STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-200

STAN-CS-73-365

AUTOMATIC PROGRAM VERIFICATION I : A LOGICAL BASIS AND ITS IMPLEMENTATION

BY

SHIGERU IGARASHI
RALPH L. LONDON
AND
DAVID C. LUCKHAM

SUPPORTED BY

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
CONTRACT NSR 05-020-500
AND
ADVANCED RESEARCH PROJECTS AGENCY

ARPA ORDER NO.457

MAY 1973

COMPUTER SCIENCE DEPARTMENT

School of Humanities and Sciences

STANFORD UNIVERSITY

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY MAY 1973
MEMO AIM-200

COMPUTER SCIENCE DEPARTMENT
REPORT STAN-CS-73-365

USC INFORMATION SCIENCES INSTITUTE
REPORT ISI/RR-73-11

AUTOMATIC PROGRAM VERIFICATION I: A LOGICAL BASIS ANO ITS IMPLEMENTATION

by

Shigeru Igarashi
Ralph L. London
and
David C. Luckham

ABSTRACT: Defining the semantics of programming languages by axioms and rules of inference yields a deduction system within which proofs may be given that programs satisfy specifications. The deduction system herein is shown to be consistent and also deduct ion complete with respect to Hoare's system. A subgoaler for the deduction system is described whose input is a significant subset of Pascal programs plus inductive assertions. The output is a set of verification conditions or lemmas to be proved, Several non-trivial arithmetic and sorting programs have been shown to satisfy specifications by using an interactive theorem prover to automatically generate proofs of the verification condit ions, Additional components for a more powerful verification system are under construction.

Authors' addresses: Igarashi, Research Institute for Mathematical Sciences, Kyoto University, Kyoto 606, Japan; London, USC Information Sciences Institute, 4676 Admiralty Way, Marina Del Rey, California 90291; Luckham, Computer Science Department, Stanford University, Stanford, California 94305.

This research is supported by the Advanced Research Projects Agency under Contracts SD-183 and DAHC 15-72-C-0308, and by the National Aeronautics and Space Administration under Contract NSR 05-020-500.

AUTOMATIC PROGRAM VERIFICATION I: A LOGICAL BASIS AND ITS IMPLEMENTATION

by

Shigeru Igarashi, Ralph L. London, and David C. Luckham

1. INTRODUCTION

Verifying that a computer program is correct has been discussed in many recent publications, for example [Hoare 1969, King 1969, McCarthy and Painter 1967]. The "correctness problem" or "verification problem" has become popular essentially because it represents a significant first step towards writing programs that can be guaranteed to do what their authors intended. There are several different interpretations of exactly what it means, Here, we adopt the point of view that a program has been "verified" when it is proved within a system of logic to be consistent with documentation, i.e. a statement of what it is supposed to do. Our discussion is restricted to programs that can be written in a very precise modern programming langage, Pascal [Wirth 1971]. Of course, we do not deal with all Pascal programs, but with a subset that is rich enough to include published algorithms such as FIND [Hoare 1971b], TREESORT3 [Floyd 1964], and a simple compiler [McCarthy and Painter 1967]. Since Pascal is an Algol-like language we expect that what is done here can be repeated without much effort for Algol or other such languages. We adopt a DOCUMENTATION LANGUAGE that is roughly speaking the language of quantified Algol Boolean expressions, (i.e. first-order number theory with definitional extension and some notational conveniences). It does not contain any constructs for representing such notions as tense (time dependency), possibility (can do), etc. that may well prove useful in describing programs. So the documentation language is a slight extension of what programmers normally use to state those conditions on computations that control their programs, Statements of the documentation language are called ASSERTIONS. A documented program is, for us, a Pascal program in which assertions have been placed between its statements at certain points. We refer to such programs with documentation as ASSERTED-PROGRAMS.

The general idea of how to go about verifying an asserted program is to reduce this problem to questions about whether certain associated logical conditions (henceforth called VERIFICATION CONDITIONS) are true of (i.e. theorems in) various standard first-order theories. The usual method of reduction [Floyd 1967] involves enumerating all possible paths between assertions in the program and then computing a verification condition for each path in terms of operations and assertions on that path: these verification conditions must then be proved. See London [1972] for a bibliography of existing programs for generating verification conditions, However, in the case of Pascal, a rigorous definition of the semantics has been given in terms of axioms and rules of inference that must be valid for each syntactic constructor: this is contained in the recent work of Hoare and Wirth[1972]. This approach to defining the semantics of a programming language yields a deduction system in which proofs that programs satisfy specifications may be given (see Hoare[1969,1971a]). Such proofs, of course, depend on the truth of first-order conditions, or to put it another way, standard first-order theories are sub-systems of the deduction system for Pascal. For the sake of illustration, Example 1 shows a proof in Hoare's system that the program in step 13 computes the quotient q and remainder r of the inputs x and y. The rules of inference used here are the rules in Table 1 (Section 3.1) and the iteration rule below. The logical conditions assumed by this proof are labeled "lemma".

Iteration: $$\frac{P \wedge Q\{A\}P,\ P \wedge \neg Q \supset R}{P\{\text{while } Q \text{ do } A\}R}$$

1.	$\text{true} \rightarrow x = x + y * 0$	Lemma 1
2.	$x = x + y * 0\ \{r \leftarrow x\}\ x = r + y * 0$	C1
3.	$x = r + y * 0\ \{q \leftarrow 0\}\ x = r + y * q$	C1
4.	$\text{true}\ \{r \leftarrow x\}\ x = r + y * 0$	C5 (1,2)
5.	$\text{true}\ \{r \leftarrow x;\ q \leftarrow 0\}\ x = r + y * q$	C7 (4,3)
6.	$x = r + y * q \wedge y \le r \rightarrow x = (r-y) + y * (1+q)$	Lemma 2
7.	$x = (r-y) + y * (1+q)\ \{r \leftarrow r-y\}\ x = r + y * (1+q)$	C1
8.	$x = r + y * (1+q)\ \{q \leftarrow 1+q\}\ x = r + y * q$	C1
9.	$x = (r-y) + y * (1+q)\ \{r \leftarrow r-y;\ q \leftarrow 1+q\}\ x = r + y * q$	C7 (7,8)
10.	$x = r + y * q \wedge y \le r\ \{r \leftarrow r-y;\ q \leftarrow 1+q\}\ x = r + y * q$	C5 (6,9)
11.	$x = r + y * q \wedge \neg y \le r \rightarrow \neg y \le r \wedge x = r + y * q$	Lemma 3
12.	$x = r + y * q\ \{\text{while } y \le r \text{ do}(r \leftarrow r-y;q \leftarrow 1+q)\}\ \neg y \le r \wedge x = r + y * q$	Iteration (10,11)
13.	$\text{true}\{((r \leftarrow x;\ q \leftarrow 0);\ \text{while } y \le r \text{ do } (r \leftarrow r-y;\ q \leftarrow 1+q))\}\ \neg y \le r \wedge x = r + y * q$	C7 (5,12)

EXAMPLE 1: FORMAL VERIFICATION OF QUOTIENT-REMAINDER PROGRAM

It is possible to generate the verification conditions for an asserted program merely by using a subgoaler for the deduction system. EXAMPLE 2 shows how such a subgoaler works on the Quotient-Remainder program of Example 1; it simply searches for a rule of inference which has the current goal as its conclusion and then generates the premisses of the rule as subgoals.

Goal.	true $\{r \leftarrow x;\ q \leftarrow 0;$ assert $x = r + y * q;$ while $y \le r$ do begin $r \leftarrow r-y;$ $q \leftarrow 1+q$ end$\}\ \neg(y \le r) \wedge (x = r + y * q)$	
Subgoal 1.	true $\{r \leftarrow x;\ q \leftarrow 0\}\ x = r + y * q$	C7 (Goal)
Subgoal 2.	$x = r + y * q\ \{$while $y \le r$ do begin $r \leftarrow r-y;$ $q \leftarrow 1+q$ end$\}\ \neg(y \le r) \wedge (x = r+y*q)$	C7 (Goal)
Lemma 3.	$(x = r + y * q) \wedge \neg(y \le r) \rightarrow \neg(y \le r) \wedge (x = r+y*q)$	Iteration (Subgoal 2)
Subgoal 3.	$(x = r+y*q) \wedge (y \le r)\ \{r \leftarrow r - y;\ q \leftarrow 1+q\}\ x = r+y*q$	iteration (Subgoal 2)
Subgoal 4.	$(x = r+y*q) \wedge (y \le r)\ \{r \leftarrow r-y\}\ x = r+y*(1+q)$	C7 (Subgoal 3), then C1 (Subgoal 3)
Lemma 2.	$(x=r+y*q) \wedge (y \le r) \rightarrow x = (r-y)+y*(1+q)$	C1 (Subgoal 4), then C5 (Subgoal 4)
Subgoal 5.	true $\{r \leftarrow x\}\ x = r+y*0$	C7 (Subgoal 1), then C1 (Subgoal 1)
Lemma 1.	true $\rightarrow x = x + y * 0$	C1 (Subgoal 5), then C5 (Subgoal 5)

-EXAMPLE 2: GENERATION OF THE VERIFICATION CONDITIONS FOR THE QUOTIENT-REMAINDER PROGRAM

Note that, for example, subgoal 4 is obtained from subgoal 3 by using C7 (composition rule) to split the compound statement at the semi-colon; Q is set to $x = r+y*(1+q)$ by applying C1 (assignment axiom) so that the other subgoal is $x = r+y*(1+q)\ \{q \leftarrow 1+q\}\ x = r+y*q$ which is an instance of the assignment axiom and hence is satisfied. If the first-order "lemmas" produced by the subgoaler are true of the relevant theories (in this case, number theory) then we know that there will be a proof verifying the Quotient-Remainder program in Hoare's system. These verification conditions are sufficient conditions.

This is the approach to generating verification conditions presented here. We use a simple subgoaling program for Hoare's deduction system. Although this program will accept a significant subset of Pascal programs, it is itself very simple since it does not analyze the object program explicitly but merely repeatedly applies a list of rules of inference, It is easily shown to be sound (see below), easily extended to accept additional syntax (FOR statements, new type declarations, etc.), and easily changed to take account of . new definitions of the semantics. We refer to this subgoaler as VCG (Verification Condition Generator); details of its implementation are given in Section 4 and sample outputs in Section 5.

However, there are problems. At any step more than one deduction rule may be applicable to generate further subgoals. To deal with this ambiguity, we have chosen a set of deduction rules (some of them derived rules in Hoare's system) for subgoal generation which is unambiguous. We shall show that this set is deduction complete. This means that if a particular verification can be proved in Hoare's system, then VCG will produce a sufficient set of verification conditions from which such a proof may then be constructed. However, these conditions may not be provable unless the user supplies certain crucial assertions at intermediate points in his program (e.g. an invariant for each loop). Finally we also need to know that the deduction system is consistent.

Section 3 deals with these logical problems. We give a small set of axioms and deduction rules, called the CORE, from which all of Hoare's rules can be derived; some sample derivations are included. A straight-forward set theoretic model of the core is constructed; this gives us a semantic proof of consistency of the core. The set of rules used by VCG is given and is shown to be consistent with the core and powerful e'nough to derive the core (hence deduction completeness). Preliminary comments, definitions and examples concerning Pascal programs, the assertion language and asserted programs are given in Section 2.

VCG is already a useful tool, Numerous example programs have been verified by manually proving the verification conditions. More interestingly, and of more promise, VCG is intended to be the initial part of an automatic verification and debugging system, The overall plan is shown in Figure 1. Asserted programs are input to VCG. The output verification conditions are simplified relative to data files containing relevant properties of the operators and functions in the conditions. It will become evident from the examples in Section 5 that a great deal of elementary simplification of verification conditions is both necessary and easy to do. The truth of many of the conditions will be established at the simplification stage. Next, the condition Analyzer is intended to reduce problems given to the theorem prover and to find bugs. It attempts to classify verification conditions according to probable method of proof and to generate simpler subproblems, and also attempts to find the "closest" similar condition that is provable when a proof of a given condition is not found. This latter kind of analysis is one method of catching bugs--finding missing assumptions in verification conditions. Currently a development of the theorem-prover of Allen and Luckham [1970] is being used successfully by J. Morales to prove conditions output by VCG for various sorting programs (see Section 5.41. This proposed system thus appears to have a good chance of being developed into something useful.

What has become evident is that VCG is not a trivial element in this type of verification system. In order to make such a system practical, the amount of documentation the user is required to supply with his program should be restricted to what would be considered natural for human understanding of what the program and its sub-programs do. At the moment VCG places rather more weight on documentation than we would like. However it is already easy to see how to extend VCG by adding some additional rules that will permit it to deduce intermediate documentation for itself in some cases.

```
                            --------------                      -------------
                            | DATA FILES |                      | DATA FILES|
                            ----------|----                     --w----------
                                 ^    |                               ^  |
   Input                         |    v                               |  v
          ---------         --------------       ------------    ----------
          |  VCG   |        | SIMPLIFIER |       | ANALYZER |-->| THEOREM |
     ---->|        |--->    |            |--->   |          |c--| PROVER  |
          ---------         --------------       ------------    ----------
                                 |                    |
                                 |                    |
                                 |                    |
                                 v                    v
                             ----------           ----------
                             | OUTPUT |           | OUTPUT |
                             ----------           ----------
```

FIGURE 1: PLANNED AUTOMATIC VERIFICATION AND DEBUGGING SYSTEM

2. PROGRAMS WITH ASSERTIONS

2.1 PASCAL,

A comprehensive definition of Pascal is published by Wirth [1971, 1972] and Hoare and Wirth [1972]. Our choice of Pascal as the programming language is motivated by the development of Hoare's deduction system and its use to define the semantics of Pascal. Pascal is an Algol-like language so a reader familiar with Algol will have no trouble understanding the examples of programs and condition generation in this paper. Thus instead of including a definition of Pascal here, we shall point out some of the main differences of concern to us between Pascal and Algol. The following example shows a program containing a procedure definition, variable declarations, a recursive function definition and a program body which calls the procedure and function; it is written first in Algol and then in Pascal.

ALGOL PROGRAM:

```
BEGIN
INTEGER ALPHA, BETA, QUOT, REM, Q, R, X, Y, I;

PROCEDURE QUOTREM(R,Q,X,Y); VALUE X, Y; INTEGER R, Q, X, Y;
BEGIN R := X; Q := 0;
      FOR I := 1 WHILE Y ≤ R DO
             BEGIN R := R - Y; Q := 1 t Q END
END;

INTEGER PROCEDURE FACT(N); INTEGER N;
BEGIN IF N = 0 THEN FACT := 1 ELSE FACT := N *FACT(N-1) END:

BETA := 3; X := 6; Y := 4;
ALPHA := FACT(BETA);
QUOTREM(QUOT, REM, X+Y, X-Y);
Q := QUOT; R := REM
END
```

PASCAL PROGRAM:

```
VAR ALPHA, BETA, QUOT, REM, Q, R, X, Y : INTEGER;

PROCEDURE QUOTREM(VAR R, Q : INTEGER; X, Y : INTEGER);
BEGIN R := X; Q := 0;
      WHILE Y ≤ R DO
             BEGIN R := R - Y; Q := 1 t Q  END
END:

FUNCTION FACT (N: INTEGER) : INTEGER:
BEGIN IF N = 0 THEN FACT := 1 ELSE FACT := N *FACT(N-1) END;
BEGIN   BETA  :=  3;  X  := 6; Y := 4;
           ALPHA  := FACT(BETA);
           QUOTREM (QUOT, REM, X+Y,X-Y);
            Q  := QUOT; R := REM
END.
```

EXAMPLE 3: A PROGRAM IN ALGOL AN0 PASCAL

The differences in declaring variables are unimportant for our purposes. The type of the function is indicated after the right parenthesis in Pascal rather than before the word “PROCEDURE” in Algol. The opening “BEGIN” in Algol appears just before the main program ‘in Pascal. In the formal parameter part of procedure and function definitions, Pascal includes the specification of the formal parameters inside the parentheses: in Algol this specification is made after the list of parameters to be called by value.

The remaining difference may be skipped until procedures are discussed in detail later. The word "VAR" in the Pascal formal parameter part means R and Q are variable parameters. The corresponding actual parameters must be variables (and not more general expressions); assignment to R or Q in the body of the procedure affects the corresponding actual parameters. The absence of "VAR" before X and Y means X and Y are value parameters in the Algol 60 sense (representing a change in the revised Pascal from the ‘original definition). The corresponding actual parameters must be expressions (of which a variable is a simple case). A value parameter represents a variable local to the procedure to which the value of the corresponding actual parameter is initially assigned upon activation of the procedure. Assignments to value parameters from within the procedure are permitted, but do not affect the corresponding actual parameters. (For further details of Pascal see Wirth [1971, 1972]).

At the moment VCG will accept a subset of legal Pascal programs built up from: assignment, while, conditional, and go to statements; recursive procedure and function definitions and calls; one-dimensional arrays are allowed on either side of assignment statements.

2.2 ASSERTIONS

Assertions are conditions on the state of the computation of a program. Thus, if assertion P is placed at some point in program A, the intention is that when A is run, every time P is encountered P must be true of the current computation state of A.

Essentially, our assertion language allows assertions to contain any well-formed formula of a standard first-order theory and in addition, non-standard relations may be introduced by definitions. In practice we have adopted a slightly more usable and readable formal language for the assertions of VCG.

(i) A term in the assertion language is a Pascal expression,

(ii) Atomic assertions are either predicates (i.e. identifiers) with terms as arguments or terms,

(iii) Assertions are well-formed logical formulas constructed from atomic assertions using logical connectives and quantifiers according to the usual well-known rules.

Here are some examples:

(1) X = Y+Z

(2) ¬(Y≤R) A (X = R+Y*Q)

(3) Z*POWER(W,I) = POWER(X,Y)

(4) ∀K((1≤K) A (K≤N-1) ⊃ A[K] ≤ A[K+1])&
PERMUTATION(A,A0).

The first three assertions are expressions in Pascal (and in fact Boolean expressions in Algol) and use a precedence among operators to simplify notation (below). Assertion (4) is not a Boolean expression in Algol (because it contains a quantifier) nor an expression in Pascal (because of the quantifier and implication),

The assertion language contains different connective symbols for both IMPLICATION and AND to improve readability of verification conditions. The precedence order of connectives and arithmetical operators, predicates, and quantifiers is:

1. &(and); 2. →(implies), ⊃ (implies); 3. =, ≠, <, >, ≤, ≥; 4. v, +, -; 5. A (and), *, /, DIV, ROD; 6. ¬, v, ∃.

This agrees with the precedence in Pascal expressions.

NOTATION: Assertions and Boolean expressions will usually be denoted by P,Q,R,S.

2.3 ASSERTED PROGRAMS

Assertions are added to programs as additional statements beginning with the special symbol ASSERT, namely

<assert statement> ::= ASSERT <assertion>

Thus an asserted program is a legal Pascal program if we imagine that the syntax of the Pascal statement is extended by adding the extra clause below to the syntax diagram of "statement" (see appendix to Wirth [1972]):

```
                 ------------          ----------------
------>  (ASSERT l--------->   |ASSERTION l---->
                 ------------          ----------------
```

The assertions at the entry and exit of a procedure definition, function definition, or main program have the word "ASSERT" replaced by "ENTRY" and "EXIT" respectively. Both entry and exit statements appear at the beginning of the unit.

There are some further restrictions. The basic rule about placing assertions in a source program is that every loop must contain at least one asset-tion. This requirement is met if there is an assertion at every iteration statement (i.e., immediately before the statement) and an assertion at every label (i.e., just after the label). Although these requirements are not a necessary condition, they are a simple and convenient sufficient condition to guarantee an assertion in every loop. An assertion is required, for the exit of a program. With no loss of generality we assume a single exit, 'Assertions may optionally be placed anywhere else. If an assertion is missing from the entrance, VCG will assume the entry assertion "UNRESTRICTED", a synonym for "TRUE". A source program with assertions placed to meet these requirements is called an ASSERTED PROGRAM. Examples of asserted programs are given in Section 5.

NOTATION: Asserted programs will be denoted by A,B,C,D.

2.4 LOGIC OF ASSERTED PROGRAMS

We review briefly here the elements of Hoare's inference system for proving properties of programs.

STATEMENTS of the logic are of three kinds.

(i) assertions,

(ii) statements of the form P{A}Q where P,Q are assertions and A is a program or asserted program.

P{A}Q means "if P is true of the input' state and A halts (or halts normally in the case that A contains a GO TO to a label not in A) then Q is true of the output state".

(iii) procedure declarations (definitions) of the form p PROC K where p is a procedure name and K is a program or asserted program (the procedure body).

There is an infinite set of variables p,q,r,... that range over procedures. Thus undeclared procedure names occurring in statements are free variables ranging over procedures.

A RULE OF INFERENCE is a transformation rule from a set of statements (premises, say $H_1,\dots,H_n$) to a statement (conclusion, say K) that is always of kind (ii). Such rules are denoted by

$$\text{R1.}\quad \frac{H_1,\dots,H_n}{K}$$

The concept of PROOF in Hoare's system is defined in the usual way as a sequence of statements that are either axioms or obtained from previous members of the sequence by a rule, A sequence is a proof of its end statement.

We use H ||- K to denote that K can be proved by assuming H. H |- K denotes the same thing for first order logic.

Some rules have the existence of a subproof as a premiss; they are of the form

$$\text{R2.}\quad \frac{H_1,\dots,H_n,\ I \Vdash J}{K}$$

Such rules permit deductions of assertions on recursive procedure calls.

We extend the definition of proof to include the notion of assumption or dependency. An arbitrary well-formed formula can appear in a proof sequence, But in such a case that formula is said to have a formula identical with itself as its (unique) assumption formula. Each formula in the sequence has an associated set of assumption formulas, which can be empty, and which must be empty if it is the end- formula in the sequence. Each rule of inference preserves the assumptions unless specified otherwise, Thus the conclusion of a rule of the form R1 is dependent on the set of assumptions that is the set-theoretic union of the sets of assumptions of the premisses. In other words, assumptions are inherited from premisses to conclusions.

Assumptions can be discharged only if the rule is of the form R2. In this case the assumption formula designated by I can be discharged from the set of assumptions associated with the conclusion designated by K, while other assumptions are inherited.

Intuitively I||- J means I implies J, and a free variable, say r, reads "for any r".

The rules of inference discussed in the following sections all have, with one exception, at most two premisses. Proofs may be represented in the usual way by binary trees.

SUBSTITUTION of an expression t for a variable x in an expression E is denoted by $E|_{t}^{x}$.

We note that the termination of a program A is not expressable in Hoare's system by statements of the form P{A}Q. On the other hand, non-termination can be expressed by statements such as TRUE{A}FALSE. There may be some indirect ways of constructing formulas that mean 'A terminates for all inputs satisfying P", and if so, it would be nice to know for what class of programs this can be done.

REMARKS:

We presuppose a standard first-order theory, which shall be denoted by T, representing the properties of the primitive functions and predicates used in Pascal. However, our construction is uniform in that choosing different first-order theories characterizing possibly different functions and predicates does not affect the framework. A standard model of the theory T is fixed and denoted by M.

In our formal system there are three kinds of procedure names we have to distinguish:
1) Procedure names for primitive procedures. For instance a library procedure whose body is inherently written in a language of lower level belongs to this category. (It is even possible for us to regard the assignment statement as such a procedure.)
2) Procedure names for declared procedures. We regard procedure declarations as the "defining axioms" of such procedure names, which constitute nonlogical axioms in our system and shall be denoted by J. We assume J does not assign more than one procedure to a name.
3) Procedure names used in derivations. In the formal system we will use procedure names which should intuitively be regarded as "free variables", which represent arbitrary procedures. In proving metatheorems we will use a name for each declared procedure.

Besides the above, each procedure name is assuhied to have "arity", so that it can represent or vary over declared procedures with, say, m variable parameters and n value parameters. Such a procedure will be called (m,n)-ary and the m (variable) parameters and the n (value) parameters will be called the left and the right parameters, respectively.

If a primitive procedure name, say q, occurs in a program about which we are to prove a certain theorem, we have to either give a set of (nonlogical1 axioms of the form P{q(x;y)}R or a defining axiom for q. In most cases, we shall assume that the procedure can be written in Pascal and that there is a defining axiom for it.

3. THE BASIS INFERENCE SYSTEM FOR VCG.

In this section we study the properties of the set V of axioms and rules of inference used by VCG. One of our main concerns is that the rules of inference in V should be unambiguous in the sense that only one rule is applicable to generate subgoals from any given goal. This will certainly be the case if no two rules have conclusions which have common substitution instances, a property which is true of V. The rules of V, which appear as Table 2 in section 3.3, are simple combinations of Hoare's original set of rules H given in Hoare [1971a, p. 116]. Having chosen V, we must establish that it is both sound and deduction complete. We shall show first that a set C of simple rules (the CORE) is sound and that any rule in H can be derived from C. We then show that V and C are inter-derivable. We shall begin by studying the relative derivability when none of the sets of rules contains go to's or array variables. The rules H are equivalent to the following set of rules.

3.1 THE CORE RULES

The set of axioms and rules of the core is given in Table 1. Rules D3 (iteration), D7 (adaptation) of H have been omitted; D4 (alternation) has been replaced by C8 (conditional). We have added the frame axiom (C2) for procedure calls and the and-or rule (C6); Hoare's substitution rule (D6) corresponds to our left and right substitution rules,

NOTATION: x, y, z - lists of variables: p, q, r - procedure names; s, t - lists of expressions; K - procedure body; p(x;y) - denotes CALL p(x;y) where x and y are the left and right parameters of p. VAR(P) denotes the free variables of P; p(x;y) PROC K denotes a declaration of the form "PROCEDURE p(x;y); BEGIN K END".

AXIOMS

C1. assignment axioms: $P|^{x}_{t}\ \{x \leftarrow t\}\ P$

C2. frame axioms: $P\{q(x;t)\}P$ provided $\neg(x \in VAR(P))$

C3. procedure declarations: p(x;y) PROC K.

C4. logical theorems: P for all P s.t. $\vdash$ P.

RULES

C5. consequence: $$\frac{P\supset Q,\ Q\{A\}R}{P\{A\}R}\quad,\quad\frac{P\{A\}Q,\ Q\supset R}{P\{A\}R}$$

C6. and/or: $$\frac{P\{A\}Q,R\{A\}S}{P\wedge R\{A\}Q\wedge S}\qquad\frac{P\{A\}Q,R\{A\}S}{P\vee R\{A\}Q\vee S}$$

C7. composition: $$\frac{P\{A\}Q,\ Q\{B\}R}{P\{A;B\}R}$$

C8. conditional: $$\frac{P\wedge R\{A\}Q,P\wedge\neg R\{B\}Q}{P\{\text{IF } R \text{ THEN } A \text{ ELSE } B\}Q}$$

C9. substitution: $$(L)\quad\frac{P(x;y)\ \{q(x;y)\}Q(x;y)}{P(z;y)\{q(z;y)\}Q(z;y)}$$

$$(R)\quad\frac{P(x;y)\ \{q(x;y)\}Q(x;y)}{P(x;s)\{q(x;s)\}Q(x;s)}$$

SUBJECT TO THE RESTRICTIONS:(i) s does not contain members of x; (ii) members of z must be distinct and y and z are disjoint.

C10. procedure call: $$\frac{p(x;y)\ \text{PROC}\ K(p),\ P\{r(x;y)\}Q\vdash P\{K(r)\}Q}{P\{p(x;y)\}Q}$$

where p does not occur in the proof of the right hand premiss, and r does not occur in any other assumption in that proof.

TABLE 1 C:THE CORE RULES.

In order to demonstrate that C is as "powerful" as H we show that any proof in H of P{A}Q can be transformed into a proof in C of P{A'}Q where A' is a program equivalent to A. An application of a rule R (that is not a rule in C) in the given proof is to be replaced by a derivation in C of the conclusion of R assuming the premisses of R. The transformed proof will use only rules of C and will prove essentially the same formal statement. It is clear that applications of Hoare's substitution rule (D6) can be replaced by successive applications of the left and right rules (C9). We therefore need only consider the following three rules.

(D4) Alternation:

$$\frac{P1\{A\}Q, \quad P2\{B\}Q}{\text{if } R \text{ then } P1 \text{ else } P2\{\text{if } R \text{ then } A \text{ else } B\}Q}$$

(D7) Adaptation:

$$\frac{P(a;e)\{p(a;e)\}R(a;e)}{P(a;e)\wedge\forall a(R(a;e)\supset S(a;e))\{p(a;e)\}S(a;e)}$$

(D3) Iteration:

$$\frac{P\{A\}S, \quad S\vdash \text{if } Q \text{ then } P \text{ else } R}{S\{\text{while } Q \text{ do } A\}R}$$

(a) 04 is derivable in C. Let P in the conditional rule (C8) be: if R then P1 else P2.

1. $P1\{A\}Q,\ P2\{B\}Q$ assumptions (premisses of D4)

2. $P\wedge R\supset P1,\ P\wedge\neg R\supset P2$

3. $P\wedge R\{A\}Q, P\wedge\neg R\{B\}$ Cl consequence (C5)1,2

4. if R then P1 else P2{if R then A else B}Q
conditional (C8)3.

(b) D7 is derivable in C.

1. $P(a;e)\{p(a;e)\}R(a;e)$ assumption (premiss D7)

2. $\forall a(R(a;e)\supset S(a;e))\{p(a;e)\}\forall a(R(a;e)\supset S(a;e))$
frame axiom (C2).

3. $P(a;e)\wedge\forall a(R(a;e)\supset S(a;e))\{p(a;e)\}R(a;e)\wedge$
$\forall a(R(a;e)\supset S(a;e))$
and rule (C6)1,2.

4. $P(a;e)\wedge\forall a(R(a;e)\supset S(a;e))\{p(a;e)\}S(a;e)$ C5,3.

Corresponding to any while statement "while Q do A" we can define a recursive procedure:

```
procedure whiledef(x;v);

if Q then begin A; call whiledef(x;v);end
else end
```

where x is the list of variables in A that are subject to change in the body A, and v is the list of all other variables in Q or A.

We consider a modified form of the iteration rule:

(D3') $\dfrac{P\{A\}S,\ S \supset \text{if } Q \text{ then } P \text{ else } R}{S\{\text{call whiledef}(x;v)\}R}$

(c). D3' is derivable in C.

1.	P{A}S	Assumption (premiss D3').
2.	S∧Q⊃P	Assumption (premiss D3')
3.	S∧¬Q⊃R	Assumption (premiss D3')
4.	S{call r(x;v)}R	Assumption
5.	P{A;call r(x;v)}R	C7, 1,4
6.	S∧Q{A;call r(x;v)}R	C5, 2,5
7.	S{if Q then begin A; call r(x;v);end else end}R	C8, 6,3
8.	S{call whiledef(x;v)}R	C10, 4,7

If we are given a proof in H of P{A}Q we may replace applications of D4 and D7 by the proofs (a) and (b); an application of D3 is replaced by a proof (c) of D3'. We will then have a proof in C of P{A'}Q where A' is the result of replacing each while statement in A by a call to the corresponding whiledef procedure. This is easily proved by induction on the length of the proof. Clearly A' is equivalent to A. This completes the proof that C is as powerful as H.

In the other direction, all of the core rules except the frame axiom and the and-or rule appear in H with minor differences and are easily shown to be derivable in H. Thus, to show that proofs in C can be carried out in H, we need only be concerned with eliminating C2 and C6.

Recall that a Pascal program must contain definitions of all called procedures except library procedures and there are a finite number of those. This places a finite bound on the number of different procedures that can ever be called in any computation of a program.

d. Lemma

||- TRUE{A} TRUE for any program A.

PROOF

We can construct a proof of TRUE{A}TRUE by using the rules (D1-D5) to generate subgoals starting from the goal TRUE{A}TRUE. Assume a list of variables $r_1, r_2, r_3 \ldots$ distinct from the list of procedure names that may be called in a computation of A. Subgoals are generated by applying the rules recursively as follows (D3 and D4 are equivalent to D3* and D4*):

(D2) Subgoals
$$\frac{\text{TRUE}\{A\}\text{TRUE}, \quad \text{TRUE}\{B\}\text{TRUE}}{\text{TRUE}\{A;B\}\text{TRUE}}$$

(D1) Subgoal
$$\frac{\text{TRUE}\{B\}\text{TRUE}}{\text{TRUE}\wedge P\{B\}\text{TRUE}, \quad (\text{TRUE}\wedge\neg P)\supset\text{TRUE}}$$

(D3)* Goal
$$\text{TRUE}\{\text{while } P \text{ do } B\}\text{TRUE}$$

(D1) Subgoals
$$\frac{\text{TRUE}\{B\}\text{TRUE}}{\text{TRUE}\wedge P\{B\}\text{TRUE},} \quad \frac{\text{TRUE}\{C\}\text{TRUE}}{\text{TRUE}\wedge\neg P\{C\}\text{TRUE}}$$

(D4)* Goal
$$\text{TRUE}\{\text{if } P \text{ then } B \text{ else } C\}\text{TRUE}$$

.(D5) Subgoal
$$\frac{\text{TRUE}\{K(r_p)\}\text{TRUE}}{\text{TRUE}\{p(x;v)\}\text{TRUE}}$$
Goal

where K is the body of p and r_p is a unique variable to be substituted for the procedure name p in every subsequent subgoal of the goal. The procedure terminates since the subgoals in each of the rules 02 - D4 are shorter than the goals, and D5 can be applied only finitely many times since the list of procedure names that can occur is finite and one of these names is eliminated from all further subgoals of a goal to which D5 applies. The length of any subgoal branch is bounded by 2nl where n is the number of procedures that can be called by A and l is the number of statements in A. The terminal subgoals are of two kinds: TRUE{x←t}TRUE (axioms) or TRUE{r_p(x;v)} TRUE. The second kind is the assumption for an application of D5 to derive a goal below it (i.e. a goal of which it is a subgoal). Thus the final subgoal tree is a proof of TRUE {A} TRUE.

(e) P{q(x;v)}P is provable if ¬(x∈VAR(P)).

This follows from lemma d by applying the adaptation rule (D7):

1. TRUE $\{q(x;v)\}$ TRUE — lemma d.

2. TRUE $\wedge(\forall x)(\text{TRUE}\supset P)\{q(x;v)\}$ P — D7,1.

3. $P\{q(x;v)\}P$ — D1,2 since x does not occur in TRUE or in P (by assumption).

This establishes that C2 can always be replaced in a CORE proof by a derivation in Hoare's system. To eliminate C6 from a CORE proof we argue as fol lows. Suppose a given proof contains an application of AND-OR, without loss of generality, let us say it is the final deduct ion. We show that this occurrence of AND-OR can be either el iminated al together or "moved up" the proof tree in the sense that it is rep laced by an AND-OR application to the premisses of the premisses of the original application, This gives us a new proof containing on ly expressions that are in the old proof. We show further that in the second case where the rule is "moved up", if the moving up procedure is repeated the rule will never again need to be applied in any new proof to the same pair of premisses it was applied to originally. Since the given proof contains a finite number of expressions this establishes that our moving up procedure terminates with a proof in which all applications of AND-OR have disappeared.

(f) LEMMA

There is a constructive procedure for eliminating applications of the AND-OR rule from CORE proofs.

PROOF.

Suppose a given CORE proof contains one deduction by AND-OR of, the form

```
                 H1,H2   H3,H4    (rule R)
                 -----   -----
D.                 I       J      (AND-OR)
                   ---------
                       K
```

where R is not AND-OR.

We give a procedure whereby either

(a) D can be replaced by a deduction of K from axioms by the rule 0 f consequence,

or

(b) D can be replaced by

D1.
$$\dfrac{\dfrac{H1', H3'}{I1} \qquad \dfrac{H2', H4'}{J1}}{K} \qquad \begin{array}{l}\text{(AND-OR)}\\ \text{(rule R)}\end{array}$$

In case (b), for each i, the subproof Hi' in D1 contains only statements occurring in the subproof Hi in D. Repeated application of the procedure cannot result in (AND-OR) being applied to the pair I, J of premisses again.

We note that since the same program part must appear in both premisses of an application of AND-OR, the immediately preceding rules deducing those premisses must either be the same rule R or one of them must be the rule of consequence.

Let us consider the AND-case of this rule first. We give the replacement procedure for different cases of rule R:

(i) AXIOMS.

An application of AND-OR to axioms

$$\dfrac{P|^{x}_{e}\ \{x\leftarrow e\}P \qquad\qquad R|^{x}_{e}\ \{x\leftarrow e\}R}{P|^{x}_{e} \wedge R|^{x}_{e}\ \{x\leftarrow e\}P\wedge R}$$

is eliminated entirely and replaced by the axiom

$$(P\wedge R)|^{x}_{e}\ \{x\leftarrow e\}\ P\wedge R$$

Applications of AND-rule to frame axioms are eliminated similarly.

(ii) CONSEQUENCE.

An occurrence of AND-OR of the form

$$\dfrac{\dfrac{P\{A\}Q1,\ Q1\supset Q}{P\{A\}Q} \quad , \quad R\{A\}S}{P\wedge R\ \{A\}Q\wedge S}$$

is replaced by

$$\frac{\dfrac{P\ \{A\}Q1,\ R\{A\}S}{P\wedge R\{A\}Q1\wedge S} \quad , \quad Q1\wedge S\supset Q\wedge S}{P\wedge R\{A\}Q\wedge S}$$

The other cases (omitted) are similar.

(iii) WHILE

$$\frac{\dfrac{P\wedge U\{A\}P,\quad (P\wedge\neg U)\supset Q}{P\{\text{while } U \text{ do } A\}Q} \qquad \dfrac{R\wedge U\{A\}R,\quad (R\wedge\neg U)\supset S}{R\{\text{while } U \text{ do } A\}S}}{P\wedge R\{\text{while } U \text{ do } A\}Q\wedge S}$$

is replaced by

$$\frac{\dfrac{P\wedge U\{A\}P,\ R\wedge U\{A\}R}{(P\wedge R)\wedge U\ \{A\}\ (P\wedge R)} \quad , \quad (P\wedge R)\wedge\neg U\supset(Q\wedge S).}{P\wedge R\{\text{while } U \text{ do } A\}Q\wedge S}$$

(iv) CONDITIONAL

$$\frac{\dfrac{P\wedge U\ \{A\}\ Q,\ P\wedge\neg U\ \{B\}\ Q}{P\{\text{if } U \text{ then } A \text{ else } B\}Q,} \dfrac{R\wedge U\{A\}\ S,\ R\wedge\neg U\{B\}\ S}{R\{\text{if } U \text{ then } A \text{ else } B\}S}}{P\wedge R\{\text{if } U \text{ then } A \text{ else } B\}Q\wedge S}$$

is replaced by

$$\frac{\dfrac{P\wedge U\{A\}Q,\ R\wedge U\{A\}\ S}{(P\wedge R)\wedge U\ \{A\}Q\wedge S} \quad , \quad \dfrac{P\wedge - U\ \{B\}Q,\ R\wedge\neg U\{B\}\ S}{(P\wedge R)\wedge - U\ \{B\}Q\wedge S}}{P\wedge R\{\text{if } U \text{ then } A \text{ else } B\}Q\wedge S}$$

Clauses for Composition and Substitution are similar to (iii) and (iv) and are omitted.

(v) PROCEDURE CALL

Procedure p has body K(p).

$$\frac{P\{r\}Q\ ||\!-\ P\{K(r)\}Q}{P\ \{p\}Q} \quad , \quad \frac{R\{r\}S\ ||\!-\ R\{K(r)\}S}{R\{p\}\ S}$$

```
     ------------------------------------------
                   P∧R{p}Q∧S
```

is replaced by

```
      P{r}Q||-P{K(r)}Q       R{r}S||-R{K(r)}S
      ----------------       ----------------
           P{r2}Q                 R{r2}S
         [subproof]             [subproof]
         P{K(r2)}Q      ,       R{K(r2)}S
         -------------------------------
 P∧R {r2}Q∧S||-    P∧R{K(r2)}Q∧S
 -------------------------------
            P∧R{p}Q∧S
```

This last transformation rule requires a word of explanation. In the replacement, the AND-OR rule has been "pushed up" and applied to assertions on K(p) instead of assertions on call p. The procedure call rule is now applied to P∧R{K(r2)}Q∧S so that the relevant assumption is P∧R{r2}Q∧S. Subproofs for P{K(r2)}Q and R{K(r2)}S have to be appended: the given procedure rule applications ensure the existence of these subproofs, For example, we know there is a subproof of P{K(r)}Q from the assumption P{r}Q; an application of the CALL rule allows us to deduce P {r2}Q, where r2 is a new name for procedure p. The assumption P{r}Q is discharged at this point. We then repeat the subproof again with r2 replacing r everywhere. However, no assumption is necessary in this repetition since P{r2}Q is proved. Thus, the complete subproof trees for the premisses of the new AND-OR application contain copies of the given auxilliary subproofs at "assumption nodes". The statements in each new tree are exactly those of the old tree except possibly for r2 in place of r. If the replacement procedure is applied to this new subproof of P∧R{K(r2)}Q∧S, the AND-OR rule need not be applied to the same pair of hypotheses (with r2 for p) again since P∧R{r2}Q∧S is now assumed true.

This completes the description of 'the replacement procedure for AND; the OR case contains almost identical clauses except that the replacements in cases (iii) and (iv) contain intermediate -applications of consequence: (P∨R)∧U⊃(P∧U)∨(R∧U).

We note that Lemma f shows also that the AND-OR rule can also be omitted from the CORE. In the presence of the other core rules, ADAPTATION may be replaced by the FRAME axioms. The previous discussion may be summarized by the following theorem:

g. THEOREM

If ||-P{A}Q then P{A'}Q is provable from the CORE where A' is equivalent to A. Conversely if P{A}Q is provable from the CORE then , |-P{A}Q.

3.2 A MODEL FOR THE CORE

We assume given a standard model M for the theory T of the true Boolean expressions of Pascal and a set J of procedure definitions. Essentially M is the standard model for arithmetic possibly augmented by standard models for data types other than the integers. The details of M itself do not concern us. We show how to extend M to a model M* for the CORE.

To simplify the notation we assume a fixed ordering of the variables $x_1, x_2, x_3, \ldots$ This allows us to represent computation state vectors over the domain D of M by infinite sequences of elements of D, $a = <a_1, a_2, a_3 \ldots>$. D* shall denote the set of all such sequences. Intuitively, state a assigns the value or interpretation a_i to x_i; this is denoted by $(x_i)_I$, The interpretation or value t_I of Boolean expressions t is defined in the usual way from standard interpretation of the primitives +,*,etc. The value of t_I applied to state a will be denoted by $t_I(a)$. A Boolean expression of n variables, say $P(x_1, \ldots, x_n)$, is interpreted in M as a subset P_M of D^n. Thus $P(x_1, \ldots, x_n)$ is true for the state vector a if $<a_1, \ldots, a_n> \epsilon P_M$.

This allows us to extend the interpretation of $P(x_1, \ldots x_n)$ to D*:

$$P(x_1, \ldots, x_n)_I = \{a | <a_1, \ldots, a_n> \epsilon P_M\}.$$

Moreover, the interpretation of substitution instances by definition satisfies:

$$a \epsilon (P(x_1, \ldots, x_n)|^{xi}_{e})_I \iff <a_1, \ldots, a_{i-l}, e_I(a), a_{i+l} \ldots> \epsilon P(x_1 \ldots x_n)_I .$$

The- interpretation of an (m,n)-ary procedure is a partial function f o'f the type $N^m \times D^n \rightarrow (D* \rightarrow D*)$ having the following properties:

1) Frame property:

$$(f(i(1), \ldots, i(m); c_1, \ldots, c_n)(a))_j = a_j ,$$

j is different from i(k) for any k such

that $1 \le k \le m$.

2) Substitution property:

$$(f(i(1),\dots,i(m);c_1,\dots,c_n)(a))_{i(k)}$$
$$= (f(j(1),\dots,j(m);c_1,\dots,c_n)(a))_{j(k)},$$
$1 \le k \le m$.

The definition of f proceeds as follows.

We define by cases the computation sequence $F(A,a)$ of program A relative to M given input a as follows.

If a is an infinite state vector, then:

(i) $F(x_i \leftarrow e, a) = \langle a_1,\dots,a_{i-1}, e(a), a_{i+1},\dots\rangle$

(ii) $F(A;B,a) = F(A,a) @ F(B,U(A,a))$

(iii) $F(\text{if } P(x_1,\dots x_n) \text{ then } A \text{ else } B, a) = \begin{cases} F(A,a) & \text{if } \langle a_1,\dots a_n\rangle \in P_I \\ F(B,a) & \text{otherwise.}\end{cases}$

(iv) $F(q(z;t),a) = a@F(K(z;t),a)$ where J contains a defining axiom for q of the form "$q(x;v)$ PROC $K(x;v)$" and $K(z;t)$ is obtained by sutstituting the actual parameters z, t for the formal. parameters x, v.

Here asb is the sequence obtained by appending b onto the end of a,

$$U(A,a) = \begin{cases} \text{end state of } F(A,a) \text{ if } F(A,a) \text{ is finite} \\ \text{undefined otherwise.}\end{cases}$$

The interpretation of program A is now defined:

$$A_I = \{\langle a,b\rangle \mid U(A,a) = b\}$$

and M is extended to M* by adding the function A_I for each Pascal CORE program A.

We can now say when a statement of the form P{A}Q is true in M* (denoted by M*|=P{A}Q):

$$M^* \models P\{A\}Q \Leftrightarrow A_I(P_I) \subseteq Q_I .$$

Finally, a statement $S(r_1,\dots,r_m)$ with assumptions $A_1(r_1,\dots,r_m),\dots,$ $A_n(r_1,\dots,r_m)$ where $r_1,\dots,r_m$ are free procedure variables, is true in M^* if and only if the following condition holds:

> If $A_1(p_1,\dots,p_m),\dots A_n(p_1,\dots,p_m)$ are true for any declared procedure names $p_1,\dots,p_m$ from J, each p_i having the same arity as r_i $(1 \le i \le m)$, then $S(p_1,\dots,p_m)$ is true.

Here are some simple properties of this model:

(i) If the range of A_I is empty then for any P and $Q, M^* \models P\{A\}Q$

(ii) If $M^* \models P\{K(q)\}Q$ then $M^* \models P\{q\}Q$ where K is the body of procedure q.

(iii) If p PROC $K(r)$ and q PROC $K(s)$ and $r_I \subseteq s_I$ then $p_I \subseteq q_I$.

(iv) A Boolean assertion is true in M^* if and only if its universal closure is true in M.

To show that M^* is a model for the CORE we will show that the axioms are true in M^* and that each of the rules of inference preserves' truth (i.e. if the premisses of the rules are true in M^* then so also are the conclusions). For simplicity we consider examples of the axioms and rules in which the statements have one free variable (three variables for the substitution rule) and in which the premisses do not have governing assumptions except in the case of the recursion rule: the argument for the general case is identical.

Consider first a typical assignment axiom $P(e)\{x_1 \leftarrow e\}P(x_1)$.

We note that $(x_1 \leftarrow e)_I = \{<a,b>: b=<e_I(a), a_2, a_3, \dots>\}$, and that $a \in P(e)_I \Leftrightarrow <e(a), a_2, \dots> \in P(x_1)_I$. Thus $(x \leftarrow e)_I(P(e)_I) \subseteq P(x_1)_I$ so 'that the assignment axiom is true in M^*.

The frame axioms are clearly true in M^*: if P does not contain x_1, say, and a, b differ only at the first position, then $a \in P_I \Leftrightarrow b \in P_I$. If $q(x_1, v)$ changes only the value of x_1 then $q_I(P_I) \subseteq P_I$.

Logical theorems are true in M* since they are true in M. Procedure&declaration axioms are assumed to be in J.

We consider next the rules of inference. The fact that Consequence, Composition and Conditional all preserve truth in M* can be shown by elementary set theoretic arguments on the interpretations of Boolean expressions and programs. Simply note that if $P \supset Q$ is true in M* then $P_I \subset Q_I$, that $(P \wedge R)_I = P_I \cap R_I$, and that $\neg R_I = D^* - R_I$.

The arguments are as follows:

CONSEQUENCE: If $P_I \subset Q_I$ and $A_I(Q_I) \subset R_I$ then $A_I(P_I) \subset R_I$.

COMPOSITION: If $A_I(P_I) \subset Q_I$ and $B_I(Q_I) \subset R_I$ then $B_I(A_I(P_I)) \subset R_I$.

CONDITIONAL: If $A_I(P_I \cap R_I) \subset Q_I$ and $B_I(P_I \cap \neg R_I) \subset Q_I$ then $(\text{if } R \text{ then } A \text{ else } B)_I(P_I) \subset Q_I$

SUBSTITUTION

Consider the case when the procedure $g(x_1,x_2;x_3)$ has two left parameters and one right parameter since this is sufficiently general. Let q have body K. Assume that x_1 and x_2 are the only variables whose values can be changed by K_I, and that x_3 is the only value that its computation depends on. We require a simple lemma which may be proved by induction on the composition of K.

h. LEMMA.

For any a if $K(x_1,x_2;x_3)_I(a)=b$ and $K(x_i\, x_j;x_3)_1(a)= c$ then $b_1 = c_i$ and $b_2 = c_j$ provided $i \neq j \neq 3$.

Let f, g be partial functions mapping D* into D such that $K(x_1,x_2;x_3)_I(a) = \langle f(a_3), g(a_3), a_3 \ldots\rangle$ and hence also $K(x_4,x_5;x_3)_I(a) = \langle a_1, a_2, a_3, f(a_3)\ g(a_3), \ldots\rangle$. If the premisses of the substitution rule are true, then:

$$a \in P(x_1\, x_2\, x_3)_I \text{ implies } \langle f(a_3), g(a_3), a_3, \ldots\rangle \in Q(x_1\, x_2\, x_3)_1$$