy, x1,...xn;
     diverge: proc; while true do null end: end diverge;
     sim: proc(inst:back);
          begin new top;
               up: proc(end_of_queue, next_to_last:);
                    begin
                         If top < 0
                         then begin
                              end_of_queue := top;
                              next_to_last := 1;
                              end;
                         else begin
                              call back(end_of_queue, next_to
                                 last:);
                              If next_to_last=1
                              then begin
                                   top := -top;
                                   next_to_last=0;
                              end;
                         end;
```

```
            end up;
        If inst=1 then "INST_1" else
        If inst=2 then "INST_2" else
              .
              .
              .
        If inst=K then "INST_K" else null;
      end;                          else return
    end sim;
  inst:=1;
  call sim(inst:diverge);
  end;
```

where "$INST_j$" is described by the three cases below.

(A) If $INST_j$ is "j:enqueue A;" then replace by:

```
    begin
        If inst=1 then top := -A; else top := A;
        inst := inst+1;
        Call sim(inst: up);
        return;
    end;
```

can't go back in case first instr.

(B) If $INST_j$ is "j:dequeue x" then replace by

```
    begin
        call back (x:dummy);
        x := -x;
        inst := inst+1;
    end;
```

(c) If $INST_j$ is "If $x_p = x_m$ then go to n" then replace by

```
begin
     If x_p = x_m
     then inst := n else inst := inst+1;
end;
```

Note that inst must have range of values $1 \le inst \le K+1$. If $|I| < K+1$ then it is necessary to represent inst by $inst^1, inst^2 \ldots inst^d$ when $inst^1, inst^2, \ldots inst^d$ is the binary representation of inst. This also eliminates the need for the arithmetic operation inst:= inst+1. The variables $x_1 \ldots x_n$ represent the program variables of the program P which is being simulated.

Now we return to the proof of the main theorem of this section. Suppose that we had a sound, complete Hoare-like proof system H for programs of the type described at the beginning of this section. Then for all L_A, L_E, and I, if

1) T is a complete proof system for L_A, and I and

2) L_A is expressive relative to L_E and I, then we should have

$$\models_I \{P\}\ A\ \{Q\}/_\phi \Longleftrightarrow \vdash_{H,T} \{P\}\ A\ \{Q\}/_\phi$$

We show that this leads to a contradiction. Choose I to be a finite interpretation with $|I| \ge 2$.

First observe that T may be chosen in a particularly simple manner; in fact, there is a decision procedure for truth of formulas in L_A relative to I. Secondly note that L_A is expressive relative to L_E and I. This was shown by proposition 5.2.4 of section 2 since I is finite. Thus both hypothesis 1) and 2) are satisfied. From the definition of partial correctness, we see that $\{true\}\ A\ \{false\}/\phi$ iff

A diverges for the initial values of its global variables. By the main lemma above, we conclude that the set of programs A such that $\vdash_I$ {true} A {false}/ ϕ holds is not r.e. On the other hand if we had

$$\vdash_I \{\text{true}\}\ A\ \{\text{false}\}/\ \phi \Leftrightarrow \vdash_{H,T} \{\text{true}\}\ A\{\text{false}\}/\phi,$$

then we could enumerate those programs A such that $\vdash_I$ {true} A {false}/ ϕ holds (simply enumerate all possible proofs and use the decision procedure for T to check applications of the rule of consequence). This, however, is a contradiction.

6. Coroutines

A coroutine will have the form:

coroutine Q_1, Q_2 end

Q_1 is the main-routine; execution begins in Q_1 and also terminates in Q_1 (the requirement that execution terminate in Q_1 is not necessary but simplifies the axiom for coroutines). Otherwise Q_1 and Q_2 behave in identical manners. If an "exit" statement is encountered in Q_1, the next statement to be executed will be the statement following the last "resume" statement executed in Q_2. Similarly, the execution of a "resume" statement in Q_2 causes execution to be restarted following the last exit statement executed in Q_1. If the "exit" ("resume") statement occurs within a call on a recursive procedure then execution must be restarted in the correct activation of the procedure. A simple example of a coroutine is given below:

```
coroutine
    begin
        while x-y < z do
```

```
                x := x+2;
                y := y+2;
                exit;
                end;
            end,
        begin
            while true do
                y := y-1;
                resume;
                y := y-2;
                resume;
                end;
            end;
        end;
```

This example illustrates how complicated a short program involving the coroutine construct can be if exit and resume statements are included within while loops. Note that if x and y are zero initially, then when the coroutine terminates $y = \lceil z/3 \rceil$.

7. Semantics of Coroutines

M[A] (s)(π) is defined as before except for M[coroutine Q_1, Q_2 end] (s)(π) which is defined in terms of two mutually recursive procedures C1 and C2 (one for each coroutine) as follows:

M[coroutine Q_1, Q_2 end] (s)(π) = C1[Q_1, π, Q_2, π, s] where C1[R_1, π_1, R_2, π_2, s] is defined by cases on R_1

(1) begin new x; A end; $R_1' \rightarrow$ C1[begin A $\frac{x^i}{x}$ end; R_1', π_1, R_2, π_2, s′] where i is the index of the first unused program identifier and

$s' = s[x^i \leftarrow e_0]$.

(2) begin q: proc($\bar{x}$); K end A end; $R_1' \rightarrow$ C1[begin A $\frac{q^i}{q}$ end; R_1', π_1', R_2, π_2, s] where i is the index of the first unused procedure name and $\pi_1' = \pi_1[q^i \leftarrow \langle(\bar{x}), K\rangle]$

(3) begin A end; $R_1' \rightarrow$ C1[A; R_1', π_1, R_2, π_2, s]

(4) exit; $R_1' \rightarrow$ C2[R_1', π_1, R_2, π_2, s]

(5)-(8) cases corresponding to assignment, conditional, while, and procedure call—see section 2.

(9) Λ (i.e. R_1 is the empty string) $\rightarrow$ s

The definition of C2[R_1, π_1, R_2, π_2, s] is the dual of the definition given above except that C2[R_1, π_1, Λ, π_2, s] = C1[R_1, π_1, Λ, π_2, s]. Thus execution of the coroutine always terminates in Q_1. Note also that the semantics given above do not allow for nested coroutines. The semantics could be modified to handle this case but nesting of coroutines is unnecessary in order to illustrate the problem that we are interested in here.

8. Axioms for Coroutines (no recursion)

In this section we give a "good" set of axioms for coroutines (nonrecursive procedures only) and describe a technique for proving correctness of coroutines which is based on the addition of "auxiliary variables". This technique was suggested in part S. Owicki. It is different from the technique described by Clint [CL73], in that the auxiliary variables represent program counters (therefore have bounded magnitude) rather than arbitrary stacks.

Axioms for Coroutines:

C1. $\{P'\}$ exit $\{R'\} \vdash \{P \wedge b\}\ Q_1\ \{R\}$
$$\frac{\{R'\}\ \text{resume}\ \{P'\} \vdash \{P' \wedge b\}\ Q_2\ \{R'\}}{\{P \wedge b\}\ \text{coroutine}\ Q_1;\ Q_2\ \text{end}\ \{R\}}$$

provided no variable free in b is global to Q_1. (This axiom is a modification of the one in [CL73].)

C2. $$\frac{\{P\}\ \text{exit}\ \{Q\}}{\{P \wedge C\}\ \text{exit}\ \{Q \wedge C\}}$$

provided that C does not contain any free variables that are changed by Q_2. (Here we assume that "exit" occurs in statement Q_1 of "coroutine Q_1; Q_2 end".)

C3. $$\frac{\{P\}\ \text{resume}\ \{Q\}}{\{P \wedge C\}\ \text{resume}\ \{Q \wedge C\}}$$

provided that C does not contain any free variables that are changed by Q_1. (Here we assume that "resume" occurs in statement Q_2 of "coroutine Q_1; Q_2 end".)

C4. Let AV be a set of variables such that $x \in AV \Rightarrow x$ appears in S' only in assignment $y := E$, with $y \in AV$. Then if P and Q are assertions which do not contain free any variables from AV and if S is obtained from S' by deleting all assignments to variables in AV, then

$$\frac{\{P\}\ S'\ \{Q\}}{\{P\}\ S\ \{Q\}}$$ (This axiom was taken from [OW76].)

We illustrate the above axioms with an example. We show that $\{x = 0 \wedge y = 0\}\ A\ \{y = \lceil z/3 \rceil\}$ where $A \equiv$ coroutine Q_1, Q_2 end is the coroutine given at the beginning of section 7. Our strategy in carrying out the proof will be to introduce auxiliary variables which distinguish the various "exit" and "resume" statements from each other and then use axiom C4 to delete these unnecessary variables as the last step of the proof. Axiom C2 enables us to apply the general exit assumption $\{P'\}$ exit $\{R'\}$ to a specific occurrence of an exit statement in Q_1. A similar comment applies to axiom C1.

We prove

```
{x = 0 ∧ y = 0}
        i := 0;
        j := 0;
        coroutine
            begin
                while x-y < z do
                    x := x+2;
                    y := y+2;
                    i := 1;
                    exit;
                    end;
                end
            begin
                while true do
                    y := y-1;
                    j := 1;
                    resume;
                    y := y-2;
                    j := 2;
                    resume;
                    end;
```

end;

$\{y = \lceil z/3 \rceil\}$

Choose $P = \{x = 0 \wedge y = 0 \wedge i = 0 \wedge j = 0\}$, $b = \{j = 0\}$

$R = \{y = \lceil z/3 \rceil\}$

$P' = \{(x=2 \wedge y=2 \wedge j=0) \vee (x=4y-8 \wedge j=1) \vee (x=4y-6 \wedge j=2)\}$

$R' = \{(x=4y-2 \wedge j=1) \vee (x=4y \wedge j=2)\}$

The invariant for the while loop of the first routine is

$INV_1 = \{(x=0 \wedge y=0 \wedge j=0) \vee (x=4y-2 \wedge x-y \leq z \wedge j=1) \vee$
$(x=4y \wedge x-y \leq z+1 \wedge j=2)\}$.

The invariant for the while loop of the second routine is $INV_2 \equiv \{(x=2 \wedge y=2 \wedge j=2) \vee (x=4y-6 \wedge j=2)\}$. It is easily checked using axioms C2-C4 together with the axioms for the assignment statement and the while statement that

$$\{P'\} \text{ exit } \{R'\} \vdash \{P \wedge b\}\ Q_1\ \{R\}$$

and

$$\{R'\} \text{ resume } \{P'\} \vdash \{P' \wedge b\}\ Q_2\ \{R'\}$$

The desired conclusion then follows by Axiom C1.

The technique of adding auxilliary variables is easily formalized (the pattern should be clear from the above example). Thus, in general, we are able to prove:

<u>8.1 Theorem:</u>

Consider the language described in sections 6 and 7 including the coroutine statement but requiring that procedures be nonrecursive. Let H'' be the Hoare-like axiom system consisting of axioms H1-H9 together with C1-C4. If T, L_A, L_Z, and I satisfy the conditions

A) T is a complete proof system for L_A and I and

B) L_A is expressive relative to L_E and I, then

$\models_I \{P\}\ A\ \{Q\} \Rightarrow \vdash_{H,T} \{P\}\ A\ \{Q\}$.

9. Coroutines and Recursion

We show that it is impossible to obtain a sound-complete system of Hoare-like axioms for a programming language allowing both coroutines and recursion provided that we do not assume a stronger type of expressibility than that defined in section 2. (We will argue in section 10 that the notion of expressibility introduced in section 2 is the natural one. We will examine the consequences of adopting a stronger definition of expressibility.)

9.1 Lemma

Consider the programming language described in section 7 with coroutines and recursive procedures, then the halting problem for programs in such a language is undecidable in all finite interpretations I with $|I| \geq 2$.

The proof of the lemma is similar to the proof of the main lemma of section 5; however, this time we reduce to the halting problem for a two stack machine rather than a queue machine. The simulation program will be a coroutine with one of its component routines controlling each of the two stacks. Each stack is represented by the successive activations of a recursive procedure local to one of the routines. Thus, stack entries are maintained by a variable local to the recursive procedure, deletion from a stack is equivalent to a procedure return, and additions to a stack are accomplished by recursive calls of the procedure. The simulation routine is given in outline form below:

```
prog_counter := 1;
Coroutine
     begin
          stack_1: proc(empty);
                new top, progress;
                progress := 1;
                while progress=1 do
                     if prog_counter = 1 then "INST_1" else
                     if prog_counter = 2 then "INST_2" else
                        .
                        .
                        .
                     if prog_counter = K then "INST_K" else null;
                     end;
                end stack_1;
          call stack_1 (1);
       end,
     begin
          stack_2: proc (empty);
               new top, progress;
               progress := 1;
               while progress=1 do
                    if prog_counter = 1 then "INST*_1" else
                    if prog_counter = 2 then "INST*_2" else
                       .
                       .
                       .
                    if prog_counter = K then "INST*_K" else null;
                    end;
               end stack_2;
          call stack_2 (1);
```

```
     end;
end;
```

where "$INST_1$",..."$INST_K$", "$INST_1^*$",..."$INST_K^*$" are encodings of the program for the two stack machine being simulated. Thus, for example, in the procedure STACK_1 we have the following cases:

(1) if $INST_j$ is PUSH X ON STACK_1, "$INST_j$" will be

```
        begin
             top:= x;
             prog_counter := prog_counter + 1;
             call stack_(0);
        end;
```

(2) if $INST_j$ is POP X FROM STACK_1, "$INST_j$" will be

```
        begin
             if empty = 1 then null;
             else begin
                  prog_counter :=prog_counter+1;
                  x:= top;
                  progress := 0;
                  end;
             end;
```

(3) if $INST_j$ is PUSH X ON STACK_2 or POP X FROM STACK_2, "$INST_j$" will simply be

```
        begin
             exit;
        end;
```

A similar encoding $INST_1^*, \ldots INST_k^*$ for the copy of the program within procedure stack_2 may be given. Statements of the form "prog_counter := prog_counter + 1" may be eliminated by introducing a fixed number of new variables to represent the binary representation of "prog-counter".

The remainder of the proof that it is impossible to obtain a sound-complete set of Hoare-like axioms for coroutines with recursive procedures is almost identical to the proof given in section 5 for recursive procedures with procedure parameters and will be omitted.

10. Discussion of Results

A number of problems are suggested by the above results. The incompleteness result of section 5 is for a block structured programming language with the following features:

(i) procedures as parameters of procedure calls

(ii) recursion

(iii) static scope

(iv) global variables

(b) internal procedures as parameters

(All of these features are found in Algol 60 [NA63] and, in fact, in PASCAL [WI73].) Are all of these features necessary for incompleteness? Recent results obtained by the author show that a sound and complete set of axioms may be obtained by modifying any one of the above features. Thus if we change from static scope to dynamic scope a complete set of axioms may be devised for a language with (i) procedures with procedure parameters, (ii) recursion, (iv) global variables, and (v) internal procedures as parameters; or if we disallow internal procedures as parameters, a complete system may be obtained for (i) procedures with procedure parameters, (ii) recursion, (iii) static scope, and (iv) global variables. A similar result holds if global variables are disallowed. An interesting open problem is whether a sound and complete system of axioms could be obtained if global variables were required to be <u>read only</u> instead of completely disallowed. Automata theoretic considerations suggest that the answer is yes.

In the case of coroutines and recursion the most important question seems to be whether a stronger form of expressibility might give completeness. The result of section 8 seems to require that any such notion of expressibility be powerful enough to allow assertions about the status of the run-time stack(s). Clint [CL73] suggests the use of stack-valued auxiliary variables to prove properties of coroutines which involve recursion. It seems likely that a notion of expressibility which allowed such variables would give completeness. However, the use of such auxiliary variables appears counter to the spirit of the axiomatic correctness. If a proof of a recursive program can involve the use of stack valued variables, why not simply replace the recursive procedures themselves by stack operations? The purpose of recursion in programming languages is to free the programmer from the details of implementing recursive constructs via stacks. Further evidence for the naturalness of the notion of expressibility that we have used is the fact that with either a) coroutines and non-recursive procedures or b) recursive procedures and no coroutines, we can obtain sound, complete sets of axioms under this notion of expressibility.

We note that the main application of procedures with procedure parameters is in numerical analysis (where one might wish to make the integrand a parameter of an integration procedure). Here, however, procedures are rarely recursive and hence can be handled by the techniques of section 3. Similarly coroutines are most frequently used in I/O routines which do not involve recursion and which can be handled by the methods of section 7 (if appropriate I/O axioms are introduced).

Finally we note that the technique of sections 5 and 9 may be applied to a number of other programming language features including a) call by name with functions and global variables, b) pointer variables with retention, c) pointer variables with recursion, and d) label variables with retention. All of these features appear to be inherently difficult to prove correct, and (one might argue) should be avoided in the design of programming languages suitable for program verification.

REFERENCES

[CL73] Clint, M. Program Proving: Coroutines. Acta Informatica, Vol. 2, pp. 50-63, 1973.

[CO75] Cook, S. A. Axiomatic and Interpretative Semantics for an Algol Fragment. Technical Report 79, Department of Computer Science, University of Toronto, 1975 (to be published in SCICOMP).

[DE73] deBakker, J. W. and L. G. L. Th. Meertens. On the Completeness of the Inductive Assertion Method. Mathematical Centre, December 1973.

[DI73] Dijkstra, E. W. A Simple Axiomatic Basis for Programming Language Constructs. Lecture notes from the International Summer School on Structured Programming and Programmed Structures, Munich, Germany, 1973.

[DO74] Donahue, James. Mathematical Semantics as a Complementary Definition for Axiomatically Defined Programming Language Constructs, in Donahue, et al., Three Approaches to Reliable Software: Language Design, Dyadic Specification, Complementary Semantics. Technical Report CSRG-45, Computer Systems Research Group, University of Toronto, December 1974.

[FL67] Floyd, R. W. Assigning Meaning to Programs in Schwartz, J. T., Ed. Mathematical Aspects of Computer Science Proc. Symposia in Applied Mathematics 19, pp. 19-32, Amer. Math. Soc., 1967.

[GO75] Gorelick, G. A Complete Axiomatic System for Proving Assertions about Recursive and Non-recursive Programs. Technical Report No. 75, Department of Computer Science, University of Toronto, January 1975.

[HO69] Hoare, C. A. R. An Axiomatic Approach to Computer Programming. CACM 12, 10 (October 1969), pp. 322-329.

[HO71] Hoare, C. A. R. Procedures and Parameters: An Axiomatic Approach. Symposium on Semantics of Algorithmic Languages, E. Engeler, Ed., Springer-Verlag, Berlin, pp. 102-116, 1971.

[HO74] Hoare, C. A. R. and P. E. Lauer. Consistent and Complementary Formal Theories of the Semantics of Programming Languages. Acta Informatica, Vol. 3, pp. 135-154, 1974.

[JO74] Jones, N. D. and S. S. Muchnick. Even Simple Programs are Hard to Analyze. TR-74-6, Department of Computer Science, The University of Kansas, November 1974 (to be published in JACM).

[MA70] Manna, Z. and A. Pnueli. Formalization of Properties of Functional Programs. JACM 17, No. 3, pp. 555-569, 1970.

[MC73] McGowan, Clement and Jayadeo Misra. A Mathematical Basis for Dijkstra-Hoare Semantics. Technical Report No. 73-73, Center for Computer and Information Sciences, Brown University, November 1973.

[OW76] Owicki, S. A Consistent and Complete Deductive System for the Verification of Parallel Programs. 8th Annual Symposium on Theory of Computing, 1976.

[WA76] Wand, M. A New Incompleteness Result for Hoare's System. 8th Annual Symposium on Theory of Computing, 1976.