PROGRAMMING LANGUAGE CONSTRUCTS FOR WHICH IT IS IMPOSSIBLE TO OBTAIN GOOD HOARE AXIOM SYSTEMS

CS-1976-17.1 (revised)
February 9, 1978

Edmund Melson Clarke, Jr.

This is a revised version of
Duke University Technical
Report CS-1976-17

PROGRAMMING LANGUAGE CONSTRUCTS FOR WHICH IT IS IMPOSSIBLE TO OBTAIN GOOD HOARE AXIOM SYSTEMS

Edmund Melson Clarke, Jr.†

Duke University, Durham, North Carolina

Abstract. Hoare axiom systems for establishing partial correctness of programs may fail to be complete because of (a) incompleteness of the assertion language relative to the underlying interpretation or (b) inability of the assertion language to express the invariants of loops. S. Cook has shown that if there is a complete proof system for the assertion language (i.e. all true statements of the assertion language) and if the assertion language satisfies natural expressibility condition then a sound and complete axiom system for a large subset of Algol may be devised. We exhibit programming language constructs for which it is impossible to obtain sound and complete sets of Hoare axioms even in this special sense of Cook's. These constructs include (i) recursive procedures with procedure parameters in a programming language which uses static scope of identifiers and (ii) coroutines in a language which allows parameterless recursive procedures. Modifications of these constructs for which sound and complete systems of axioms may be obtained are also discussed.

Keywords and Phrases: Hoare axioms, soundness, relative completeness, procedure parameters, coroutines

CR Categories: 5.24, 5.27, 4.29 (language design)

†A large portion of this research was completed while the author was a graduate student at Cornell University with the support of an IBM Research Fellowship.

1.1 Background

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 [12]. 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

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

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 axiom system? One property a good system should have is soundness ([10],[6]). A deduction system is sound iff every theorem is actually true. Another property is completeness [4], which means that every true statement is provable. From the Gödel incompleteness theorem we see 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 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. For example, when dealing with the integers, 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 axiom system independently of its assertion language?

One way of answering this question was proposed by S. Cook [4]. Cook gives a Hoare axiom system for a subset of Algol including the while statement and nonrecursive procedures. He then proves that if there is a complete proof system for the assertion language (i.e. all true statements of the assertion language) and if the assertion language satisfies a natural expressibility condition, then every true partial correctness assertion will be provable. Gorelick [7] extends Cook's work to recursive procedures. Similar completeness results are given by deBakker and Meertens [5] and by Manna [13].

1.2 New Results of This Paper

Modern programming languages provide constructs which are considerably more complicated than the while statement, and 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 a good (i.e. sound and complete) Hoare axiom system. This question is of obvious importance in the design of programming languages whose programs can be naturally proved correct.

We first consider the problem of obtaining a sound and complete system of axioms for an Algol-like programming language which allows precedure names as

parameters in procedure calls. We prove that in general it is impossible to obtain such a system of axioms even if we disallow calls of the form "Call P(...,P,...)". (Calls of this form are necessary to directly simulate the lambda calculus by parameter passing.) We then consider restrictions to the programming language which allow one to obtain a good axiom system.

The incompleteness result is obtained for a block-structured programming language with the following features:

(i) procedure names as parameters of procedure calls

(ii) recursion

(iii) static scope

(iv) global variables

(v) internal procedures

All these featues are found in Algol 60 [14] and in Pascal [16]. We also show that a sound and complete axiom system can be obtained by modifying any one of the above features. Thus if we change form static scope to dynamic scope, a complete set of axioms may be obtained for (i) procedures with procedure parameters, (ii) recursion, (iv) global variables, and (v) internal procedures; or if we disallow internal procedures, a complete system may be obtained for (i) procedures with procedure parameters, (ii) recursion, (iii) static scope, and (iv) global variables. As far as we know, this is the first axiomatic treatment of procedure parameters.

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 [15]. If, however,

procedures are recursive, no such simple method can give completeness. These observations generalize to languages with parallelism and recursion.

Additional programming language constructs for which it is impossible to obtain good axioms are discussed in Section 8.

1.3 Outline of Paper

The development of these results is divided into two parts--the first dealing with procedures as parameters and the second with the coroutine construct. In Section 2 a formal description is given for a programming language with static scope, global variables, and procedures with procedure parameters. This is followed by a discussion of Cook's expressibility condition. Modifications necessary to handle dynamic scope are also discussed. In Section 3 we prove that it is impossible to obtain a sound and complete axiom system for this language. In Sections 4, 5, and 6 we discuss restrictions sufficient to insure that good Hoare axioms can be found. Sections 8 and 9 are devoted to completeness and incompleteness results for the coroutine construct and follow the same outline as was used in the first part of the paper. The paper concludes with a discussion of the results and remaining open problems.

2. A Simple Programming Language and its Semantics.

As in [4] 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 specified and the expression language L_E in which the terms forming the right hand sides of assignment statements and (quantifier-free) boolean expressions of conditionals and while statements are specified.

Both L_A and L_E are first order languages with equality and L_A is an extention of L_E. The variables of L_E are called program identifiers (PROG_ID) and are 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), an assignment of functions on D to the function symbols of L_A and an assignment of predicates on D to the predicate symbols of L_A. We will use the notation $|I|$ for 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).

Let I be an interpretation with domain D. A program state is an ordered list of pairs of the form:

$$(v_1.d_1)\ (v_2.d_2)\ \ldots\ (v_n.d_n)$$

where each v_i is a variable identifier and each d_i is an element of D. Thus a program state is similar to the association list used in the definition of Lisp. If s is a program state and v is a variable identifer than s(v) is the value associated with the first occurrence of v in s. Similarly, ADD (s,v,d) is the program state obtained by adding the pair (v.d) to the head of list s, and DROP(s,v) is the program state obtained from s by deleting the first pair which contains v. VAR(s) is the set of all variable identifiers appearing in s.

If t is a term of L_A with variables $x_1, x_2, \ldots, x_n$ and s is a program state, then we will use the notation t(s) to mean

$$t\ \frac{s(x_1),\ldots,s(x_n)}{x_1,\ldots,x_n}$$

i.e. the term obtained by simultaneous substitution of $s(x_1)$ for x_1, ..., $s(x_n)$ for x_n.

Likewise we may define P(s) where P is a formula of L_A. It is frequently convenient to identify a formula P with the set of all program states which make P true, i.e. with the set $\{s|I[P(s)]=true\}$. If this identification is made, then false will correspond to the empty state set and true will correspond to the set of all program states.

We consider a simple programming language which allows assignment, procedures calls, while, compound and block statements. Procedure declarations have the form "proc $q(\bar{x}:\bar{p})$; $K<\bar{x},\bar{p}>$ end" where q is the name of the procedure, $\bar{x}$ is the list of formal variable parameters, $\bar{p}$ is the list of formal procedure parameters, and $K<\bar{x},\bar{p}>$ is a statement involving the parameters $\bar{x}$ and $\bar{p}$. A procedure call has the form "call $q(\bar{a}:\bar{P})$" where $\bar{a}$ is the list of actual variable parameters and $\bar{P}$ is the list of actual procedure parameters. To simplify the treatment of parameters we restrict the entries in $\bar{a}$ to be simple program identifiers. We further require that procedure names be declared before they appear in procedure calls. An environment e is a finite set of procedure declarations which does not contain two different declarations with the same name. If π is a procedure declaration, then $ADD[e,\pi]$ is the environment obtained from e by first deleting all procedure declarations which have the same name as π, and then adding π.

Meanings of statements are specified by a meaning function $M=M_I$ which associates with statement S, state s, and environment e a new state s'. Intuitively s' is the state resulting if S is executed with initial state s and initial environment e. The definition of M is given operationally in a rather non-standard manner which makes extensive use of renaming. This type of

definition allows static scope of identifiers without the introduction of closures to handle procedures. The definition of M[S](e,s) is by cases on S:

(1) S is "begin new x; B<x> end" $\longrightarrow$ DROP(M[begin B<x^i> end](e,s'),x^i) where i is the index of the first program identifier not appearing in S, e, or VAR(s) and s'=ADD(s,x^i,a_o). (a_o is a special domain element which is used as the initial value of program identifiers.)

(2) S is "begin proc q($\bar{x}$:$\bar{p}$); K<$\bar{x}$,$\bar{p}$,q> end; B<q> end"$\longrightarrow$M[begin B<q^i> end](e',s) where i is the index of the first procedure identifier not occurring in B<q> or e and e' = ADD(e, "proc q^i($\bar{x}$:$\bar{p}$); K<$\bar{x}$,$\bar{p}$,q^i> end").

(3) S is "begin B_1; B_2 end"$\longrightarrow$ M[begin B_2 end](e,M[B_1](e,s))

(4) S is "begin end"$\longrightarrow$s

(5) S is "x:=t"$\longrightarrow$s' where s'=ADD(DROP(s,x), x, I[t(s)])

(6) (conditional) S is "b $\rightarrow B_1,B_2$"$\longrightarrow$ $\begin{cases} M[B_1](e,s) & \text{if } s \in b \\ M[B_2](e,s) & \text{otherwise} \end{cases}$

(7) (while) S is "b*B"$\longrightarrow$ $\begin{cases} M[b*B](e,M[B](e,s)) & \text{if } s \in b \\ s & \text{otherwise} \end{cases}$

(8) S is "call q($\bar{a}$;$\bar{P}$)"$\longrightarrow$ $\begin{cases} M[K<\bar{a},\bar{P}>](e,s) & \text{if "proc q}(\bar{x}:\bar{p}); \text{ K}<\bar{x},\bar{p}> \text{ end" } \in e, \\ & \quad \text{length } (\bar{a})=\text{length}(\bar{x}), \text{ and length}(\bar{p}) \\ & \quad =\text{length } (\bar{P}), \\ \text{undefined} & \text{otherwise} \end{cases}$

Sometimes it will be easier to work with computation sequences than with the definition of M directly. A computation sequence C of the form

$$C \equiv (S_0,e_0,s_0)\ldots(S_i,e_i,s_i)\ldots$$

gives the statement, environment and program state during the i^{th} step in the computation of $M[S_0](e_0,s_0)$. Since the rules for generating computation sequences may be obtained in a straightforward manner form the definition of M, they will not be included here,

The meaning function M may be easily modified to give dynamic scope of identifiers. With dynamic scope when an identifier is referenced, the most recently declared active copy of the identifier is used. This will occur with our model if we omit the renaming of variables which is used in clauses (1) and (2) in the definition of M. Thus, for example,
M[begin new x; B end](e,s)=DROP(M[begin B end](e,s'),x) where $s'=ADD(s,x,a_0)$. Unless explicitly stated we will always assume static scope of identifiers in this paper.

Partial correctness assertions will have form {P} S {Q}/e where S is a program statement, P and Q are formulas of L_A, and e is an environment.

2.1 Definition

{P} S {Q}/e is true with respect to I ($\models_I$ {P} S {Q}/e) iff
$\forall s,s'[s\varepsilon P \wedge M[S](e,s)=s' \longrightarrow s'\varepsilon Q]$ and every procedure which is global to S or to some procedure declaration in e is contained in e. If Γ is a set of partial correctness assertions and every assertion in Γ is true with respect to I, then we write $\models_I \Gamma$.

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

2.2 Definition

L_A is expressive with respect to L_E and I iff for all S, Q, e there is a

formula of L_A which expresses the weakest precondition for partial correctness $wp(S,e,Q)=\{s \mid M[S](e,s)$ is undefined or $M[S](e,s) \in Q\}$. (Note that we could have alternatively used the strongest postcondition $SP(S,e,P)=\{M[S](e,s) \mid s \in P\}$.)

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. Not every choice of L_A, L_E, and I gives expressibility. Cook demostrates this in the case where the assertion and expression languages are both the language of Presburger Arithmetic. Wand [18] gives another example of the same phenomenon. More realistic choices of 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.

2.3 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.

Proof. Let D be the domain of I and suppose that $|D|<\infty$. Let S be a statement, e an environment, and Q a formula of L_A. Suppose that $x_1,..,x_n$ are the variables which occur free in Q, global to S, or global to some procedure in e. Since D is finite, there exists a finite set of n-tuple $\Gamma = \{(a_1^j,\ldots,a_n^j) \mid 1 \le j \le m\}$ such that $s \in wp(S,e,Q)$ iff for some n-tuple $(a_1^j,\ldots,a_n^j)$ in Γ we have $s(x_i) = a_i^j$ for $1 \le i \le n$. If $R = \bigvee_{1\le j\le m} x_1 = a_1^j \wedge x_2 = a_2^j \wedge \ldots \wedge x_n = a_n^j$, then it is not difficult to show that R expresses $wp(S,e,Q)$.

If H is a Hoare 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} A {Q}/e occurs as a line in such a proof, then we write $\vdash_{H,T}\{P\}\ S\ \{Q\}/e$. In a similar manner, we may define $\Gamma \vdash_{H,T} \Delta$ where Γ and Δ are sets of partial correctness assertions.

2.2 Definition

A Hoare axiom system H for a programming language PL is sound and complete (in the sense of Cook) iff for all T, 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,

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

3. Recursive Procedures with Procedure Parameters

In this section we prove:

3.1 Theorem

It is impossible to obtain a system of Hoare axioms H which is sound and complete in the sense of Cook for a programming language which allows:

(i) procedures as parameters of procedure calls

(ii) recursion

(iii) static scope

(iv) global variables

(v) internal procedures

Remark: In section 4 show that it is possible to obtain a sound, complete system of Hoare axioms by modifying any one of the above features. To obtain the incompleteness result, only procedure identifiers are needed as parameters of procedure calls. The completeness proof allows, in addition, variable parameters which are passed by direct syntactic substitution.

In order to prove the theorem we need the following lemma.

3.2 Lemma

The Halting Problem is undecidable for programs in a programming language with features (i) - (v) above all finite interpretations I with $|I| \geq 2$.

The proof of the lemma uses a modification of a result of Jones and Muchnick [11]. Note that the lemma is not true for flowchart schemes or while schemes. 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 whether any program state is repeated. In the case of recursion one might expect that the program could be viewed as a type of pushdown automaton (for which the Halting Problem is decidable). This is not the case if we allow procedures as parameters. The static scope execution rule, which states that procedure calls are elaborated in the environment of the procedure's declaration rather than in the environment of the procedure call, allows the simulation program to access values normally buried in the runtime stack without first "popping the top" of the stack.

Formally we show that it is possible to simulate a queue machine which has three types of instructions 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 go to L--conditional branch. Since the Halting Problem for queue machines is undecidable, the desired result follows.

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 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

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

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

```
proc sim(:back);
      begin new top, bottom, progress;

            <declaration of local procedure up>

            progress:=1;
            while progress=1  do
             begin
               if prog_counter=1 then "INST_1" else
               if prog_counter=2 then "INST_2" else
                                  :
                                  :
               if prog_counter=K then "INST_k" else progress:=0
             end
            end
      end
end sim;
prog_counter:=1;
empty_queue:=1;
call sim(:loop)
```

The variable "empty_queue" tells whether the queue contains any elements. "prog_counter" is the instruction counter for the program being simulated. If the size the queue program is greater than the number of elements in the domain of the interpretation, then "prog_counter" may be replaced by a fixed number of new variables which hold its binary representation. "progress" is used to indicate when control should be returned to the previous activation of the procedure "sim". The procedure "loop" diverges for all values of its parameters; it will be called when an attempt is made to remove an entry from the empty queue. Declarations for "empty_queue", "prog_counter", "progress", "loop" and the program variables for the queue machine are omitted from the outline of the simulation program.

The appropriate encoding for queue machine instructions is given by cases:

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

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

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

```
begin
   if empty_queue ≠ 1 then top:=A;
   else begin top:=-A;
              empty_queue:=0
        end
   prog_counter;=prog_counter+1;
   call sim(:up);
   progress:=0
end
```

Note that we are assuming that the first instruction in any queue program will be an "enqueue" instruction. Note also that if "progress" ever becomes

0, the simulation program will eventually terminate.

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

```
begin
   if empty_queue=1 then call loop( );
   call back (x, bottom;);
   If bottom=1 then empty_queue:=1;
   x:=-x;
   prog_counter:=prog_counter+1
end
```

If the queue is not empty, "back" will correspond to the local procedure "up" declared in the previous activation of "sim". On return from the call on "back" the first parameter x will contain the value of "top" in the first activation of "sim". The second parameter of "back" ("up") is only used when "back" is called from within "up".

Finally, we must describe the procedure "up" which is used by "sim" in determining the value of the first element in the queue and deleting that element:

```
proc up (front_of_queue, first:)
     if top < 0
     then begin
            front_of_queue:=top;
            first:=1
          end
     else begin
            call back (front_of_queue, first:);
            If first=1 then begin top;=-top;
                                   first:=0
                            end
          end
end up;
```

After a call on "up", the parameter "front_of_queue" will contain the value of "top" in the first activation of "sim". The parameter "first" is used in marking the queue element which will henceforth be first in the queue.

This completes the description of the simulation program. Contour diagrams [17] describing the simulation of the queue program "enqueue 5; dequeue X" are given in figures 1 and 2. We now return to the proof of the incompleteness theorem. Suppose that there were a sound, complete Hoare axiom system H for programs of the type described at the beginning of this section. 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 condradiction. 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 to L_E and I; this was shown by the lemma in Section 2 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

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

we can enumerate those programs S such that $\models_I$ {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.

4.1 Completeness Results

A major source of complexity in languages which allow procedure parameters is self-application e.g. calls of the form "call P(...P...)". If self-application is allowed, the lambda calculus may be directly simulated by parameter passing. The reader will note, however, that the incompleteness result of section 3 holds even if self-application is not allowed. In restricting the programming language so that a sound and complete axiom system may be obtained, we will disallow self-application. This restriction may be enforced by requiring that actual procedure parameters be either formal procedure parameters or names of procedures with no procedure formal parameters.

A second source of complexity associated with parameter passing is sharing. Sharing occurs when some variable in a program may be referenced by two different names. (A formal treatment of sharing is given in [6]). The incompleteness result of section 3 may also be obtained if sharing is not allowed. We will assume in the remainder of the paper that sharing is not allowed; we will require that whenever a procedure call of the form "call $q(\bar{a}:\bar{P})$" is executed in environment e, all of the variables in $\bar{a}$ are distinct and no parameter in $\bar{a}$ is global to the declaration of q or to any procedure in e which may be activated indirectly by the call on q.

Once sharing and self-application have been disallowed a "good" axiom system may be obtained by modifying any one of the five features of theorem 3.1. These results are sumarized in figure 3. In order to establish the completeness results of figure 3, sound and complete axiom systems must be given for languages (2)-(6). Due to space limitations, we will only consider

language 5 in this paper. Languages 2 and 3 are treated in [2]. Good axiom systems for languages 4 and 6 are similar to the axiom system described in section 4.3 and will not be discussed here.

4.2 The Range of a Statement

Consider the following program segment:

```
proc F(y:p);
     If y>1
     then begin y:=y-2; call p(y:F) end
     else y:=0
     end F;
proc G(w:q); z:=z+w; call q(w:G) end G;
call F(x:G);
```

Observe that the only procedure calls which can occur during the execution of the program segment are "call F(x:G)" and "call G(x:F)". In general, let S_0 be a statement and e_0 an environment; the range of S_0 with respect to e_0 is the set of pairs $\langle$call $q_i(\bar{a}:\bar{P}), e_i\rangle$ for which there is a valid computation sequence of the form:

$$(S_0,e_0,s_0),\ldots,(\text{call } q_i(\bar{a}:\bar{P}),e_i,s_i),\ldots$$

If static scope of identifiers is used, the range of a statement S_0 with respect to environment e_0 may be infinite. This is because of the renaming at block entry which occurs in clauses (1) and (2) in the definition of M. If, however, dynamic scope is used, then the range of a statement (with respect to a particular environment) must be finite; in fact, there is a simple algorithm for computing the range of a statement. The range of S with respect environment e is given by RANGE (S,e,ϕ) where the definition of RANGE(S,e,π) is given by cases on S:

(1) S≡begin new x; A end"$\longrightarrow$ RANGE(begin A end,e,π)

(2) S≡"begin proc q($\bar{y}$:$\bar{r}$); L end; A end"$\longrightarrow$ RANGE(begin A end, e',π)

where e' = ADD(e, proc q($\bar{y}$:$\bar{r}$); L end)

(3) $S \equiv$ "begin A_1; A_2 end" $\longrightarrow$ RANGE(begin A_2 end, e, RANGE(A_1,e,π))

(4) S="begin end" $\longrightarrow \pi$

(5) S="z:=e" $\longrightarrow \pi$

(6) S="b$\longrightarrow A_1, A_2$" $\longrightarrow$ RANGE(A_2,e,RANGE(A_1,e,π))

(7) S="b*A" $\longrightarrow$ RANGE(A,e,π)

(8) S="call q($\bar{a}$:$\bar{P}$)" $\longrightarrow$
$$\begin{cases} \pi & \text{if } \langle \text{call } q(\bar{a}:\bar{P}), e\rangle \in \pi. \\ \text{RANGE}(K\langle \bar{a}, \bar{P}\rangle, e, \pi') & \text{where } \pi' = \pi \cup \{\langle \text{call } q(\bar{a}:\bar{P}), e\rangle\} \\ & \text{and "proc } q(\bar{x}:\bar{p}); K\langle \bar{x}, \bar{p}\rangle \text{end"} \in e, \\ & \text{otherwise.} \end{cases}$$

This same property of dynamic scope provides a simple algorithm for determining whether the execution of a statement S in environment e will result in sharing.

4.3 Good Axioms for Dynamic Scope

The axioms and rules of inference in the proof system DS for language 5 (dynamic scope of identifiers) may be grouped into three classes: axioms for block structure B1-B3, axioms for recursive procedures with procedure parameters R1-R6, and standard axioms for assignment, conditional, while, and consequence H1-H4.

Axioms for Block Structure:

(B1) $$\frac{\{U \underline{x}^{i}_{x} \wedge x = a_0\} \text{ begin A end } \{V \underline{x}^{i}_{x}\}/e}{\{U\} \text{ begin new x; A end } \{V\}/e}$$

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

(B2a) $$\frac{\{U\}\ \text{begin A end}\ \{V\}/e \cup \{\text{proc } q(\bar{x}:\bar{p});\ K \text{ end}\}}{\{U\}\ \text{begin proc } q(\bar{x}:\bar{p});\ K \text{ end; A end } \{V\}/e}$$

(B2b) $$\frac{\{U\}\ A\ \{V\}/e_1}{\{U\}\ A\ \{V\}/e_2}$$

provided that $e_1 \subseteq e_2$ and e_2 does not contain the declaration of two different procedures with the same name.

(B3a) $$\frac{\{U\}\ A\ \{V\}/e}{\{U\}\ \text{begin A end}\ \{V\}/e}$$

(B3b) $$\frac{\{U\}\ A_1\ \{V\}/e,\ \{V\}\ \text{begin } A_2 \text{ end } \{W\}/e}{\{U\}\ \text{begin } A_1;\ A_2 \text{ end } \{W\}/e}$$

Axioms for Recursive Procedures with Procedure Parameters:

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

(R1) $$\frac{\{U_0\}\text{call } F_0(\bar{x}_0:\bar{P}_0)\{V_0\}/e_0,\ldots,\{U_n\}\ \text{call } F_n(\bar{x}_n:\bar{P}_n)\{V_n\}/e_n \quad \vdash\ \{U_0\}K_0\langle\bar{P}_0\rangle\{V_0\}/e_0,\ldots,\{U_n\}K_n\langle\bar{P}_n\rangle\{V_n\}/e_n}{\{U_0\}\text{call } F_0(\bar{x}_0:\bar{P}_0)\{V_0\}/e_0,\ldots,\{U_n\}\text{call } F_n(\bar{x}_n:\bar{P}_n)\{V_n\}/e_n}$$

where "proc $F_i(\bar{x}_i:\bar{p}_i)$; $K_i\langle\bar{p}_i\rangle$ end" ε e_i for $0 \le i \le n$.

Axioms R2-R6 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

4.3.1 Definition: Let procedure q have declaration "proc $q(\bar{x};\bar{p})$; $K<\bar{x},\bar{p}>$end". A variable y is active with respect to "call $q(\bar{a};\bar{P})$" in environment e if y is either global to $K<\bar{a},\bar{P}>$ or is active with respect to a call on a procedure in e from within $K<\bar{a},\bar{P}>$. If y is not active with respect to "call $q(\bar{a};\bar{P})$" then y is said to be inactive (with respect to the particular call). Similarly a term of the assertion language is inactive if it contains only inactive variables. A substitution σ is inactive with respect to "call $q(\bar{a};\bar{P})$" provided that it is a substitution of inactive terms for inactive variables.

(R2) $$\frac{\{U\}\ \text{call}\ q(\bar{a}:\bar{P})\{V\}/e}{\{U\sigma\}\ \text{call}\ q(\bar{a}:\bar{P})\ \{V\sigma\}\ /e}$$

provided σ is inactive with respect to "call $q(\bar{a}:\bar{P})$" and e.

(R3) $$\frac{\{U(r_0)\}\ \text{call}\ q(\bar{a}:\bar{P})\ \{V(r_0)\}\ /e}{\{\exists r_0\ U(r_0)\}\ \text{call}\ q(\bar{a}:\bar{P})\ \{\exists r_0\ V(r_0)\}/e}$$

provided that r_0 is inactive with respect to "call $q(\bar{a}:\bar{P})$" and e.

(R4) $$\frac{\{U\}\ \text{call}\ q(\bar{a}:\bar{P})\ \{V\}/e}{\{U \wedge T\}\ \text{call}\ q(\bar{a}:\bar{P})\{V \wedge T\}/e}$$

provided that no variable which occurs free in T is active in "call $q(\bar{a}:\bar{P})$".

(R5) $$\frac{\{U\}\ \text{call}\ q(\bar{x}:\bar{P})\ \{V\}/e}{\{U\frac{\bar{a}}{\bar{x}}\}\text{call}\ q(\bar{a}:\bar{P})\ \{V\frac{\bar{a}}{\bar{x}}\}/e}$$

provided that no variable free in U or V occurs in $\bar{a}$ but not in the corresponding position of $\bar{x}$. ($\bar{x}$ is the list of formal parameters of q. This axiom will not be sound if sharing is allowed.)

Since procedures are allowed as parameters of procedure calls, it is possible for the execution of a syntactically correct statement to result in a procedure call with the wrong number of actual parameters. If dynamic scope of identifiers is used, this eventuality may be handled by the following axiom:

(R6) {true} call $q(\bar{a}:\bar{P})$ {false} / {proc $q(\bar{x}:\bar{p})$; K end}

provided that length($\bar{a}$) $\neq$ length($\bar{x}$) or length($\bar{P}$) $\neq$ length($\bar{p}$).

Standard Axioms for Assignment, Conditional, While and Consequence. These axioms (H1-H4) are widely discussed in the literature and will not be stated here.

We illustrate the use of the above axioms by two examples. The first example illustrates dynamic scope of identifiers. The second example shows how procedure parameters may be handled.

Example 1: We prove

```
{true}
begin new x;
     proc q; z:=x  end;
     x:=1;
     begin new x; x:=2; call q end
end;
{z=2}/ϕ
```

Let e be the enviroment {proc q; z:=x end};

(1) $\{x=2 \wedge y=1\}$ z:=x $\{z=2\}/\phi$ H1

(2) $\{x=2 \wedge y=1\}$ call q $\{z=2\}/e$ R1

(3) {y=1} begin x;=2; call q end {z=2}/e H1, B3

(4) {x=1} begin new x; x:=2; call q end {z=2}/e B1

(5) {true}
begin x:=1;
begin new x; x:=2; call q end
end
{z=2}/e H1, B3

(6) {true}
begin new x;
proc q; z:=x end;
x:=1;
begin new x; x:=2; call q end
end
{z=2}/ϕ B1, B2

Note that if static scope were used instead of dynamic scope, the correct postcondition would be {z=1}.

Example 2: We prove

$\{x=2x_0+1 \wedge z=0\}$
proc F(y:p);
If y>1
then begin y:=y-2; call p(y:F) end
else y:=0
end F;
proc G(w:q); z:=z+w; call q(w:G) end G;
call F(x:G)
$\{z=x_0^2\}/\phi$

Let e be the enviroment containing the declarations of F and G. Let K_1<p> and K_2<q> be the bodies of procedures F and G respectively. Since the range of "call F(x;G)" with respect to e consists of <call G(x:F), e> and <call F(x:G), e> it is sufficient to determine the effects of "call G(x:F)" and "call F(x:G)" when executed in environment e.

We assume:

(1) $\{y=2y_0+1 \wedge z=z_0\}$ call F(y:G) $\{z=z_0+y_0^2\}$ /e

and

(2) $\{w=2w_0+1 \wedge z=z_0\}$ call G(w:F) $\{z=z_0+(w_0+1)^2\}/e$.

Using these assumptions it is straighforward to prove:

(3) $\{y=2y_0+1 \wedge z=z_0\}\ K_1\langle G\rangle\ \{z=z_0+y_0^2\}/e$

and

(4) $\{w=2w_0+1 \wedge z=z_0\}\ K_2\langle F\rangle\ \{z=z_0+(w_0+1)^2\}/e$

By axiom R1, we obtain

(5) $\vdash \{y=2y_0+1 \wedge z=z_0\}$ call F(y:G) $\{z=z_0+y_0^2\}/e$

and

(6) $\vdash \{w=2w_0+1 \wedge z=z_0\}$ call G(w:F) $\{z=z_0+(w_0+1)^2\}/e$.

By axioms R5 and line 5

(7) $\vdash \{x=2y_0+1 \wedge z=z_0\}$ call F(x:G) $\{z=z_0+y_0^2\}/e$

By axiom R2 with the inactive substitution of 0 for z_0 and x_0 for y_0, we ge

(8) $\vdash \{x=2x_0+1 \wedge z=0\}$ call F(x:G) $\{z=x_0^2\}/e$

Line 8 together with two applications of B2 gives the desired result.

5. Soundness

In this section we outline a proof that the axiom system DS for programs with dynamic scope of identifiers is sound. We argue that if T is a sound proof system for the true formulas of the assertion language L_A then

$$\vdash_{DS,T}\{P\}\ A\ \{Q\}/e \text{ implies } \models_I\{P\}\ A\ \{Q\}/e,$$

The argument uses induction on the structure of proofs; we show that each instance of an axiom is true and that if all of the hypothesis of a rule of inference are true, the conclusion will be true also.

The only difficult case is rule of inference R1 for procedure calls. We assume that the hypothesis

$$\{U_0\}\text{call } F_0(\bar{x}_0:\bar{P}_0)\{V_0\}/e_0,\ldots,\{U_n\}\text{call } F_n(\bar{x}_n:\bar{P}_n)\{V_n\}/e_n$$

$$\vdash \{U_0\}K_0<\bar{P}_0>\{V_0\}/e_0,\ldots,\{U_n\}K_n<\bar{P}_n>\{V_n\}/e_n$$

of R1 is true and prove that

$$\vdash_I \{U_i\}\text{ call } F(\bar{x}_i:\bar{P}_i)\{V_i\}/e_i$$

must hold for $0\leq i\leq n$. Without loss of generality we also assume that the proof used to obtain

$$\{U_0\}\ K_0<\bar{P}_0>\{V_0\}/e_0,\ldots,\{U_n\}\ K_n<\bar{P}_n>\{V_n\}/e_n$$

from

$$\{U_0\}\text{call } F_0(\bar{x}_0:\bar{P}_0)\{V_0\}/e_0,\ldots,\{U_n\}\text{call } F_n(\bar{x}_n:\bar{P}_n)\{V_n\}/e_n$$

does not involve any additional applications of the axiom for procedure calls.

To simplify the proof we introduce a modified meaning function M_j. $M_j[S](e,s)$ is defined in exactly the same manner as $M[S](e,s)$ if S is not a procedure call. For procedure calls we have $M_j[\text{call } F(\bar{a}:\bar{P})](e,s)=M_{j-1}[K<\bar{a},\bar{P}>](e,s)$ if $j > 0$, "proc $F(\bar{x}:\bar{p})$; $K<\bar{x},\bar{p}>$ end"$\in e$, length $(\bar{x})$ = length $(\bar{a})$, and length $(\bar{P})$ = length $(\bar{p})$. $M_j[\text{call } F(\bar{a}:\bar{P})](e,s)$ is undefined otherwise. Thus M_j agrees with M on statements for which the maximum depth of procedure call does not exceed j-1.

We also extend the definition of partial correctness given in Section 2.

We write $\models {}^{j}\{P\}\ S\ \{Q\}/e$ iff $\forall s,s'[s\varepsilon P\ \wedge\ M_j[S](e,s)=s' \rightarrow s'\varepsilon Q]$

In the following lemma we state without proof some of the properties of M_j.

5.1 Lemma: (Properties of M_j)

(a) $\models {}^{0}\{U\}$ call $F(\bar{a}:\bar{P})\{V\}/e$ for all U, F, V, e.

(b) Suppose that $\Gamma \vdash \Delta$ where Γ and Δ are sets of partial correctness formulas of the form $\{P\}\ A\ \{Q\}/e$ and the formulas of Δ are obtained from those in Γ without use of axiom R1. Then $\models {}^{j}\Gamma$ implies $\models {}^{j}\Delta$

(c) If $\models {}^{j}\{U\}\ K<\bar{a},\bar{P}>\ \{V\}/e$ holds and the procedure with declaration "proc $F(\bar{x},\bar{p})$; $K<\bar{x},\bar{p}>$ end" is in e, then $\models {}^{j+1}\{U\}$ call $F(\bar{a}:\bar{P})\ \{V\}/e$ must hold also.

(d) If $M[S](e,s)=s'$ then there is a $k>0$ such that $j\geq k$ implies $M_j[S](e,s)=s'$.

The proofs of (a), (c), and (d) follow directly from the definitions of M_j. The proof of (b) is straightforward, since use of axiom R1 for procedure calls has been disallowed.

We return to the soundness proof for R1. By part (a) of the lemma

$$\models {}^{0}\{U_i\}\ \text{call}\ F_i(\bar{x}_i:\bar{P}_i)\{V_i\}/e_i,\quad 0\leq i\leq n$$

By the hypothesis of R1 and part (b) of the lemma, we see that

$$\models {}^{j}\{U_i\}\ \text{call}\ F_i(\bar{x}_i:\bar{P}_i)\{V_i\}/e_i,\quad 0\leq i\leq n$$

implies

$$\models {}^{j}\{U_i\}\ K_i<\bar{P}_i>\{V_i\}/e_i,\quad 0\leq i\leq n.$$

By part (c) of the lemma,

$$\models {}^{j}\{U_i\}\ \text{call}\ F_i(\bar{x}_i:\bar{P}_i)\{V_i\}/e_i,\quad 0\leq i\leq n$$

implies

$$\models {}^{j+1}\{U_i\}\ \text{call}\ F_i(\bar{x}_i:\bar{P}_i)\{V_i\}/e_i,\quad 0\leq i\leq n.$$

Hence, by induction we have for all $j \geq 0$

$$\models^{j} \{U_i\} \text{ call } F_i(\bar{x}_i:\bar{P}_i)\{V_i\}/e_i, \quad 0 \leq i \leq n,$$

Let $s \in U_i$ and suppose that $s' = M[\text{call } F_i(\bar{x}_i:\bar{P}_i)](e,s)$ then there is a $k>0$ such that $j \geq k$ implies $M_j[\text{call } F_i(\bar{x}_i:\bar{P}_i)](e,s)=s'$. Since $\models^{j} \{U_i\}$ call $F_i(\bar{x}_i:\bar{P}_i)$ $\{V_i\}/e$, we conclude that $s' \in V_i$.

Thus $\models_I \{U_i\}$ call $F_i(\bar{x}_i:\bar{P}_i)\{V_i\}/e_i$ holds for $0 \leq i \leq n$ and the proof of soundness is complete for R1. We leave the proof of soundness for the other axioms and rules of inference to the interested reader.

6. Completeness.

In this section we outline a proof that the axiom system DS is complete in the sense of Cook. Let T be a complete proof system for the true formulas of the assertion language L_A. Assume also that the assertion language L_A is expressive with respect to the expression language L_E and interpretation I. We prove that

$$\models_I \{U\}\ S\ \{V\}/e \text{ implies } \vdash_{DS,T} \{U\}\ S\ \{V\}/e.$$

The proof uses induction on the structure of the statement S and is generalization of the completeness proof for recursive procedures without procedure parameters given in [7]. Due to the length of the proof we will only consider the case where S is a procedure call; other cases will be left to the reader.

Assume that $\{U_0\}$ call $F_0(\bar{a}_0:\bar{P}_0)$ $\{V_0\}/e_0$ is true. We show that $\{U_0\}$ call $F_0(\bar{a}_0:\bar{P}_0)\{V_0\}/e_0$ is provable. Let "call $F_1(\bar{a}_1:\bar{P}_1)$",...,"call $F_n(\bar{a}_n:\bar{P}_n)$" be the procedure calls in the range of "call $F_0(\bar{a}_0:\bar{P}_0)$" and let e_i be the environment corresponding to "call $F_i(\bar{a}_i:\bar{P}_i)$". We assume that F_i has

declaration "proc $F_i(\bar{x}_i:\bar{P}_i)$; $K_i\langle\bar{x}_i,\bar{P}_i\rangle$ end", that $\bar{r}_i$ is the list of variables which are active in "call $F_i(\bar{x}_i:\bar{P}_i)$", and that $\bar{r}_i'$ is the list of variables which are active in "call $F_i(\bar{a}_i:\bar{P}_i)$". We also choose $\bar{c}_i$ to be a list of new variables which are inactive in "call $F_i(\bar{x}_i:\bar{P}_i)$" and "call $F_i(\bar{a}_i:\bar{P}_i)$"

To shorten notation, let

$$R_i \equiv \{\bar{r}_i=\bar{c}_i\}$$

$$R_i' \equiv \{\bar{r}_i'=\bar{c}_i\}$$

$$W_i \equiv SP(\text{call } F_i(\bar{x}_i:\bar{P}_i),\ e_i,\ R_i)$$

$$W_i' \equiv SP(\text{call } F_i(\bar{a}_i:\bar{P}_i),\ e_i,\ R_i')$$

$$L \equiv U_0 \frac{\bar{c}_0}{\bar{r}_0'} \quad .$$

Recall that SP(S,e,U) is the <u>strongest postcondition</u> corresponding to statement S and precondition U in environment e. Since L_A is expressive, it follows that W_i and W_i' may be represented by formulas of L_A for $0\le i\le n$.

We will show that

$$\{R_i\}\ \text{call } F_i(\bar{x}_i:\bar{P}_i)\ \{W_i\}/e_i \qquad (7.1)$$

is provable for all i, $0\le i\le n$. From this result it follows that $\{U_0\}$ call $F_0(\bar{a}_0:\bar{P}_0)$ $\{V_0\}/e_0$ is also provable. To see that this last part of the argument is correct, observe that

(a) $\vdash \{R_0'\}$ call $F_0(\bar{a}_0:\bar{P}_0)$ $\{W_0'\}/e_0$ by 7.5 and axiom R5 since $R_0' = R_0 \frac{\bar{a}_0}{\bar{x}_0}$ and $W_0' = W_0 \frac{\bar{a}_0}{\bar{x}_0}$.

(b) $\vdash \{R_0' \wedge L\}$ call $F_0(\bar{a}_0:\bar{P}_0)\{W_0' \wedge L\}/e_0$ by axioms R4.

(c) $\vdash \{\exists \bar{c}_0 [R_0' \wedge L]\}$ call $F_0(\bar{a}_0:\bar{P}_0)\{\exists c_0[W_0' \wedge L]\}/e_0$ by axiom R3.

(d) $\vdash U \longrightarrow \exists \bar{c}_0 [R_0' \wedge L]$ since T is a complete proof system for L_A and since $\models U_0 \equiv \exists \bar{c}_0[\bar{r}_0'=\bar{c}_0 \wedge U \frac{\bar{c}_0}{\bar{r}_0'}]$.

(e) $\models \exists \bar{c}_0[W_0' \wedge L] \longrightarrow SP(\text{call } F_0(\bar{a}_0:\bar{P}_0), e_0, U_0)$. Since L and the variables $\bar{c}_0$ are inactive with respect "call $F_0(\bar{a}_0:\bar{P}_0)$", we have

$$\models \exists \bar{c}_0[W_0' \wedge L] \equiv \exists \bar{c}_0[SP(\text{call } F_0(\bar{a}_0:\bar{P}_0), e_0, R_0') \wedge L)]$$

$$\equiv \exists \bar{c}_0[SP(\text{call } F_0(\bar{a}_0:\bar{P}_0), e_0, R_0' \wedge L)]$$

$$\equiv SP(\text{call } F_0(\bar{a}_0:\bar{P}_0), e_0, \exists \bar{c}_0[R_0' \wedge L])$$

$$\equiv SP(\text{call } F_0(\bar{a}_0:\bar{P}_0), e_0, U_0)$$

(f) $\vdash \exists \bar{c}_0[W_0' \wedge L] \longrightarrow SP(\text{call } F_0(\bar{a}_0:\bar{P}_0), e_0, U_0)$. This follows from (e) since T is a complete proof system for L_A.

(g) $\vdash \{U_0\}$ call $F_0(\bar{a}_0:\bar{P}_0)$ $\{SP(\text{call } F_0(\bar{a}_0:\bar{P}_0), e_0, U_0)\}/e_0$ by (c), (e), (f) and the rule of consequence.

(h) $\vdash SP(\text{call } F_0(\bar{a}_0:\bar{P}_0), e_0, U_0) \longrightarrow V_0$ since $\models \{U_0\}$ call $F_0(\bar{a}_0:\bar{P}_0)\{V_0\}/e_0$ and since $SP(\text{call } F_0(\bar{a}_0:\bar{P}_0), e_0, U_0)$ is the strongest postcondition corresponding to U_0 and "call $F_0(\bar{a}_0:\bar{P}_0)$".

(i) $\vdash \{U_0\}$ call $F_0(\bar{a}_0:\bar{P}_0)$ $\{V_0\}/e_0$ by (g),(h), and the rule of consequence.

It is still necessary to prove 7.1. We will show that

$$\{R_0\} \text{ call } F_0(\bar{x}_0:\bar{P}_0) \{W_0\}/e_0, \ldots, \{R_n\} \text{ call } F_n(\bar{x}_n:\bar{P}_n) \{W_n\}/e_n$$

$$\vdash \{R_0\}\ K_0\langle\bar{P}_0\rangle\ \{W_0\}/e_0, \ldots, \{R_n\} \text{ call } K_n\langle\bar{P}_n\rangle\ \{W_n\}/e_n. \qquad (7.2)$$

The proof of 7.1 will then follow by the axiom R1 for procedure calls. Proof of 7.2 is by induction on the structure of K_i using an induction hypothesis which is somewhat more general than what we need to prove.

7.3 Lemma: Let K be a statement and let R and W be predicates such that $\models \{R\}\ K\ \{W\}/e$ and such that the range of K with respect to e is included in $\langle\text{call } F_0(\bar{a}_0:\bar{P}_0),e_0\rangle,\ldots,\langle\text{call } F_n(\bar{a}_n:\bar{P}_n),e_n\rangle$, then

$$\{R_0\}\text{call } F_0(\bar{x}_0:\bar{P}_0)\{W_0\}/e_0,\ldots,\{R_n\}\text{call } F_n(\bar{x}_n:\bar{P}_n)\{W_n\}/e_n \vdash \{R\}\ K\ \{W\}/e$$

Proof: Proof is by induction on the structure of K. We will only consider the case where K is a procedure declaration i.e. K≡"begin proc $q(\bar{x}:\bar{p})$; L end; S end". If $\models \{R\}\ K\ \{W\}/e$ then we must also have $\models \{R\}\ K'\ \{W\}/e'$ where K' ≡ "begin S end" and e' = ADD(e, "proc $q(\bar{x}:\bar{p})$; L end"). Note that the range of K' with respect to e' is included within the range of K with respect to e. By the induction hypothesis we have that

$\{R_0\}\text{call } F_0(\bar{x}_0:\bar{P}_0)\ \{W_0\}/e_0,\ldots,\{R_n\}\text{call } F_n(\bar{x}_n:\bar{P}_n)\{W_n\}/e_n \vdash \{R\}\ K'\ \{W\}/e'$. By axiom B2, we see that $\{R_0\}\text{call } F_0(\bar{x}_0:\bar{P}_0)\{W_0\}/e_0,\ldots,\{R_n\}\text{call } F_n(\bar{x}_n:\bar{P}_n)\{W_n\}/e_n$ $\vdash \{R\}\ K\ \{W\}/e$.

Other cases in the proof of lemma 7.3 are left to the interested reader. Note that once lemma 7.3 has been established, 7.2 follows from the observation that $\vdash \{R_i\}\ K_i\langle\bar{P}_i\rangle\{W_i\}/e_i$, $0 \le i \le n$.

7. Coroutines.

A coroutine has 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 (this requirement 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, execution of a resume statement in Q_2 causes execution to be restarted following the last exit statement 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 formal operational specification of the semantics for coroutines is given in [1].

If recursive procedures are disallowed, a sound and complete axiom system may be obtained for the programming language of Section 2 with the addition of the coroutine construct. Such a system, based on the addition of auxiliary variables, is described in [2]. The axiom for the coroutine statement is similar to the one used by Clint [3]. However, the strategy used to obtain completeness is different from that advocated by Clint; auxiliary variables represent program counters (and therefore have bounded magnitude) rather than arbitrary stacks.

7.1 Theorem: There is a Hoare axiom system H for the programming language described above, including the coroutine construct but requiring that procedures be non-recursive, which is both sound and complete in the sense of Cook.

8. Coroutines and Recursion.

We show that it is impossible to obtain a sound-complete system of Hoare 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 9 that the notion of expressibility introduced in Section 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 features described in Sections 2 and 7 including both parameterless recursive procedures and the coroutine statement.

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

Proof: 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

accomplished by recursive calls of the procedure. The simulation routine is given in outline form below:

```
Prog_counter:=1;
Coroutine
   begin
     proc stack_1;
       new top, progress;
       progress:=1;
       while progress=1 do
          if prog_counter=1 then "INST1" else
          if prog_counter=2 then "INST2" else
                :
                :
          if prog_counter=K then "INSTK" else NULL
          end
     end stack_1;
     call stack_1
   end,
   begin
     proc stack_2;
       new top, progress;
       progress:=1;
       while progress=1 do
        if prog_counter=1 then "INST1*" else
        if prog_counter=2 then "INST2*" else
             :
             :
        if prog_counter=K then "INSTK*" 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 $INST^*_1,\ldots INST^*_K$ for the copy of the program within procedure stack_2 may be given.

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

The proof is similar to the proof of Theorem 3.1 and will be omitted.

9. Discussion of Results and Open Problems.

A number of open problems are suggested by the above results. An obvious question is whether there are other ways of restricting the programming language of Section 2 so that a sound and complete set of axioms can be obtained. For example, from Section 4 we know that such an axiom system could be obtained simply by disallowing global variables. Suppose that global variables were restricted to be read only instead of entirely disallowed. Would it then be possible to obtain a sound and complete axiom system? Automata theoretic considerations merely show that the type of

incompleteness argument used in this paper is not applicable.

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 7 seems to require that any such notion of expressibility be powerful enough to allow assertions about the status of the runtime stack(s). Clint [3] suggests the use of stack-valued auxiliary variables to prove properties of coroutines which involve recursion. It seems possible 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 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.

Finally we note that the technique of Sections 6 and 8 may be applied to a number of other programming language features including (a) call by name with functions and global variables, (b) unrestricted pointer variables with retention, (c) unrestricted pointer variables with recursion, and (d) label variables with retention. All these features present difficulties with respect to program proofs, and (one might argue) should be avoided in the design of programming languages suitable for program verification.

References

[1] 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.

[2] Clarke, Jr., E. M. Pathological Interaction of Programming Language Features. Technical Report CS-1976-15, Computer Science Department, Duke University, September 1976.

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

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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)

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

[14] Naur, P.(ed.) Revised Report on the Algorithmic Language Algol 60. CACM 6, 1(January, 1963), pp. 1-17.

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

[16] Wirth, N. The Programming Language PASCAL. Acta Informatica 1, 1, 1971, pp. 35-63.

[17] Johnston, J. B., "The Contour Model of Block Structured Processes". ACM SIGPLAN Symp. on Data Structures in Programming Languages, Univ. of Florida, Gainesville, February 1971.

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

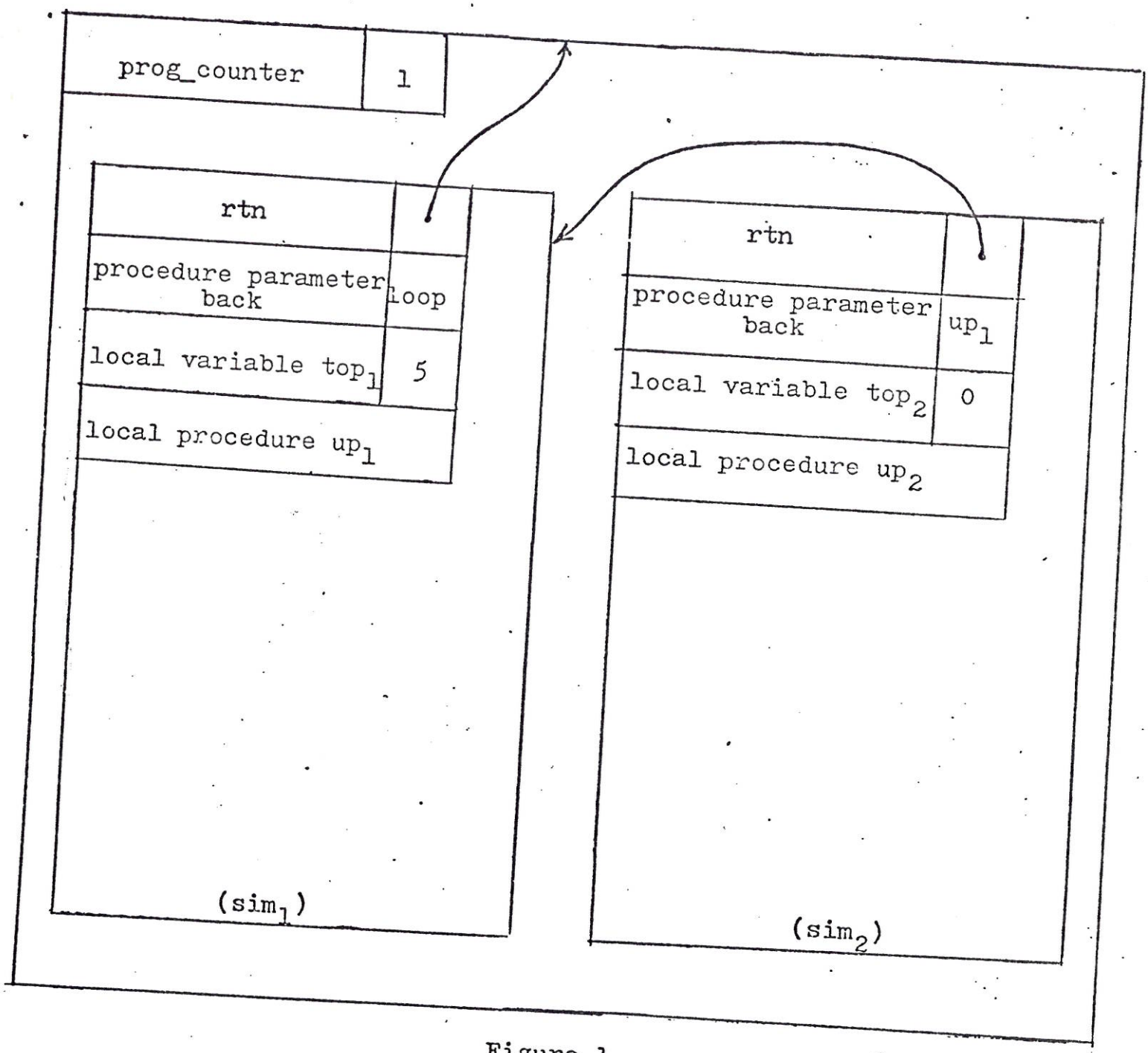

Figure 1

Contour diagram illustrating how instruction "enqueue 5" is simulated. Different activations of recursive procedure "sim" are distinguished by subscripts.

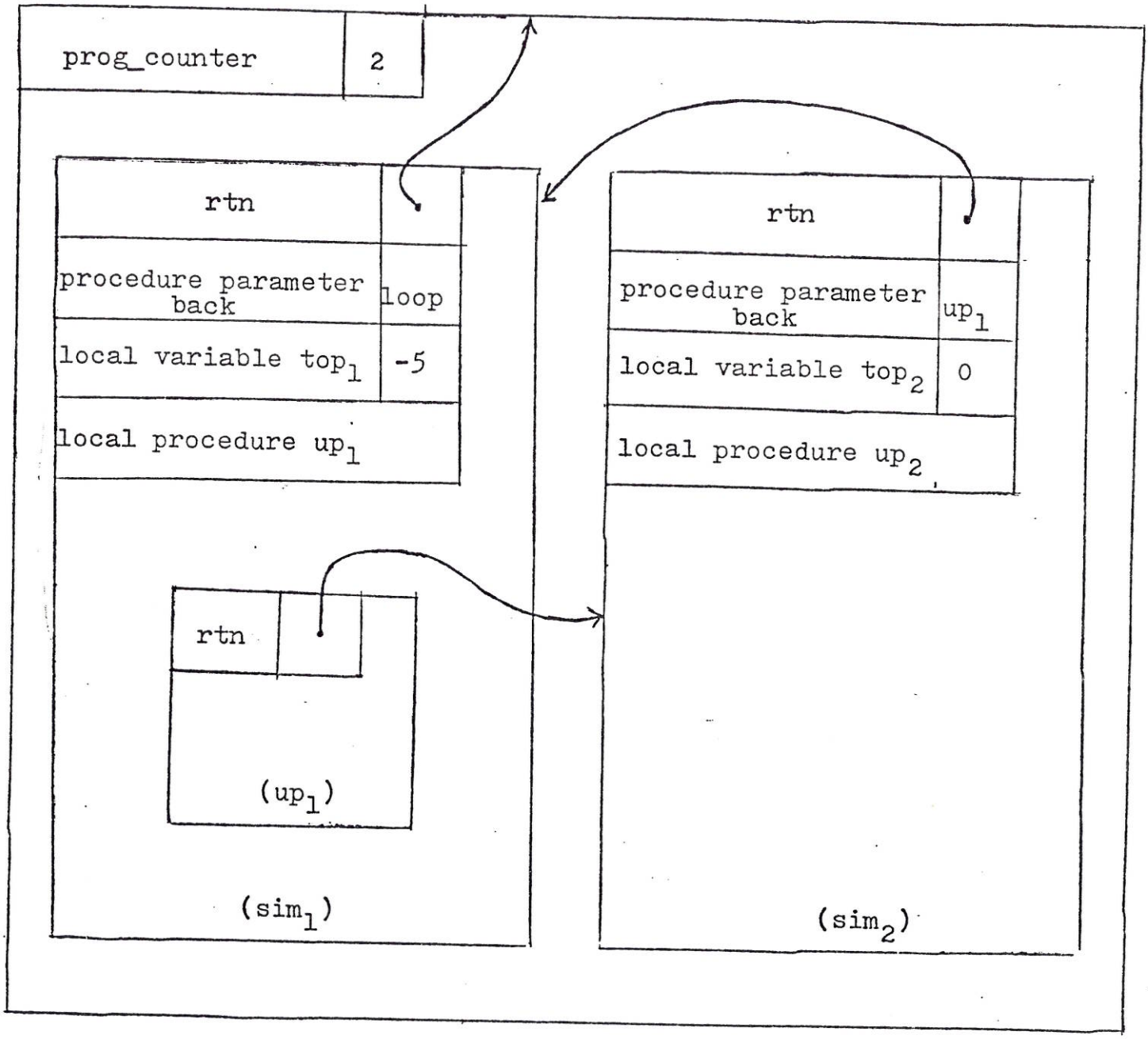

Figure 2

Contour disgram illustrating how instruction "dequeue X" is simulated. Local procedure up_1 is called from within second activation of procedure "sim".

	Language 1	Language 2	Language 3	Language 4	Language 5	Language 5
(1) Procedures with procedure parameters	inc.	no procedure names as parameters	inc.	inc.	inc.	inc.
(2) Recursion	inc.	inc.	no recursion	inc.	inc.	inc.
(3) Global variables	inc.	inc.	inc.	global variables disallowed	inc.	inc.
(4) Static Scope	inc.	inc.	inc.	inc.	dynamic scope	inc.
(5) Internal procedures	inc.	inc.	inc.	inc.	inc.	internal procedures not allowed
Sound and Complete Hoare axiom System	no	yes	yes	yes	yes	yes

Figure 3 THEOREM SUMMARY

(No Sharing or Self-Application)

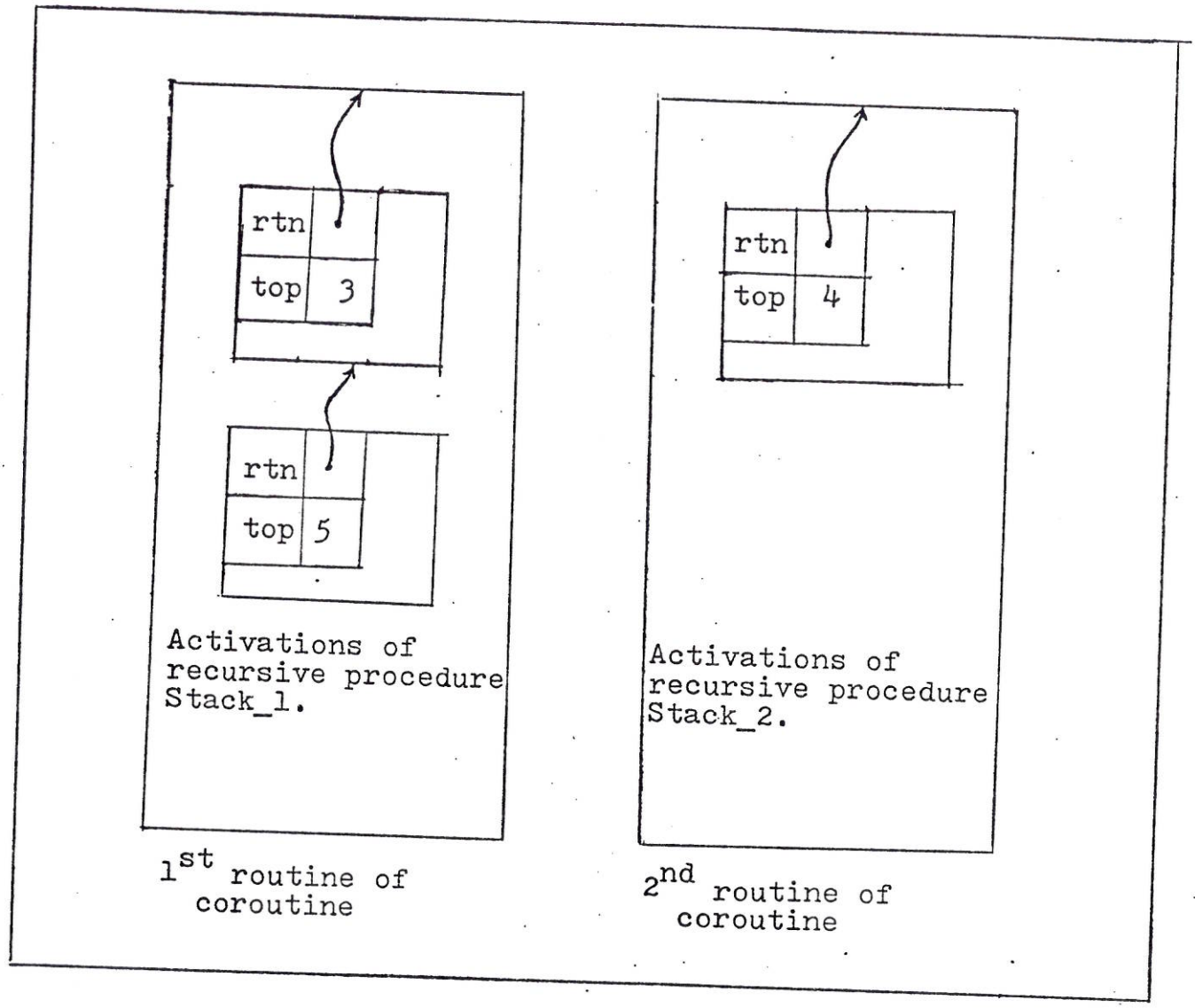

Figure 4

Simulation of two stack machine with program "push 3 on stack 1; push 4 on stack 2; push 5 on stack 1" by coroutine with local recursive procedures.