PATHOLOGICAL INTERACTION OF PROGRAMMING LANGUAGE FEATURES

CS-1976-15

Edmund Melson Clarke, Jr.

Department of Computer Science
Duke University
Durham, N. C. 27706

September 1976

PATHOLOGICAL INTERACTION OF PROGRAMMING LANGUAGE FEATURES

1.1 Introduction

Drug interaction is a frequently encountered phenomenon in pharmacology. Two drugs, each of which has beneficial effects, may interact when administered together and create a serious pathological condition. In this paper we describe a similar phenomenon which may be observed in axiomatic treatments of programming language semantics. We show that there are combinations of programming language constructs for which it is impossible to to obtain a "good" axiom system, even though each construct by itself possesses a "good" axiom. Such pathological interactions are of obvious interest in the design of programming languages whose programs can be naturally proved correct.

In this paper we examine the interaction between recursion and parallelism at a very elementary level. Specifically, we consider a block structured programming language which allows both parameterless recursive procedures and coroutines. If procedures are not allowed to be recursive, there is a simple method for proving correctness of coroutines based on the addition of auxiliary variables (CL73, OW75). Likewise if the coroutine construct is disallowed, a "good" system of axioms for parameterless recursive procedures may be obtained (HO71, GO75, DO76). We prove, however, that if both recursive procedures and coroutines are allowed, then in a certain well defined sense it is impossible to obtain a good axiom system.

Additional combinations of programming language features for which it is impossible to obtain good systems of axioms are

discussed in Section 6.

1.2 Outline of Paper

In Section 2 we introduce S. Cook's notion of an expressible assertion language and specify precisely what is meant by the phrase "good Hoare-like axiom system." In Sections 3 and 4 good Hoare-like axiom systems are given for parameterless recursive procedures and for the coroutine construct. Section 5 contains the proof that it is impossible to obtain a good Hoare-like axiom system for a programming language which allows both recursive procedures and coroutines. The paper concludes with a discussion of the results and remaining open problems.

2.1 Good Hoare-like Axiom Systems

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

$$\frac{\{P \wedge b\}\ S\ \{P\}}{\{P\}\ \underline{\text{while}}\ b\ \underline{\text{do}}\ S\ \{P \wedge \sim b\}}$$

The axioms and inference rules are designed to capture the meanings of the individual statements of the programming language. Proofs

of correctness for programs are constructed by using these axioms together with a proof system for the assertion language.

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 theorem is indeed true. Another property is completeness (CO75), which means that every true statement is provable. From logic we know 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) Hoare-like axiom system there will be assertions {P} S {Q} which are true but not provable within the system. The question is whether this incompleteness reflects some inherent complexity of the programming language constructs or whether it is due entirely to the incompleteness of the assertion language.

How can we talk about the completeness of a Hoare-like axiom system independently of its assertion language? One way of answering this question was proposed by S. Cook (CO75). He gives a Hoare-like axiom system for a subset of Algol including the while statement and non-recursive procedures. He 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 recursive procedures.

A more detailed discussion of these ideas follows in Section 2.2.

2.2 Assertion Languages, Expression Languages, and Expressibility

As in (CO74) we distinquish two logical systems involved in 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 unquantified boolean expressions of conditionals and while statements are described. Both L_A and L_E are first order languages with equality, with L_A being an extension of L_E.

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. Once an interpretation I has been specified, meanings may be assigned to the variable free terms and closed formulas of L_A.

Let I be an interpretation with domain D. Meanings of programming language statements are specified by a meaning function $M = M_I$ which associates with a statement S, state s, and environment π a new state s'. The state gives the element of D associated with each variable name. The environment π indicates which procedure declarations are accessible.

Partial correctness assertions will have the form {P} A {Q}/E where S is a program statement, P and Q are formulas of L_A, and E is a set of procedure declarations.

2.2.1 Definition: {P} A {Q}/E is true with respect to I ($\models_I$ {P} S {Q}/E) iff $\forall s,s'\left[I\{P(s)\} = \text{true} \wedge M\{S\}(s,\pi) = s' \rightarrow I\{Q(s')\} = \text{true}\right]$ where π is the environment corresponding to E, i.e. $\pi(q)$ = "q:proc; K end" iff "q:proc; K end: $\in$ E.

To discuss the completeness of an axiom system independently of its assertion language we first introduce Cook's notion of expressibility.

2.2.2 Definition: L_A is expressive with respect to L_E and I iff for all statements S, environments π, and formulas Q in L_A there is a formula of L_A which expresses the weakest precondition $wp(S,\pi,Q)$ corresponding to S, π, and Q. (Formally $wp(S,\pi,Q)$ can be defined by:

$$wp(S,\pi,Q) = \{s \mid M\{S\}(s,\pi)\uparrow \text{ or } M\{S\}(s,\pi) \in Q\}.$$

If L_A is expressive with respect to L_E and I, then invariants of while loops and recursive procedures will be expressible by formulas of L_A. Although some choices of L_A, L_E, and I do not give expressibility, it is possible to argue that realistic choices for L_A, L_E, and I do give expressibility. If L_A and L_E 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_E and I. Also, if the domain of I is finite, expressibility is assured.

2.2.3 Lemma: If L_A, L_E are first order languages with equality and the domain of I is finite, then L_A is expressive with respect to L_E and I.

If H is a Hoare-like axiom system and T is a proof system for the assertion language L_A (relative to I), then a proof in the system (H,T) will consist of a sequence of partial correctness assertions $\{P\}\ S\ \{Q\}/E$ and formulas of L_A each of which is either an axiom (of H or T) or follows from previous formulas by a rule of inference (of H or T). If $\{P\}\ S\ \{Q\}/E$ occurs as a line in such

a proof, then we write $\vdash_{H,T} \{P\}\ S\ \{Q\}/E$.

2.2.4 Definition: A Hoare-like axiom system H for a programming language PL is sound and complete (in the sense of Cooke) iff for all L_A, L_E, and I, such that (a) L_A is expressive with respect to L_E and I and (b) T is a complete proof system for L_A with respect to I,

$$\models_I \{P\}\ S\ \{Q\}/E \iff \vdash_{H,T} \{P\}\ S\ \{Q\}/E.$$

3.1 Parameterless Recursive Procedures

A procedure declaration will have the form "q:proc; K end". A procedure call will have the form "call q". For simplicity we require that procedures be declared before they are used. We assume a semantics for procedure calls which is based on the copy rule of Algol 60, i.e. the execution of a procedure call results in the insertion of the procedure body at the point in the program where the call occurs. (Identifier conflicts caused by this substitution are eliminated by systematically changing the identifiers in effect at the point of the call). The above conventions may be made precise by introducing a formal operational specification of the semantics of procedures (CK76). An example of a procedure declaration is:

```
F:proc;
   If i=0 v i=1 then x:=x+1
   else begin new m; m:=i;
              i:=m-1; call F;
              i:=m-2; call F;
              end;
end
```

3.2 Axioms for Recursive Procedures

This section contains a "good" set of axioms for parameterless recursive procedures. The axioms are similar to those in (HO71), (GO75), and (DO76) except that the Algol 60 static scope rule is used for both variable identifiers and procedure identifiers (also see axioms A1-A8 in Appendix I).

The first axiom R1 is an induction axiom which allows proofs to be constructed using induction on depth of recursion.

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

provided that E does not contain a procedure "q:proc; K' end" which is different from "q:proc; K end".

Axioms R2 and R3 enable an induction hypothesis to be adapted to a specific procedure call. Before stating these axioms we define what it means for a variable to be inactive with respect to a procedure call.

3.2.1 Definition: Let the procedure q have declaration "q:proc; K end". A variable y is active with respect to q if y is either global to K or is active with respect to some global procedure called from within K. If y is not active with respect to "call q" then y is said to be inactive (with respect to that particular call). Simiarly, a term of the assertion language L_A is inactive if it contains only inactive variables. A substitution σ is inactive with respect to "call q" provided that it is a substitution of inactive terms for inactive variables.

R2. $$\frac{\{P\}\ \text{call q}\ \{Q\}/E}{\{P\sigma\}\ \text{call q}\ \{Q\sigma\}/E}$$

provided that the substitution σ is inactive with respect to q.

R3. $\dfrac{\{P\}\ \text{call q}\ \{Q\}/E}{\{P \wedge T\}\ \text{call q}\ \{Q \wedge T\}/E}$

provided that no variables which occur free in T are active in q.

We illustrate the axioms by proving

$\{x=0 \wedge i=i_0\}$ call F $\{x = f(i_0)\}/E$,

where E contains the declaration of the procedure F described in Section 3.1 and f(j) is the j^{th} Fibonacci number. In order to make the induction work, it is necessary to prove a slightly more general result; we prove:

$$\{x=x_0 \wedge i=i_0\} \text{ call F } \{x=x_0+f(i_0)\}/E. \qquad (1)$$

By axiom R1 it is sufficient to show that

```
{x=x0 ∧ i=i0}
if i=0 ∧ i=1 then x:=x+1
else begin new m; m:=i;                                    (2)
          i:=m-1; call R;
          i:=m-2; call R;
          end
{x=x0+f(i0)}/E
```

is provable if we first assume

$$\{x=x_0 \wedge i=i_0\} \text{ call R } \{x=x_0+f(i_0)\}/E. \qquad (3)$$

By the axiom for the conditional we must show

$$\{x=x_0 \wedge i=i_0 \wedge (i=0 \vee i=1)\}\ x:=x+1\ \{x:=x_0+f(i_0)\}/E \qquad (4)$$

and

```
{x=x0 ∧ i=i0 ∧ i>1}
begin new m; m:=i;
      i:=m-1; call R;                                      (5)
```

i:=m-2; call R;

end

$\{x=x_0+f(i_0)\}/E$

are provable. Step (4) follows from the fact that $f(0)=f(1)=1$. Step (5) will follow from the fact that $f(i_0)=f(i_0-1)+f(i_0-2)$ and the axioms for assignment, composition of statements and block structure, if we can first prove

$$\{x=x_0 \wedge i=i_0-1 \wedge m'=i_0\} \text{ call } R \{x=x_0+f(i_0-1) \wedge m'=i_0\}/E \tag{6}$$

and

$$\{x=x_0+f(i_0-1) \wedge i=i_0-2 \wedge m'=i_0\} \text{ call } R \{x=x_0+f(i_0-1)+f(i_0-2)\}/E. \tag{7}$$

To obtain (6) we use axioms R2 and R3. From the induction assumption (3) and axiom R2 with $\sigma=\frac{i_0-1}{i_0}$ we obtain

$$\{x=x_0 \wedge i=i_0-1\} \text{ call } R \{x=x_0+f(i_0-1)\}/E. \tag{8}$$

By axiom R3 with $T=\{m'=i_0\}$ we get

$$\{x=x_0 \wedge i=i_0-1 \wedge m'=i_0\} \text{ call } R \{x=x_0+f(i_0-1) \wedge m'=i_0\}/E.$$

A similar argument may be used to derive (7) and complete the proof.

Gorelick (GO75) generalizes the argument used above to apply to arbitrary recursive procedures. Although Gorelick's original proof implicitly assumes dynamic scope of variables, it is possible to modify the proof so that it applies the more common static scope rules assumed above. If L_R is the language described in Section 3.1 including parameterless recursive procedures and block structure with static scope, then:

3.2.2 Theorem: The Hoare-like axiom system H_R consisting of axioms A1-A8 together with R1-R3 is sound and complete (in the sense of Cook)

for proving assertions of the form {P} A {Q}/E where A is a program in L_R.

4.1 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 absolutely 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 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. Nesting of coroutine statements is not allowed.

A formal operational specification of the semantics for coroutines is given in (CK76). A simple example of a coroutine is:

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

This example illustrates the use of "exit" and "resume" statements within while loops. Note that if x and y are 1 initially, then the coroutine will terminate with $y=z^2$.

4.2 Axioms for Coroutines (Recursive procedures not allowed)

In this section we give a "good" set of axioms for coroutines[1] and describe a technique for proving correctness of coroutines which is based on the addition of "auxiliary variables". This technique was suggested to the author by Susan Owicki. It is different from the technique described by Clint (CL73), in that the auxiliary variables represent program counters (and therefore have bounded magnitude) rather than arbitrary stacks.

C1. (Coroutines)

$$\frac{\{P'\}\ \text{exit}\ \{R'\} \vdash \{P \wedge b\}\ Q_1\ \{R\} \qquad \{R'\}\ \text{resume}\ \{P'\} \vdash \{P' \wedge b\}\ Q_2\ \{R'\}}{\{P \wedge b\}\ \text{coroutine}\ Q_1, Q_2\ \text{end}\ \{R\}}$$

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

C2. (Exit)

$$\frac{\{P'\}\ \text{exit}\ \{R'\}}{\{P' \wedge C\}\ \text{exit}\ \{R' \wedge C\}}$$

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").

---

[1] Although there are serious problems in extending coroutine axioms to handle recursive procedures, non-recursive procedures may be handled in a straightforward manner. For simplicity, we assume in this section that all procedures have been disallowed. Thus, the environment component E is omitted from partial correctness assertions.

C3. (Resume)

$$\frac{\{R'\}\ \text{resume}\ \{P'\}}{\{R' \wedge C\}\ \text{resume}\ \{P' \wedge C\}}$$

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

C4. (Auxiliary variables)

Let AV be a set of variables such that $x \in AV \Rightarrow x$ appears in S' only in assignments y:=e with $y \in AV$. If P and Q are assertions which do not contain any free 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 is essentially the same as the auxiliary variable axiom in (OW76).)

We illustrate the axioms with an example. We show that $\{X=1 \wedge y=1 \wedge z=z_0\}\ A\ \{x=z_0^2\}$ where A ≡ "coroutine Q_1, Q_2 end" is the coroutine given in Section 4.1. Our strategy in carrying out the proof will be to introduce auxiliary variables to distinquish 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 adapt the general exit assumption {P'} exit {R'} to a specific occurrence of an exit statement in Q_1. A similar comment applies to axiom C3 for the resume statement. We prove

$\{x=1 \wedge y=1 \wedge z=z_0\}$

i:=0; j:=0;

coroutine

```
        while y≠z do
              y:=y+1; x:=x+1;
              i:=1; exit;
              end,
        while true do
              y:=y-2; j:=1; resume;
              y:=y+1; j:=2; resume;
              end
        end
```

Choose $P = \{x=1 \wedge y=1 \wedge z=z_0 \wedge i=0 \wedge j=0\}$

$b = \{j=0\}$

$R = \{x=z_0^2\}$

$P' = \{(x=3 \wedge y=2 \wedge j=0 \wedge y \leq z_0) \vee (x=y^2+2y+1 \wedge j=1 \wedge y<z_0-1) \vee (x=y^2-y+1 \wedge j=1 \wedge y \leq z_0)\}$

$R' = \{(x=y^2+3y+3 \wedge j=1 \wedge y \leq z_0-2) \vee (x=y^2 \wedge j=2 \wedge y \leq z_0)\}$

The invariant for the while loop of the first routine is

$INV_1 = \{(x=1 \wedge y=1 \wedge j=0 \wedge y \leq z_0) \vee (x=y^2+3y+3 \wedge j=1 \wedge y \leq z_0-2) \vee (x=y^2 \wedge j=2 \wedge y \leq z_0)\}$

The invariant for the while loop of the second routine is

$INV_2 = \{(x=3 \wedge y=2 \wedge j=0 \wedge y \leq z_0) \vee (x=y^2-y+1 \wedge j=2 \wedge y \leq z_0)$

By using axioms C2-C4 together with the axioms for the assignment statement and the while statement, it is possible to prove that

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

and

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

both hold. For example, to prove b) we assume $\{R'\}$ resume $\{P'\}$ and prove $\{P' \wedge b\}\ Q_2\ \{R'\}$. In order to prove $\{P' \wedge b\}\ Q_2\ \{R'\}$, we show that

c) $P' \wedge b \rightarrow INV_2$

d) $\{INV_2\}$

```
    while true do
      y:=y-2; j:=1; resume;
      y:=y+1; j:=2; resume;
      end
```

$\{INV_2 \wedge \sim true\}$

e) $INV_2 \wedge \sim true \rightarrow R'$ are true.

Steps c) and e) are easily verified. Step d) follows from the while axiom and the sequence of assertions below:

d1)	$\{INV_2 \wedge true\}$ y:y-2; j=1 $\{R' \wedge j=1\}$	A1 assignment
d2)	$\{R' \wedge j=1\}$ resume $\{P' \wedge j=1\}$	C3 resume
d3)	$\{P' \wedge j=1\}$ y:=y+1; j=2 $\{R' \wedge j=2\}$	A1 assignment
d4)	$\{R' \wedge j=2\}$ resume $\{P' \wedge j=2\}$	C3 resume
d5)	$P' \wedge j=2 \rightarrow INV_2$	arithmetic

Once a) and b) have been established, the desired conclusion follows immediately by axiom C1.

The technique of adding auxiliary variables is easily formalized. (The pattern should be clear from the above example.) If L_c is the programming language described in Section 4.1 including the coroutine statement but disallowing recursive procedures, then we can prove the following general theorem.

4.2.1 Theorem: The Hoare-like axiom system H_c consisting of axioms A1-A8 together with C1-C4, is sound and complete (in the sense of Cook) for proving assertions of the form $\{P\}$ A $\{Q\}$ where A is a program in L_c.

5.1 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 2.2. (We will argue in Section 6 that the notion of expressibility introduced in Section 2.2 is the natural one. We will also examine the consequences of adopting a stronger notion of expressibility.) Let $L_{c,r}$ be the programming language with the features described in Sections 3.1 and 4.1 including both recursive procedures and the coroutine statement.

5.1.1 Lemma: The Halting problem for programs in the language $L_{c,r}$ is undecidable for all finite interpretations I with $|I| \geq 2$.

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. This is not the case for a language which allows both recursive procedures and coroutines. We will show how to simulate a two stack machine by means of a program in the language $L_{c,r}$. Since the Halting problem is undecidable for two stack machines, the desired result will follow. 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 "top" local to the recursive procedure, deletion from a stack is equivalent to a

procedure return, and additions to a stack are accompolished by recursive calls of the procedure. The simulation routine is given in outline form below:

```
Prog_counter:=1;
Coroutine
     begin
          stack_1:proc;
               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
     end,
     begin
          stack_2:proc;
               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;
    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_1;
    end;
```

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

```
    begin
        prog_counter:=prog_counter+1;
        x:=top;
        progress:=0;
    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 $INST_1^*$,...$INST_K^*$ 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".

5.1.2 Theorem: It is impossible to obtain a system of Hoare-like axioms H for the programming language $L_{c,r}$ which is sound and complete in the sense of Cook.

Proof: Suppose that there were a sound, complete Hoare-like axiom system H for programs in $L_{c,r}$. Thus for all L_A, L_E, and I, if (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

$$\models_I \{P\}\ S\ \{Q\}/E \iff \vdash_{H,T} \{P\}\ S\ \{Q\}/E$$

This leads to a contradiction. Choose I to be a finite interpretation with $|I| \geq 2$. Observe that T may be chosen in a particularly simple manner; in fact there is a decision procedure for the truth of formulas in L_A relative to I. Note also that L_A is expressive relative to L_E and I. This was shown by Lemma 2.2.3, since I is finite. Thus both hypothesis (a) and (b) are satisfied. From the definition of partial correctness, we see that {true} S {false}/ϕ holds iff S diverges for the initial values of its global variables. By the lemma above, we conclude that the set of programs S such that $\models_I$ {true} S {false}/ϕ holds is not recursively enumerable. On the other hand, since

$$\models_I \{true\}\ S\ \{false\}\ S\ \{false\}/\phi \iff \vdash_{H,T} \{true\}\ S\ \{false\}/\phi$$

we can enumberate those programs S such that $\vdash_I$ {true} S {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.

6.1 Discussion of Results and Open Problems

Theorem 5.1.2 shows how two well behaved programming language constructs may interact so that it is impossible to obtain an axiom system for the resulting language which is both sound and complete. A natural question is whether a more powerful notion of expressibility might give completeness. In the case of coroutines and recursion, the result of section 5.1 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 high level programming languages. If a proof of a recursive procedure 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.

Finally we note that the technique of Section 5.1 may be applied to a number of other programming language features including (a) procedures with procedure parameters if we allow global variables, internal procedures as parameters, and recursion, (b) call by name parameter passing with functions and global variables, (c) unrestricted 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

(CK76) Clarke, Jr., E.M. Programming Language Constructs for Which it is Impossible to Obtain Good Hoare-like Axioms. Technical Report No. 76-287, Computer Science Department, Cornell University, August 1976.

(CL75) 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, Computer Science Department, 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.

(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.

(GO75) Gorelick, G. A Complete Axiomatic System for Proving Assertions about Recursive and Non-recursive Programs. Technical Report No. 75, Computer Science Department, 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, Computer Science Department, University of Kansas, November 1974 (to be published in JACM).

(MA70) Manna, Z. and A. Pneuli. Formalization of Properties of Functional Programs. JACM 17, No. 3, pp. 555-569, 1970.

(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.

APPENDIX

Basic Set of Axioms (Axioms for Block Structure with Static Scope, etc.)

A1. $$\frac{\{U \wedge x^i = e_0\} \text{ begin } A \frac{x^i}{x} \text{ end } \{V\}/E}{\{U\} \text{ begin new } x;\ A \text{ end } \{V\}/E}$$

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

A2. $$\frac{\{U\} \text{ begin } A \frac{q^i}{q} \text{ end } \{V\}/E \cup \{q^i : \text{proc};\ K \frac{q^i}{q} \text{ end}\}}{\{U\} \text{ begin } q:\text{proc};\ K \text{ end};\ A;\ \text{end } \{V\}/E}$$

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

A3. $$\frac{\{U\}\ A\ \{V\}/E_1}{\{U\}\ A\ \{V\}/E_2}$$

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

A4. (a) $$\frac{\{U\}\ A\ \{V\}/E}{\{U\} \text{ begin } A \text{ end } \{V\}/E}$$

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

A5-A8. Usual axioms for assignment conditional, while and consequence (see (HO69)). Note of course that each of these axioms must be modified to make explicit the set E of procedure declarations.