Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages

Unrestricted Procedure Calls in Hoare's Logic†

Robert Cartwright
Computer Science Department
Cornell University, Ithaca, N.Y. 14853

Derek Oppen
Computer Science Department
Stanford University, Stanford, Ca. 94305

Abstract

This paper presents a new version of Hoare's logic including generalized procedure call and assignment rules which correctly handle aliased variables. Formal justifications are given for the new rules.

1. Introduction

Despite the widespread acceptance of Hoare's logic as the most suitable formalism for verifying programs written in procedural languages, the logic still suffers from several shortcomings. In our view, the two most significant problems are:

1. The limitation on programming language constructs, particularly procedures, imposed by commonly accepted rules of the logic.

2. The absence of formal justifications for most of these rules.

Of the sets of Hoare proof rules proposed in the literature [Hoare 69] [Hoare and Wirth 73] [London, et al. 77] [Igarashi, London and Luckham 75] [Donahue 76] [Cook 75] [Gorelick 75], none is sound for all programs in any reasonably complicated procedural language such as PASCAL. In particular, all of the rules place significant restrictions on procedures and procedure calls. Even the most comprehensive procedure call rule proposed to date (for EUCLID by [Guttag, et al. 77]) must:

1. Prohibit aliasing in procedure calls.

2. Disallow passing procedures and functions as parameters.

3. Require that value parameters be read-only (i.e. constant parameters).

4. Prohibit declaring a procedure within a procedure of the same name.

5. Require that the global variables accessed by a procedure be accessible at the point of every call.

Another disturbing aspect of current versions of Hoare's logic is that most of the proposed rules have never been formally justified. Perhaps as a result, many of the rules have counter-examples. The few sets of rules that have been proved sound apply only to idealized programming languages [Cook 75] [Gorelick 75] [Oppen 75] [Donahue 76]. The proof rule systems proposed for practical programming languages such as PASCAL [Hoare and Wirth 73] and EUCLID [London, et al. 77] have never been formally justified, and all contain serious errors and omissions.

As a step toward solving these problems, we will concentrate on devising and rigorously justifying rules for procedure calls in PASCAL -- including calls involving aliasing. First, we will develop a simple procedure call rule (patterned after [Hoare 71] along the same lines as the EUCLID rule [Guttag, et al. 77]) for calls where no aliasing is present. Next, we will propose generalized assignment and procedure call rules for contexts where aliasing is permitted. Both generalized rules collapse to the corresponding simple rules if no aliasing is present.

We will sketch soundness and relative completeness proofs for all of our new rules. In the process, we will propose a new mathematical definition for the meaning of statements in Hoare's logic which seems sufficiently comprehensive to handle most procedural languages, yet is intuitively tractable.

2. Mathematical Foundations

Before we can formulate and justify our proof rules, we must establish the mathematical foundations for our version of Hoare's logic. We introduce three sets of definitions.

2.1 State Vectors and Access Sequences

From an informal viewpoint, a *state vector* is a sequence of bindings of program variables to data values, and procedure names to procedure bodies (e.g. a LISP association list). An *access sequence* is a canonical name for an entry in a state vector. For example, the access sequence

† This research has been partially supported by National Science Foundation Grants MCS76-14293 and MCS76-000327.

for the variable x is <'x> (since x typically means the value of the variable x, we use the notation 'x to refer to the variable itself). The access sequence for the array element a[l] is <'a,l>.

More formally, we let D denote the set of data values that program variables may assume, and let I and I' denote the set of program identifiers a,b, c,...,and quoted program identifiers 'a,'b,'c,..., respectively. We let B denote the set of procedure bodies. A variable-specifier is any legal left-hand side of an assignment statement. A simple variable is a variable-specifier consisting of a single identifier. For example, a[x] and x are both variable-specifiers; x is a simple variable, but a[x] is not.

For the sake of simplicity, we limit our attention to a subset of PASCAL restricting the set of variable-specifiers to simple variables and singly subscripted array references. Similarly, we assume the data value domain for our PASCAL dialect has the form $[\bigcup_{j\in J} D_j] \cup [\bigcup_{j,k\in J} (D_j \rightarrow D_k)]$ where the sets $D_j, j\in J$, are disjoint sets of primitive data objects (e.g. integers, characters, booleans) and $(D_j \rightarrow D_k)$ denotes the set of mappings (arrays) from D_j into D_k. We call each set D_j and $(D_j \rightarrow D_k)$ a type. These restrictions are made only for explanatory purposes. All of our results generalize to arbitrary PASCAL data domains.

We define the access sequence corresponding to the simple variable v as the singleton sequence <'v>. For a variable-specifier of the form a[e] (where a is an array and e is an expression), the access sequence is $<'a,e_0>$ where $e_0 \in D$ is the value of e. We define two access sequences to be disjoint if and only if neither is an initial segment of the other.

Let H be a finite set of variable declarations v : T_v(where v is a program identifier and T_v is a type) and procedure declarations procedure $p(\alpha_p)$; B_p (where p_j is a program identifier,α_p is a sequence of var and value parameter declarations and B_p is the remainder of the procedure body). We call H a declaration set. A state vector s consistent with H is a mapping from I (identifiers) into D (data values) ∪ B (procedure bodies) such that each variable v declared in H is bound to a data value of type T_v and each procedure p is bound to the body procedure $p(\alpha_p)$; B_p.

Typically, we are only interested in only a finite restriction of s -- specifically the bindings of the variables and procedure names declared in H. In this case, we can think of s as a finite sequence of ordered pairs (x,d) where x is a program identifier declared in H and d is its binding.

We let A and S denote the set of access sequences and the set of state vectors respectively.

2.2 Value and Update Functions

We introduce two functions Value and Update to access and modify states, analogous to the array access and update functions defined by [McCarthy 63]. Value maps a state vector s and an access sequence α into the binding of α in s. Update maps a state vector s, an access sequence α, and a value d into the state vector s' where s' is identical to s except that the entry within s' specified by α has the new value d.

In more formal terms, Value is a mapping from SxA into D∪B and Update is a mapping from SxAx(B∪D) into S satisfying the following axioms:

1. Value(Update(s,α,e),α)=e for arbitrary state vector s, access sequence α, and value e, provided the entry specified by α exists in s.
2. Value(Update(s,α_1,e),α_2) = Value(s,α_2) if α_1, and α_2 are disjoint access sequences and the entries specified by α_1 and α_2 exist in s.
3. Let Select be the standard array access function mapping $(D_i \rightarrow D_j) \times D_i$ into D_j for all i,j. Then Value(Update(s,<'v>,d),<'v,e>) = Select(d,e) for arbitrary state vector s, identifier v, array value d, and data value e, provided e is in the domain of d.
4. Let Store be the standard array update function mapping $(D_i \rightarrow D_j) \times D_i \times D_j$ into $(D_i \rightarrow D_j)$ for all i,j. Then Value (Update(s,<'v,e>, d), <'v>) = Store(Value(s,<'v>),e,d) for arbitrary state vector s, identifier v, and data values d and e, provided e and d belong to the domain and range of Value (s,<'v>) respectively.

We extend Value and Update to apply to sequences of disjoint access sequences as follows:

1. Value*(s,<$\alpha_1,\ldots,\alpha_n$>) = <Value(s,α_1),..., Value(s,α_n)> for arbitrary state vector s and access sequences $\alpha_1,\ldots,\alpha_n$, provided the entries specified by $\alpha_1,\ldots,\alpha_n$ exist in s.
2. Update*(s,<$\alpha_1,\ldots,\alpha_n$>, <$d_1,\ldots,d_n$>) = Update(...Update(s,α_1,d_1)...,α_n,d_n) for arbitrary access sequences $\alpha_1,\ldots,\alpha_n$ and values $d_1,\ldots,d_n$ provided the specified updates are well-defined.
3. Let $\alpha_1,\ldots,\alpha_n$ be disjoint access sequences such that $\alpha_{i_1},\alpha_{i_2},\ldots,\alpha_{i_k}$ have the form <'v,e_ℓ>,ℓ=1,2,...,k, where 'v is an identifier; and e_ℓ is a data value. Let $\alpha_{j_1},\alpha_{j_2},\ldots,\alpha_{j_{n-k}}$ be the remaining access sequences, and let d denote Value(s,<'v>). Then Update*(s,<$\alpha_1,\ldots,\alpha_n$>,<$d_1,\ldots,d_n$>) = Update*(s,<$\alpha,\alpha_{j_1},\alpha_{j_2},\ldots,\alpha_{j_{n-k}}$>, <Store(...Store(d,$e_1$,$d_{j_1}$)...,$e_k$,$d_{i_k}$),$d_{j_1}$, ...,$d_{j_{n-k}}$>) provided the specified updates are well-defined.

The final axiom above merely collects updates to various elements of the same array and combines them into a single update of the entire array. We can use this axiom to convert an arbitrary sequence of disjoint updates to an equivalent set of simple updates (i.e. updates of simple variables rather than array elements). For example,

$$\underline{\text{Update}}^*(s, \langle\langle 'a,1\rangle, \langle 'b\rangle, \langle 'a,4\rangle, \langle 'c\rangle\rangle, \langle 1,2,3,4\rangle) =$$
$$\underline{\text{Update}}^*(s, \langle\langle 'a\rangle, \langle 'b\rangle, \langle 'c\rangle\rangle, \langle \underline{\text{Store}}(\underline{\text{Store}}(\underline{\text{Value}}(\langle 'a\rangle, s), 1, 1), 4, 3), 2, 4\rangle.$$

We denote the set of sequences of access sequences by A^*.

2.3 Definition of Truth

In this section, we define the syntax and meaning of statements in our version of Hoare's logic.

2.3.1 The Base Logic

We assume we are given a base first order theory (L,M) (for the program data domain), consisting of a logical language L with equality and a model M for L, with the following properties:

1. The domain of the model M includes D (data values), I' (quoted identifiers), A (access sequences), A^* (sequences over A), and B (procedure bodies).
2. The variables of L include two disjoint sets: I (programming language identifiers) and V, a set of logical variables which may not appear within programs.
3. The logic includes the binary function $\oplus$ and the unary function Seq. The $\oplus$ operator concatenates two sequences, i.e. $\langle u_1,\ldots,u_m\rangle \oplus \langle v_1,\ldots,v_n\rangle = \langle u_1,\ldots,u_m,v_1,\ldots,v_n\rangle$.

   Seq maps a data object d (specifically a quoted identifier, a data value, or an access sequence) into the singleton sequence $\langle d\rangle$. With the functions $\oplus$ and Seq, we can construct arbitrary members of A and A^*.
4. The logic includes all the primitive functions of programming language including array access and update functions (Select and Store). We let a[e], where a is an identifier and e is a term, abbreviate the term Select (a,e).
5. The logic includes a characteristic predicate P_T for each data type T in D. We will use the familiar notation x : T to abbreviate $P_T(x)$.
6. The logic includes the predicates Disjoint and Pair-disjoint with domains A^* and A^*xA^* respectively. Disjoint $(\langle\alpha_1,\ldots,\alpha_n\rangle)$ is true if and only if access sequences α_i and α_j are disjoint for all i,j such that $i \neq j$. Pair-disjoint $(\langle\alpha_1,\ldots,\alpha_m\rangle, \langle\beta_1,\ldots,\beta_n\rangle)$ is true if and only if α_i and β_j are disjoint for all i,j.

Given an arbitrary variable specifier v, we can construct a term v^* in L such that the meaning of v^* is the access sequence for v. If v is a simple variable x, then v^* is simply Seq('x). If v is an array element a[e], then v^* is Seq('a) $\oplus$ Seq(e). We will frequently employ this construction in our proof rules.

2.3.2 Extended Terms and Formulas

For the sake of clarity, we prohibit formulas of L from using program identifiers as bound (quantified) variables. In addition, to conveniently handle updates to the state vector, we extend the logical language L to include updated formulas and terms. We define an extended formula (term) of L as follows. An extended formula (term) has a recursive definition identical to that of an ordinary formula (term) [Enderton 72] except that there is an additional mechanism (called an update) for building new formulas and terms from existing ones. Given an extended formula (term) α, the form $[\![\bar{v} \leftarrow \bar{t}]\!]\alpha$ is also an extended formula (term), where $\bar{v}$ is a sequence of disjoint variable-specifiers and $\bar{t}$ is a corresponding sequence of ordinary (not updated) terms in L. We will call $[\![\bar{v} \leftarrow \bar{t}]\!]$ a simultaneous update. Henceforth, we will simply use the term formula (term) to refer to an extended formula (extended term).

2.3.3 Hoare Assertions and Statements

Let Q be an arbitrary formula in L and let $x_1,\ldots,x_n$ be the program identifiers which occur in Q. Let H be a declaration set including declarations for $x_1,\ldots,x_n$. A Hoare assertion has the form

$$H|Q.$$

Let A be a program segment and P and Q be formulas in L. Let H be a declaration set including declarations for all the free program variables and procedure names in A, P, and Q. A Hoare statement has the form

$$H|P\{A\}Q$$

We define the meaning of Hoare assertions and statements as follows. Let H|Q be an arbitrary Hoare assertion. The definition of truth for H|Q is identical to the standard first-order definition of truth for Q [Enderton 72] except:

1. H|Q is vacuously true for states inconsistent with H.
2. The meaning of the updated formula (term) $[\![\bar{v} \leftarrow \bar{t}]\!]\ \alpha$ for state s is the meaning of the formula (term) α for state Update*$(s,\bar{v}^*,\bar{t})$ where $\bar{v}^*$ denotes the sequence of access sequences corresponding to $\bar{v}$.

Let H|P{A}Q be an arbitrary Hoare statement and let Eval be an interpreter (a partial function) mapping states x program-segments into states. Then H|P{A}Q is true if and only if for all states s either

1. H|P is false for s.
2. Eval(s,A) is undefined.
3. Q is true for Eval(s,A).

2.3.4 Standard Proof Rules

The standard simple Hoare proof rules have obvious analogs in our version of the logic.

The most fundamental rules -- consequences, composition, and substitution -- have the following form:

1. Consequence $\dfrac{H|P \supset Q,\ H|Q\ \{A\}\ R,\ H|R \supset S}{H|P\ \{A\}\ S}$

2. Composition $\dfrac{H|P\{A\}Q,\ H|Q\{B\}R}{H|P\{A;B\}R}$

3. Substitution $\dfrac{H|P\{A\}Q}{H|P(t/x')\ \{A\}\ Q(t/x')}$

   where x' is a logical variable and Q(t/x') denotes Q with every free occurrence of x' replaced by t (renaming bound variables).

The other standard rules which we will take as given are:

4. Declaration

$$\dfrac{H(\bar{x}'/\bar{x},\bar{p}'/\bar{p}) \cup \{\overline{x:T},\overline{p:B}\}\ |\ P(\bar{x}'/\bar{x})\{A\}Q(\bar{x}'/\bar{x})}{H|P\ \{\underline{begin}\ \overline{x:T};\ \overline{p:B};\ A\ \underline{end}\}\ Q}$$

   where $\overline{x:T}$ and $\overline{p:B}$ are sequences of variable and procedure declarations, and $\bar{x}'$ and $\bar{p}'$ are sequences of fresh program variables and procedure names corresponding to $\bar{x}$ and $\bar{p}$.

5. Simultaneous Assignment $\dfrac{H|\mathrm{Disjoint}(\bar{v}^*)}{H|[\![\bar{v}\leftarrow\bar{t}\,]\!]P\{\bar{v}\leftarrow\bar{t}\}\ P}$

   where $\bar{v}\leftarrow\bar{t}$ is a disjoint assignment and where $\bar{v}^*$ is the sequence of access sequence terms in L corresponding to $\bar{v}$.

2.3.5 Reasoning about Updated Formulas

In order to prove Hoare assertions involving updated formulas, we need special axioms about updates. For disjoint updates modifying entire formulas, the following axioms (derived from the corresponding axioms for Update*) are sufficient:

1. $[\![\,\bar{x} \leftarrow \bar{t}\,]\!]Q \equiv Q(\bar{t}/\bar{x})$ where $\bar{x}$ is a sequence of distinct simple variables, and Q is a formula containing no updates.

2. Let $v_1,\ldots,v_n$ be disjoint variable specifiers where $v_{i_1},\ldots,v_{i_k}$ have the form $a[e_\ell]$, $\ell=1,\ldots,k$, where a is a particular array identifier. Let $v_{j_1},\ldots,v_{j_{n-k}}$ be the remaining variable-specifiers. Let $\bar{v}'$ denote the sequence of variable-specifiers $a,v_{j_1},\ldots,v_{j_{n-k}}$ and let $\bar{t}'$ denote the sequence of terms $<\mathrm{Store}(\ldots\ \mathrm{Store}(a,e_1,t_{i_1})\ \ldots,e_k,t_{i_k}),t_{j_1},\ldots,t_{j_{n-k}}>$. Then

$$[\![\,\bar{v} \leftarrow \bar{t}\,]\!]\ Q \equiv [\![\,\bar{v}' \leftarrow \bar{t}'\,]\!]Q.$$

Given an arbitrary disjoint simultaneous update $\bar{v} \leftarrow \bar{t}$, we can eliminate the update $[\![\bar{v} \leftarrow \bar{t}\,]\!]$ from a formula of the form $[\![\bar{v} \leftarrow \bar{t}\,]\!]Q$ where Q is update free by using axiom 2 to eliminate all assignments to array elements and then applying axiom 1. We can similarly eliminate all updates from a formula of the form $[\![\ldots]\!]\,[\![\bar{v} \leftarrow \bar{t}\,]\!]Q$ where Q is update-free by repeatedly applying the same simplification procedure.

3. Simple Procedure Call Rule

In this section we assume that our PASCAL subset:

1. Prohibits aliasing in procedure calls.
2. Disallows passing procedures and functions as parameters.
3. Requires that the global variables accessed by a procedure be explicitly declared at the head of the procedure and that these variables be accessible at the point of every call.

Under these assumptions, it is straightforward to formulate a procedure call rule by treating procedure calls as simultaneous assignments to the variables passed to the procedure. The assigned values are any values consistent with the input-output assertions for the procedure.

Let p be declared as $\underline{\text{procedure}}\ p(\underline{\text{var}}\ \overline{x:T_x};\ \underline{\text{value}}\ \overline{y:T_y});\ \underline{\text{global}}\ \bar{z};\ B$ in the declaration set H. B may not access any global variables other than $\bar{z}$. Let H' be H augmented by the declarations $\overline{x:T_x}$ and $\overline{y:T_y}$ (prior declarations of $\bar{x}$ and $\bar{y}$ are replaced). Let P and Q be formulas containing no free program variables other than $\bar{x},\bar{y},\bar{z}$ and $\bar{x},\bar{z}$ respectively. Let $\bar{v}$ be the free logical variables of P and Q, and let $\bar{x}'$ and $\bar{z}'$ be fresh logical variables corresponding to $\bar{x}$ and $\bar{z}$. Then the (non-recursive) simple procedure call rule has the following form:

$$\dfrac{H\ |\ \mathrm{Disjoint}(\bar{a}^*\oplus\bar{z}^*),\quad H'|\ P\{B\}\ Q \qquad H|\forall\bar{v}[P(\bar{a}/\bar{x},\bar{b}/\bar{y})\supset Q(\bar{x}'/\bar{x},\bar{z}'/\bar{z})]\supset[R\supset[\![\bar{a},\bar{z}\leftarrow\bar{x}',\bar{z}']\!]S]}{H|R\ \{p(\bar{a};\bar{b})\}\ S}$$

It is important to note that the free logical variables $\bar{x}'$ and $\bar{z}'$ in the second premise are implicitly universally quantified. The rule forces $R\supset [\![\bar{a},\bar{z} \leftarrow \bar{x}',\bar{z}'\,]\!]S$ to be true for arbitrary $\bar{x}'$ and $\bar{z}'$ consistent with $\forall\bar{v}[P(\bar{a}/\bar{x},\bar{b}/\bar{y})\supset Q(\bar{x}'/\bar{x},\bar{z}'/\bar{z})]$. In contrast, the EUCLID procedure call rule explicitly omits the corresponding quantifier -- permitting false deductions.

3.1 Soundness

If Eval is properly defined, it is easy to prove the soundness of the simple procedure call rule. Let s be an arbitrary state, consistent with H such that H|R is true for s and Eval(s, $p(\bar{a};\bar{b})$) is defined. We must show S is true for Eval(s,$p(\bar{a};\bar{b})$). Let s' be $[\![\bar{x}',\bar{z}' \leftarrow \bar{x}_0,\bar{z}_0\,]\!]$s where $\bar{x}_0,\bar{z}_0$ are the output values of $\bar{x}$ and $\bar{z}$ in the call $p(\bar{a};\bar{b})$ (i.e. the values of $\bar{x}$ and $\bar{z}$ in the state Eval ($[\![\bar{x},\bar{y} \leftarrow \bar{a},\bar{b}\,]\!]$s,B)). Since s' satisfies both $\forall\bar{v}[P(\bar{a}/\bar{x},\bar{b}/\bar{y})\supset Q(\bar{x}'/\bar{x},\bar{z}'/\bar{z})]$ and R in the second premise, s' must also satisfy $[\![\bar{a},\bar{z} \leftarrow \bar{x}',\bar{z}'\,]\!]$S. By the definition of Eval,

$$\mathrm{Eval}\ (s,p(\bar{a},\bar{b})) = [\![\bar{a},\bar{z} \leftarrow \bar{x}_0,\bar{z}_0\,]\!]s = [\![\bar{a},\bar{z} \leftarrow \bar{x}',\bar{z}'\,]\!]s'.$$

Hence Eval (s,$p(\bar{a},\bar{b})$) satisfies S. Q.E.D.

Although the soundness of the procedure call rule does not depend on the third assumption listed above (the accessibility of the procedure globals at the point of every call), the assumption is necessary to prove that Eval obeys static scope rules. The natural definition of Eval (which we used in the soundness proof) employs dynamic scope rules. If the third assumption holds then static and dynamic scope rules are semantically equivalent.

3.2 Relative Completeness

It is also reasonably straightforward to prove that the simple procedure call rule is relatively complete for non-recursive programs in the sense of [Cook 75]. We assume that the assertion language L is expressive, i.e. that given an arbitrary assertion P in L and a program segment A the strongest post-condition Q of A given pre-assertion P is definable in L. To show that the rule is complete relative to the completeness of the other proof rules and the axiomatization of the extended base logic, it suffices to show that for any program segment A and post-assertion Q, the weakest liberal pre-condition P is provable. The proof proceeds by contradiction.

Assume $p'(\bar{a}';\bar{b}')$ is a procedure call for which the rule is not complete. Let $p(\bar{a};\bar{b})$ be the deepest procedure call in the evaluation of $p'(\bar{a}';\bar{b}')$ for which the simple procedure call is not complete. Let H be the declaration set at the point of the call, and let p be declared as $\underline{\text{procedure}}\ p(\underline{\text{var}}\ \overline{x:T_x};\text{value}\ \overline{y:T_y});\ \text{global}\ \bar{z};\ B$ in H. Let S be an arbitrary post-assertion for $p(\bar{a};\bar{b})$. We define Q' as the strongest post-condition for B given the pre-assertion $\bar{x},\bar{y},\bar{z} = \bar{x}_i,\bar{y}_i,\bar{z}_i$. By assumption $H'|P\{B\}Q'$ is provable. We define Q to be $\exists \bar{y}' Q'(\bar{y}'/\bar{y})$. By the rule of consequence, $H|P\{B\}Q$ must be provable. In addition, $R \equiv$

$$\forall \bar{x}',\bar{z}'[Q(\bar{a}/\bar{x}_i,\bar{b}/\bar{y}_i,\bar{z}/\bar{z}_i,\bar{x}'/\bar{x},\bar{z}'/\bar{z}) \supset [\![\bar{a},\bar{z} \leftarrow \bar{x}',\bar{z}']\!]S]$$

is clearly a provable pre-condition of the rule.

Assume R is not the weakest liberal precondition. Then there exists a state s consistent with H such that R is false and either $\underline{\text{Eval}}(s,p(\bar{a};\bar{b}))$ is undefined or S is true for $\underline{\text{Eval}}(s,p(\bar{a};\bar{b}))$. Let s' be $[\![\bar{x},\bar{y} \leftarrow \bar{a},\bar{b}]\!]s$. Either $\underline{\text{Eval}}(s,B)$ is undefined or Q is true for $\underline{\text{Eval}}(s',B)$. In the former case, $Q(\bar{a}/\bar{x}_i,\bar{b}/\bar{y}_i,\bar{z}/\bar{z}_i,\bar{x}'/\bar{x},\bar{z}'/\bar{z})$ must be false for all $\bar{x}',\bar{z}'$ since $Q(\bar{a}/\bar{x}_i,\bar{b}/\bar{y}_i,\bar{z}/\bar{z}_i)$ is false for all $\bar{x}$, $\bar{z}$. Hence R is true, generating a contradiction. In the other case, $Q(\bar{a}/\bar{x}_i,\bar{b}/\bar{y}_i,\bar{z}/\bar{z}_i,\bar{x}'/\bar{x},\bar{z}'/\bar{z})$ is true only for states with $\bar{x}'$ and $\bar{z}'$ equal to the values of $\bar{x}$ and $\bar{z}$ in $\underline{\text{Eval}}(s',B)$. But for such $\bar{x}'$ and $\bar{z}'$, $\underline{\text{Eval}}(s,p(\bar{a};\bar{b})) = [\![\bar{a},\bar{z} \leftarrow \bar{x}',\bar{z}']\!]s$. Consequently, $[\![\bar{a},\bar{z} \leftarrow \bar{x}',\bar{z}']\!]S$ is true for all states satisfying $Q(\bar{a}/\bar{x}_i,\bar{b}/\bar{y}_i,\bar{z}/\bar{z}_i,\bar{x}'/\bar{x},\bar{z}'/\bar{z})$ implying R is true. Again, we have a contradiction. Q.E.D.

3.3 A Sample Proof

Let's consider a simple example which less sophisticated procedure call rules handle incorrectly. Let swap be a standard integer variable swap procedure defined as follows:

```
procedure p(var x,y : integer);
begin
   pre x=xi ∧ y=yi;
   x,y ← y,x;
   post y=xi ∧ x=yi
end;
```

By the simultaneous assignment rule, we must show $x,y{:}\underline{\text{integer}}|x=x_i \wedge y=y_i \supset [\![x,y \leftarrow y,x]\!]\,y=x_i \wedge x=y_i$

to establish the declared pre and post-assertions for swap. By the $[\![\ ]\!]$ substitution axiom (axiom 1 in 2.3.5),

$$[\![x,y \leftarrow y,x]\!]\ y=x_i \wedge x=y_i \equiv x=x_i \wedge y=y_i$$

which is precisely the pre-assertion. Q.E.D.

Now let us consider a sample application of the procedure call rule. Assume we want to prove:

$$a{:}\underline{\text{array}}\ \underline{\text{integer}}\ \underline{\text{of}}\ \underline{\text{integer}},\ i{:}\underline{\text{integer}}|\ a[i]=a_o \wedge i=i_o\{p(a[i],i)\}\ a[i_o]=i_o \wedge i=a_o.$$

Let H denote $\{a{:}\underline{\text{array}}\ \underline{\text{integer}}\ \underline{\text{of}}\ \underline{\text{integer}},\ i{:}\underline{\text{integer}}\}$; P' denote the substituted pre-condition $a[i]=x_i \wedge i=y_i$; Q' denote the substituted post-condition $y'=x_i \wedge x'=y_i$; R denote $a[i]=a_o \wedge i=i_o$; and S denote $a[i_o]=i_o \wedge i=a_o$. By the simple procedure call rule, we must show

1. $H|\text{Disjoint}(\langle' a,i\rangle, \langle' i\rangle)$.
2. The correctness of the input-output assertions for the procedure body.
3. $H|\forall x_o,y_o\ [P'\supset Q'] \supset [R\supset [\![a[i],i \leftarrow x',y']\!]S]$.

Since 1) is trivial, and we have already proved 2), it suffices to prove 3). First we transform $[\![a[i],i \leftarrow x',y']\!]S$ into $[\![a,i \leftarrow \underline{\text{Store}}(a,i,x'),y']\!]S \equiv \text{Store}(a,i,x')[i_o]=i_o \wedge y'=a_o$.

Since $i=i_o$ by hypothesis in R,

$$S' \equiv \underline{\text{Store}}(a,i_o,x')[i_o] = i_o \wedge y'=a_o$$
$$\equiv x'=i_o \wedge y'=a_o.$$

By applying the equality hypotheses in R, we transform $x'=i_o \wedge y'=a_o$ into $x'=i \wedge y'=a[i]$ -- which an immediate consequence of $P'\supset Q'$ when x_i,y_i are instantiated as a[i] and i respectively. Q.E.D.

3.4 Handling Recursion

Our simple rule can be extended to handle mutually recursive procedures by generalizing Hoare's original approach to the problem [Hoare 71]. However, we must impose the following additional restriction on our PASCAL subset to ensure the soundness of the rule:

> No procedure named p may be declared within the scope of another procedure named p.

Our rule is not unique in this respect. Every other proposed procedure call rule+ requires an equivalent restriction. The restriction is necessary because the input-output specifications for a procedure p may be assumed for any procedure call within a procedure declared in the scope of p.

Let procedure $p_i(\underline{\text{var}}\ \overline{x_i:T_{x_i}};\ \underline{\text{value}}\ \overline{y_i:T_{y_i}};$ global $\bar{z}_i$; B_i, $i=1,2,\ldots,n$ be a sequence of procedure declarations at the head of some block. Let P_i and Q_i, $i=1,\ldots,n$ be assertions containing no free program variables other than $\bar{x},\bar{y}_i,\bar{z}_i$ and $\bar{x}_i,\bar{z}_i$ respectively. Let $\bar{v}_i$ be the free logical variables in P_i and Q_i. Let H be a declaration set containing the declarations of $p_1,\ldots,p_k$ and let H' denote H with these declarations

+ An exception is a rule currently under development by DeBakker, but his rule requires extensive rewriting of the program text.

replaced by "forward" procedure declarations which only specify the the procedures' formal parameters. Let H'_i denote H' augmented by the declarations $\overline{x:T_x}$, $\overline{y:T_y}$ (prior declarations of x and y are replaced). For i=1,...,n we define the recursion hypothesis I_i as the rule:

$$\begin{array}{l} H \mid \underline{\text{Disjoint}}(\bar{c}^* \oplus \bar{z}_i{}^*) \\ H \mid \forall \bar{v}_i [P_i(\bar{c}/\bar{x}_i, \bar{d}/\bar{y}_i) \supset Q_i(\bar{x}'_i/\bar{x}_i, \bar{z}'_i/\bar{z}_i)] \supset \\ \quad [\theta_1 \supset [\![\bar{c}, \bar{z}_i \leftarrow \bar{x}'_i, \bar{z}'_i]\!] \theta_2] \\ \hline H \mid \theta_1 \{p_i(\bar{c};\bar{d})\} \theta_2 \end{array}$$

where $\theta_1, \theta_2, \bar{c}, \bar{d}$, and H are arbitrary. Then the recursive version of the rule has the form:

$$\begin{array}{l} H \mid \underline{\text{Disjoint}}(\bar{a}^* \oplus \bar{z}_i{}^*) \\ I_1, \ldots, I_n \vdash H'_j \mid P_j\{B_j\} Q_j, \quad j=1,\ldots,n \\ H \mid \forall \bar{v}_i [P_i(\bar{a}/\bar{x}_i, \bar{b}/\bar{y}_i) \supset Q_i(\bar{x}_i'/\bar{x}_i, \bar{z}_i'/\bar{z}_i)] \supset \\ \quad [R\ [\![\bar{a}, \bar{z}_i \leftarrow \bar{x}_i', \bar{z}_i']\!] S] \\ \hline H \mid R\{p_i(\bar{a};\bar{b})\} S \end{array}$$

where $I_1, I_2, \ldots, I_n \vdash H'_j \mid P_j\{B_j\} Q_j$ means we may use the special rules I_i to prove $H'_j \mid P_j\{B_j\} Q_j$.

Unlike Hoare's original rule and the EUCLID rule, our recursive rule is relatively complete, even for the programs utilizing mutual recursion. Of the rules previously proposed in the literature, our rule most closely resembles that of [Gorelick 75]. Gorelick uses a more complex set of potentially mutually recursive procedures instead of $p_1, \ldots, p_n$ and divides the procedure call rule into two parts: a rule of modification and a rule of invariance. We originally formulated our procedure call rules in two part form, but abandoned the approach after we failed to devise a complete two-part rule. Gorelick achieves relative completeness by restricting actual <u>var</u> parameters to simple variables.

We can prove that the recursive version of the simple procedure call rule is sound by generalizing the argument we used for the non-recursive rule. First, we construct the sequences of procedures $p^0{}_i, p^1{}_i, \ldots, p^k, \ldots; i=1,\ldots,n$ as follows. We let $p^0{}_i$ be a non-terminating procedure with parameters identical to p_i. For k=1,2,..., we let p^k be defined by the procedure p^k(<u>var</u> $\overline{x_i:T_{x_i}}$; <u>value</u> $\overline{y_i:T_{y_i}}$); global $\bar{z}_i$; $B_i(p_j{}^{k-1}/p_j, j=1,\ldots,n)$, i.e. by the same declaration as p_i except each call $p_j(\bar{c};\bar{d})$ within the body of p_i is replaced by the call $p^{k-1}{}_j(\bar{c};\bar{d})$. Clearly, if the evaluation of an arbitrary call $p_i(\bar{a},\bar{b})$ requires less than k levels of nested calls on $p_1, p_2, \ldots, p_n$, then the call $p^k{}_i(\bar{a},\bar{b})$ is equivalent to $p_i(\bar{a},\bar{b})$. (Note that this statement does not hold if the restriction on procedure names is violated.) By the soundness of the non-recursive rule and simple induction on k, we know that the recursive rule is sound if we interpret p_j in the premises by $p^{k-1}{}_j$, j=1,...,n and p_i in the conclusion by $p^k{}_i$. Without loss of generality we may assume $p_i(\bar{a},\bar{b})$ terminates; otherwise, the rule is vacuously true. Let k be any integer greater than the maximum recursion calling depth on $p_1, \ldots, p_n$ in the evaluation of $p_i(\bar{a};\bar{b})$. By assumption, the premises are true for any interpretation of p_j, j=1,...,n consistent with H . Hence they must hold for p_j interpreted as $p^{k-1}{}_j$, implying the conclusion of the rule holds for $p^k{}_i(\bar{a},\bar{b})$. Since $p^k{}_i(\bar{a},\bar{b})$ is equivalent to $p_i(\bar{a},\bar{b})$, the conclusion of the rule must be true. Q.E.D.

The relative completeness of the recursive rule can be established by a similar inductive generalization of the proof for the non-recursive rule. We assume L is expressive. The proof proceeds by induction on the structure of a program. Let $p_1, \ldots, p_n$ be a sequence of procedures declared at the head of a block such that the rule is complete for the procedures declared within $p_1, \ldots, p_n$. Assume we are given recursion hypotheses for all of the procedures containing the procedures $p_1, \ldots, p_n$. We want to show that the weakest liberal pre-condition is provable for any procedure call in the block body given an arbitrary post-assertion. For each procedure p_i, we let P_i be $\bar{x}_i, \bar{y}_i, \bar{z}_i = \bar{x}^0{}_i, \bar{y}^0{}_i, \bar{z}^0{}_i$ and let Q'_i be the assertion defining the strongest post-condition of B_i given pre-condition P_i. As before, we let Q_i be $\exists\, \bar{y}'_i Q'_i(\bar{y}'_i/\bar{y}_i)$.

Let $q(\bar{c};\bar{d})$ be an arbitrary call in B_i such that q is either p_j for some j or a procedure containing the procedures $p_1, \ldots, p_n$. By the same argument we used in the non-recursive case, the weakest pre-condition of $q(\bar{c};\bar{d})$, given an arbitrary post-assertion S, is provable. Hence, since the remaining rules of the logic are complete by assumption, $P_i\{B_i\} Q_i$, i=1,...,n is provable. By applying the same argument again, we conclude that the weakest pre-condition of any call on a procedure q in the block body is provable.

By induction on the structure of a program, we can repeatedly apply the previous argument to derive that the procedure call rule is complete for calls in the body of the program. Q.E.D.

4. <u>Rules for Programs with Aliasing</u>

We now extend our version of Hoare's logic to handle aliasing. The modifications required are surprisingly minor.

Hoare's original assignment axiom has the form:

$$P(e/x)\{x \leftarrow e\} P$$

where x is a simple variable, e is an expression (term in the logical language L) and P is a formula. This axiom is invalid if x is a reference parameter or an array reference, since there may be syntactically distinct variables in P with access sequences identical to x. While Hoare's substitution style axiom can be patched to handle array assignment (by viewing the assignment $a[e_1] \leftarrow e_2$ as an abbreviation for the simple assignment $a \leftarrow \underline{\text{store}}(a, e_1, e_2)$), it completely breaks down in the case of aliasing.

In contrast, our assignment call rule does not rely on the concept of substitution (although it collapses to that form in trivial cases). As a result, our rule is able to handle array assignment and aliasing without any modification.

4.1 Reference Parameters

In a programming language with unrestricted reference parameters like PASCAL, we interpret procedure calls as passing the access sequences (i.e. abstract addresses) of the actual reference parameters to the procedure. In other words, the interpreter (Eval) binds a formal reference parameter to the access sequence of the corresponding actual parameter. For example, if p is a procedure with the single reference parameter x, then the procedure call $p(\alpha)$, where α is a variable specifier, binds x to the access sequence for α and evaluates the procedure body. In a language like PASCAL, every reference to a formal reference parameter is automatically dereferenced.

If x is a formal reference parameter bound to an actual parameter α, an assignment to x in the procedure body changes the binding of α (the variable to which x is bound); it does not change the binding of x. The binding of the formal reference parameter x is unchanged for the duration of the call.

Consequently, we consider PASCAL's notation for referring to formal reference parameters misleading. To remedy the situation in our PASCAL dialect, we require that every reference to a formal reference parameter x in the body of the procedure have the form $x\uparrow$ instead of x. (We have taken the $\uparrow$ operator from Pascal, where it serves a "dereferencing" operator for pointers.) For instance, if x is a reference parameter, then the standard Pascal statement $x \leftarrow x + 1$ is written as $x\uparrow \leftarrow x\uparrow + 1$ in our dialect. We also require formal reference parameter declarations to have the form x_i:ref T_i instead of x_i:T_i.

To accommodate aliasing within our logic, we must extend the set of Hoare assertions to include terms of the form $x\uparrow$ where x is declared in the declaration set H as x:ref T for some type T. We prohibit the dereferencing operator from appearing in other contexts. The meaning of $x\uparrow$, given state s consistent with H, is Value(s,Value(s,<'x'>)). The access sequence for $x\uparrow$ is the value of x. Consequently, the access sequence term for $x\uparrow$ is simply x.

Our proof rule for assignments to dereferenced formal reference parameters is identical to our ordinary assignment rule:

$$[\![x\uparrow \leftarrow e]\!]P\ \{x\uparrow \leftarrow e\}P$$

where we extend the definition of the simultaneous update $[\![\bar{v} \leftarrow \bar{t}]\!]\ \alpha$ as follows. Let α be a term or formula in L; $\bar{v}$ be a sequence of variable specifiers possibly including dereferenced formal reference parameters; and $\bar{t}$ be a corresponding sequence of terms (not containing updates). The meaning of $[\![\bar{v} \leftarrow \bar{t}]\!]\ \alpha$ for s is the meaning of α for Update*(s,$\bar{v}$*,$\bar{t}$) where Update* is extended to overlapping access sequences. Update* is defined by exactly the same axioms as before, except axiom 2) (Section 2.2) no longer requires the access sequences $<\alpha_1,\ldots,\alpha_n>$ to be disjoint. Informally, a simultaneous update $\bar{v} \leftarrow \bar{t}$ with overlapping variable-specifiers is performed in left-to-right order.

The soundness and relative completeness of the assignment rule stated above are an immediate consequence of the fact that

$$\text{Eval}(s,x\uparrow \leftarrow e) = \text{Update}(s,x,e).$$

In order to reason about updated formulas containing updates to dereferenced variables, we need the following axioms about updates. Let P and Q be arbitrary formulas, $u_1,\ldots,u_k$ be arbitrary terms, and $\bar{v} \leftarrow \bar{t}$ be an arbitrary simultaneous update. Then:

1. $[\![\bar{v} \leftarrow \bar{t}]\!](P\wedge Q) \equiv [\![\bar{v} \leftarrow \bar{t}]\!]P \wedge [\![\bar{v} \leftarrow \bar{t}]\!]Q$.
2. $[\![\bar{v} \leftarrow \bar{t}]\!](P\vee Q) \equiv [\![\bar{v} \leftarrow \bar{t}]\!]P \vee [\![\bar{v} \leftarrow \bar{t}]\!]Q$.
3. $[\![\bar{v} \leftarrow \bar{t}]\!](P\supset Q) \equiv [\![\bar{v} \leftarrow \bar{t}]\!]P \supset [\![\bar{v} \leftarrow \bar{T}]\!]Q$.
4. $[\![\bar{v} \leftarrow \bar{t}]\!]\neg P \equiv \neg[\![\bar{v} \leftarrow \bar{t}]\!]P$.
5. $[\![\bar{v} \leftarrow \bar{t}]\!]\forall\bar{x}P \equiv \forall\bar{x}\ [\![\bar{v} \leftarrow \bar{t}]\!]P$ where $\bar{x}$ not free in $\bar{t}$.
6. $[\![\bar{v} \leftarrow \bar{t}]\!]\exists\bar{x}P \equiv \exists\bar{x}\ [\![\bar{v} \leftarrow \bar{t}]\!]P$ where $\bar{x}$ not free in $\bar{t}$.
7. $[\![\bar{v} \leftarrow \bar{t}]\!]\ P_i(u_1,\ldots,u_k) \equiv$
   $P_i([\![\bar{v} \leftarrow \bar{t}]\!]u_1,\ldots,[\![\bar{v} \leftarrow \bar{t}]\!]u_k)$
   for every predicate symbol P_i (including equality).
8. $[\![\bar{v} \leftarrow \bar{t}]\!]\ f_i(u_1,\ldots,u_k) =$
   $f_i([\![\bar{v} \leftarrow \bar{t}]\!]u_1,\ldots,[\![\bar{v} \leftarrow \bar{t}]\!]u_k)$
   for every function symbol f_i.

These axioms enable us to move updates inside a formula to the point where they apply only to variable specifiers and logical variables. We also need axioms for updates to logical variables and variable specifiers. Let $v_1,\ldots,v_n$ be variable specifiers and $t_1,\ldots,t_n$ be corresponding terms. Let $[\![\ldots]\!][\![v_1,\ldots,v_n \leftarrow t_1,\ldots,t_n]\!]\alpha$ be an arbitrary updated variable specifier. Then:

1. $[\![\ldots]\!](v_n{}^*=\alpha^*) \supset [\![\ldots]\!]\ [\![\bar{v} \leftarrow \bar{t}]\!]\alpha = t_n$.
2. $[\![\ldots]\!](v_n{}^*\oplus\text{Seq}(d)=\alpha^*) \supset [\![\ldots]\!][\![\bar{v} \leftarrow \bar{t}]\!]\alpha = [\![\ldots]\!]\text{Select}(t_1,d)$.
3. $[\![\ldots]\!](v_n{}^*=\alpha^*\oplus\text{Seq}(d)) \supset [\![\ldots]\!][\![\bar{v} \leftarrow \bar{t}]\!]\alpha =$
   $[\![\ldots]\!][\![v_1,\ldots,v_{n-1} \leftarrow t_1,\ldots,t_n]\!]\text{Store}(\alpha,d,t_n)$.
4. $[\![\ldots]\!]\text{Disjoint}(\text{Seq}(v_n{}^*)\oplus\text{Seq}(\alpha^*)) \supset$
   $[\![\ldots]\!][\![\bar{v} \leftarrow \bar{t}]\!]\alpha =$
   $[\![\ldots]\!][\![v_1,\ldots,v_{n-1} \leftarrow t_1,\ldots,t_{n-1}]\!]\ \alpha$.

Since updates do not affect logical variables, the following axiom holds for arbitrary updated logical variable $[\![\ldots]\!]\ x'$:

5. $[\![\ldots]\!]x' = x'$.

The soundness of all the axioms for updates is an immediate consequence of the definition of truth for updated formulas.

We can use the axioms for updates to convert an arbitrary formula to update-free form. To accomplish this transformation, we repeatedly apply the following procedure. First, we push all updates inside the formula so that they apply only to variable specifiers and logical variables. We eliminate all updates to logical variables by applying axiom 5) above. Then for each updated variable specifier $[\![\ldots]\!]\ [\![\bar{v} \leftarrow \bar{t}]\!]\ \alpha$, we perform a case split on the relationship between $[\![\ldots]\!]v_n{}^*$ and α^* and apply the appropriate reduction (axioms 1), 2), 3), or 4) above) to each case, reducing the complexity of the updates involved.

While the update elimination procedure is of dubious practical value (since it can exponentially increase the size of a formula), it demonstrates that our axioms for updates are complete relative to the unextended base theory.

4.2 Generalized Simultaneous Assignment Rule

Given the generalized concept of update described in the previous section, we can generalize the simultaneous assignment axiom to permit overlapping variables on the left-hand side of the statement. The new simultaneous assignment axiom is identical to the old one except that the disjointness premise is omitted. Let $\bar{v} \leftarrow \bar{t}$ be a simultaneous assignment statement; P be a formula; and H be a declaration set declaring all the program variables appearing in P, $\bar{v}$, or $\bar{t}$. Then the generalized assignment rule states

$$H \mid [\![\bar{v} \leftarrow \bar{t}]\!]\, P \quad \{\bar{v} \leftarrow \bar{t}\}\quad P .$$

The soundness and completeness of the rule are an immediate consequence of the fact that <u>Eval</u>$(s, \bar{v} \leftarrow \bar{t}) = [\![\bar{v} \leftarrow \bar{t}]\!]s$ and definition of truth for statements in the logic.

4.3 Generalized Procedure Call Rule

Assume our PASCAL subset satisfies the restrictions listed in Section 3. Our generalized procedure call rule is nearly identical to the simple rule. Let p be declared as <u>procedure</u> p(<u>var</u> $\overline{x:\underline{ref}\ T_x}$; <u>value</u> $\overline{y:T_y}$); <u>global</u> $\bar{z}$; B in the declaration set H; let P and Q be formulas containing no free program variables other than $\bar{x}, \bar{x}\uparrow, \bar{y}, \bar{z}$ and $\bar{x}, \bar{x}\uparrow, \bar{z}$ respectively; let $\bar{v}$ be the free logical variables in P and Q; let $\bar{x}'$ and $\bar{y}'$ be fresh logical variables corresponding to $\bar{x}$ and $\bar{y}$; let R and S be formulas; and let H' denote H augmented by $\overline{x:\underline{ref}\ T_x}$, $\overline{y:T_y}$, and <u>Pair-disjoint</u>$(\bar{x}, \bar{x}^* \oplus \bar{y}^*)$ (where prior declarations of $\bar{x}$ and $\bar{y}$ are replaced). Then:

$$\frac{\begin{array}{l} H' \mid P\{B\}Q, \\ H \mid \bar{v}[P(\bar{a}^*/\bar{x}, \bar{a}/\bar{x}\uparrow, \bar{b}/\bar{y}) \supset Q(\bar{x}'/\bar{x}\uparrow, \bar{z}'/\bar{z})] \supset \\ \qquad [R \supset [\![\bar{a}, \bar{z} \leftarrow \bar{x}', \bar{z}']\!]S] \end{array}}{H \mid R\{p(\bar{a};\bar{b})\}S}$$

The disjointness hypothesis in H' asserts that the acess sequences for the formal parameters are disjoint from the passed actual reference parameter access sequences. From this hypothesis we can deduce that the dereferenced formal reference parameters do not any of the formal parameters as aliases. We must add an analogous hypothesis to the declarion rule given in Section 2.3.4.

4.3.1 Soundness and Relative Completeness

The soundness and relative completeness proofs for the generalized procedure call rule differ only in trivial details from the corresponding proofs for the simple rule. The only complication concerns the definition of <u>Eval</u>. We must not let <u>Eval</u> be confused by formal parameter names which match actual reference parameter names. The simplest solution is to force <u>Eval</u> to rename the actual parameters conflicting with formal parameter names before evaluating the procedure body. After evaluating the procedure body, <u>Eval</u> performs the appropriate simultaneous assignment.

4.3.2 A Sample Proof Involving Aliasing

Let swap be the standard integer swap procedure defined by

```
procedure swap(var x,y ref integer);
begin
  pre  x↑=xi ∧ y↑=yi;
  x↑,y↑ ←y↑,x↑;
  post y↑=xi ∧ x↑=yi
end; .
```

First we prove the correctness of the pre and post assertions. Let H be a declaration set including the declaration of swap. Let H' be H augmented by the formal parameter declarations of swap and the disjointness hypothesis. By the simultaneous assignment rule, proving the pre and post assertions for swap reduces to proving the verification condition:

$$H' \mid x\uparrow=x_i \wedge y\uparrow=y_i \supset [\![x\uparrow, y\uparrow \leftarrow y\uparrow, x\uparrow]\!](y\uparrow=x_i \wedge x\uparrow=y_i).$$

Moving the update inside generates the equivalent assertion:

$$H' \mid x\uparrow=x_i \wedge y\uparrow=y_i \supset [\![x\uparrow, y\uparrow \leftarrow y\uparrow, x\uparrow]\!]y\uparrow=x_i \wedge [\![x\uparrow, y\uparrow \leftarrow y\uparrow, x\uparrow]\!]x\uparrow=y_i$$

which immediately reduces to:

$$H' \mid x\uparrow=x_i \wedge y\uparrow=y_i \supset x\uparrow=x_i \wedge [\![x\uparrow, y\uparrow \leftarrow y\uparrow, x\uparrow]\!]x\uparrow=y_i$$

Since x and y are both <u>ref</u> <u>integers</u> we know that $H' \mid x=y \vee$ <u>Disjoint</u>(<u>Seq</u>(x)⊕<u>Seq</u>(y)).

In the former case (x=y), $[\![x\uparrow, y\uparrow \leftarrow y\uparrow, x\uparrow]\!]x\uparrow=x\uparrow$ reducing the verification condition to

$$H' \mid x\uparrow=x_i \wedge y\uparrow=y_i \supset x\uparrow=x_i \wedge x\uparrow=y_i$$

which is true since x=y. In the other case (x and y disjoint), $[\![x\uparrow, y\uparrow \leftarrow y\uparrow, x\uparrow]\!]x\uparrow=y\uparrow$, reducing the verification condition to

$$H' \mid x\uparrow=x_i \wedge y\uparrow=y_i \supset x\uparrow=x_i \wedge y\uparrow=y_i$$

which is an obvious tautology. Q.E.D.

Now let us examine a sample application of the generalized procedure call rule involving aliasing. Let H include the declarations a:<u>array</u> <u>integer</u> <u>of</u> <u>integer</u>, i:<u>integer</u>, j:<u>integer</u>. Assume we want to prove:

$$H \mid a[i]=a_1 \wedge a[j]=a_2\{swap(a[i],a[j])\}\, a[j]=a_1 \wedge a[i]=a_2 .$$

By the generalized procedure call rule, we must show

$$H \mid \forall x_0 y_0 [a[i]=x_0 \wedge a[j]=y_0 \supset y'=x_0 \wedge x'=y_0] \supset [a[i]=a_1 \wedge a[j]=a_2 \supset [\![a[i], a[j] \leftarrow x', y']\!](a[j]=a_1 \wedge a[i]=a_2)].$$

Let S' denote the consequent of the final implication. Moving the updates within S' further inside yields

$$[\![a[i], a[j] \leftarrow x', y']\!]a[j]=a_1 \wedge [\![a[i], a[j] \leftarrow x', y']\!]a[i]=a_2$$

which reduces to

$$y'=a_1 \wedge [\![a[i], a[j] \leftarrow x', y']\!]a[i]=a_2.$$

We instantiate the logical variables x_0, y_0 in the major hypothesis as a_1 and a_2 respectively, giving us the hypothesis

$$a[i]=a_1 \wedge a[j]=a_2 \supset y'=a_1 \wedge x'=a_2 .$$

Since the premise of this hypothesis is identical to the minor hypothesis, we deduce the new hypothesis

$$y'=a_1 \wedge x'=a_2 \;.$$

If $i \neq j$ then S' reduces precisely to this formula. On the other hand, if $i=j$ then S' reduces to

$$y'=a_1 \wedge y'=a_2$$

which is a simple consequence of the hypotheses $i=j$, $a[i]=a_1 \wedge a[j]=a_2$, and $y'=a_1 \wedge x'=a_2$. Q.E.D.

4.3.3 Handling Recursion

The recursive form of the generalized procedure call rule is completely analogous to the recursive generalization of the simple procedure call rule. The soundness and relative completeness proofs are also nearly identical to those for the simple rule.

5. Reducing the Complexity of Proofs Involving Aliasing

Although our rules for procedures with aliasing are relatively simple and easy to understand, they are rather cumbersome to use in practice, because they force all variable parameters to be passed by reference. Many procedures exploiting aliasing are designed to work only for a small subset of the possible aliasing configurations. If all variable parameters are passed by reference, the pre and post assertions for such a procedure must include a long list of disjointness assumptions.

We believe that a procedural programming language should provide two distinct classes of formal variable parameters: those which can have aliases and those which cannot. The explicit syntactic differentiation between these two classes greatly reduces the number of possible aliasing configurations, simplifying reasoning about updates.

To incorporate this modification into our PASCAL dialect, we establish the following new syntax for procedures:

$$\begin{array}{l} \underline{\text{procedure}}\ p(\underline{\text{var}}\ \overline{w{:}\underline{\text{ref}}\ T_w},\ \overline{x{:}T_x};\ \underline{\text{value}}\ \overline{y{:}T_y});\\ \quad \underline{\text{aliased}}\ \underline{\text{global}}\ \bar{z}_1;\\ \quad \underline{\text{global}}\ \bar{z}_2;\\ \quad B \end{array}$$

where $\bar{w}$ are reference parameters (as described in Section 4.1), $\bar{x}$ are variable parameters which have no aliases within the procedure, $\bar{y}$ are standard value parameters, $\bar{z}_1$ are global variables which may have aliases in the procedure and $\bar{z}_2$ are global variables which may not.

Within the procedure code block B, an assignment to any parameter v other than a reference parameter has the standard form:

$$v \leftarrow e \;.$$

In contrast, all references to a reference parameter must be explicitly dereferenced. Hence, an assignment to a reference parameter w has the form:

$$w\!\uparrow \leftarrow e \;.$$

The generalized procedure call rule (without recursion) for this extension of PASCAL has the following form. Let p be declared as shown above in a declaration set H; let P and Q be formulas in L containing no program variables other than $\bar{w},\bar{w}\!\uparrow,\bar{x},\bar{y},\bar{z}_1,\bar{z}_2$ and $\bar{w},\bar{w}\!\uparrow,\bar{x},\bar{z}_1,\bar{z}_2$ respectively; let $\bar{v}$ be the free logical variables in P and Q; let $\bar{w}',\bar{x}',\bar{z}_1',\bar{z}_2'$ be logical variables corresponding to $\bar{w}\!\uparrow,\bar{x},\bar{z}_1,\bar{z}_2$ respectively; let R and S be arbitrary formulas; and let H' be H augmented by $\overline{w{:}\underline{\text{ref}}\ T_w}$ $\overline{x{:}T_x}$, $\overline{y{:}T_y}$, and Pair-disjoint$(\bar{w},\bar{w}^*\oplus\bar{x}^*\oplus\bar{y}^*\oplus\bar{z}_2^*)$ (with prior declarations of $\bar{w},\bar{x},\bar{y}$ deleted). Then

$$\frac{\begin{array}{l} H' \mid P\{B\}Q,\\ H \mid \bar{v}[P(\bar{a}^*/\bar{w},\bar{a}/\bar{w}\!\uparrow,\bar{b}/\bar{x},\bar{c}/\bar{y})\\ \qquad Q(\bar{w}'/\bar{w}\!\uparrow,\bar{x}'/\bar{x},\bar{z}_1'/\bar{z}_1,\bar{z}_2',\bar{z}_2)]\\ \qquad [R\ [\![\bar{a},\bar{b},\bar{z}_1,\bar{z}_2 \leftarrow \bar{w}',\bar{x}',\bar{z}_1',\bar{z}_2']\!]S] \end{array}}{H \mid R\{p(\bar{a},\bar{b},\bar{c})\}Q}$$

The soundness and relative completeness proofs for the modified rule are essentially unchanged from before.

6. Eliminating the Remaining Restrictions

Our most general procedure call rules still require the following restrictions:

1. No parameters or functions may be passed as parameters.
2. Every global variable accessed in a procedure must be accessible at the point of every call.
3. No procedure named p may be declared within the scope of a procedure p.

As [Donahue 76] has pointed out, restriction 2) can be eliminated by making the declaration rule rename new variables within program text. A similar strategy can be used to eliminate restriction 3). In essence, this approach makes the rules rename program identifiers so that restrictions 2) and 3) hold after the renaming. We dislike the idea, however, because it modifies the text of a program (and any embedded assertions) in the course of a proof.

Fortunately, neither of these restrictions handicaps the programmer in any way. They simply force him to unambiguously name his variables and procedures. For this reason, we believe these two restrictions are a reasonable part of a practical programming language definition.

In constrast, the remaining restriction--the prohibition of procedures and functions as parameters--prevents the programmer from using an important language construct. In some application areas (such as numerical analysis), procedures and functions as parameters are nearly indispensable. We intend to extend Hoare's logic to handle this language construct in a subsequent paper.

7. Acknowledgments

We are grateful to Ralph London for discovering several errors in an earlier draft of this paper.

References

[Apt and DeBakker 77]
Apt, K. R. and J. W. DeBakker. Semantics and Proof Theory of PASCAL Procedures, Tech. Rpt., Stichting Mathematisch Centrum, Amsterdam, 1977.

[Cook 75]
Cook, S. Axiomatic and Interpretive Semantics for an Algol Fragment, Tech. Rpt. 79, Dept. of Comp. Sci., Univ. of Toronto, Feb., 1975.

[Donahue 76]
Donahue, J. E. Complementary Definitions of Programming Language Semantics, Springer-Verlag, Berlin, 1976.

[Enderton 72]
Enderton, H. A Mathematical Introduction to Logic, Academic Press, New York, 1972.

[Gorelick 75]
Gorelick, G. A. A Complete Axiomatic System for Proving Assertions about Recursive and Non-recursive Programs, Tech. Rpt. 75, Dept. of Comp. Sci., Univ. of Toronto, Jan., 1975.

[Guttag 77]
Guttag, J. V., J. J. Horning, and R. L. London. A Proof Rule for EUCLID Procedures, Tech. Rpt., Xerox PARC, Palo Alto, Ca., 1977.

[Igarashi, London, and Luckham 75]
Automatic Program Verification I: A Logical Basis and Its Implementation, Acta Informatica 4 (1975), pp. 145-182.

[Hoare 69]
Hoare, C. A. R. An Axiomatic Approach to Computer Programming, CACM 12, 10 (Oct., 1969), pp. 322-329.

[Hoare 71]
Hoare, C. A. R. Procedures and Parameters: An Axiomatic Approach, Symp. on Semantics of Algorithmic Languages, E. Engler (ed.), Springer-Verlag, Berlin, 1971, pp. 102-116.

[Hoare and Wirth 73]
Hoare, C. A. R. and N. Wirth. An Axiomatic Definition of the Programming Language PASCAL, Acta Informatica 1 (1971).

[London 77]
London, R. L., et al. Proof Rules for the Programming Language EUCLID, Tech. Rpt., Xerox PARC, Palo Alto, Ca., 1977.

[Oppen 75]
Oppen, D. C. On Logic and Program Verification, Tech. Rpt. 82, Dept. of Comp. Sci., Univ. of Toronto, 1975.