A PROCEDURE CALL PROOF RULE (WITH A SIMPLE EXPLANATION)*

by

David Gries
and
Gary Levin

TR 79-379

Computer Science Department
Cornell University
Ithaca, New York 14853

May 1979

*This work was partially supported by the National Science Foundation under grant MCS 76-22360.

1. Introduction

We present a procedure call proof rule for a language with

(a) one-dimensional arrays and records, in which array elements and record fields may themselves be arrays or records; pointers are not allowed;

(b) procedures with global variables, var (call by reference) parameters, and ALGOL 60 value parameters;

(c) procedure calls in which no aliasing is allowed among the global variables and var arguments, but no restriction is placed on value arguments.

(d) no recursion, although it could be introduced with little difficulty.

An attempt is made to keep the notation as simple as possible and to make the proof rule as clear and understandable as possible. We will, at the end of this report, compare briefly this proof rule with others found in the literature.

Also appearing in this report is a definition of multiple assignment. This is given because the concept is necessary for understanding procedure calls, and also because the definition is much simpler, shorter and more general than the one previously published by the first author [4].

Procedure calls with aliasing will be discussed, and an idea will be outlined for a definition of such calls that reflects the notion that such aliasing, while perhaps useful at times, should be treated as a separate, special case.

2. References and aliasing

Consider a "variable" like b[i].s, where b is an array of records with field s. Such a *reference*, as we will call it, consists of an identifier -- b -- concatenated with a (possibly null) *selector* -- [i].s -- which is essentially a sequence of subscripts and field names. The symbol ϵ denotes the null selector. The symbol $\circ$ will be used from time to time to denote concatenation of identifiers and selectors -- usually when identifiers are used to represent arbitrary selectors, e.g. $b\circ s1\circ s2$. Note also that $b = b\circ\epsilon$.

Two references are <u>aliased</u> if one is an initial segment of the other, which means that the latter references part of the value of the former. Thus b.r and b are aliased because b begins b.r. The two references b.r and b.s are not aliased.

When dealing with subscripts in examining selectors for aliasing, the <u>values</u> of the subscripts, and not their syntactic representations, are to be used. For example, b[i] and b[j] are aliased only if $i=j$, while b[i+1].r and b[i].r are never aliased. Clearly, aliasing is not a syntactic property.

Note carefully that b[i] and i are not aliased.

We write

(2.1) $\text{disjoint}(\bar{x})$, where $\bar{x} = x_1, \ldots, x_n$,

to mean that no aliasing occurs among the x_i. Finally,

(2.2) $\text{pdisjoint}(\bar{x};\bar{y})$, where $\bar{x},\bar{y}$ are sequences of references,

means that no x_i is aliased to a y_j.

3. <u>Multiple assignment and the substitution rule</u>

The Pascal report [6] defines assignment as $\{P^x_e\}\ x{:=}e\{P\}$, where P^x_e is defined as

(a) P^x_e denotes conventional substitution if x is an identifier;

(b) $P^{b[i]}_e = P^b_{(b;\ i:e)}$ for an array b;

(c) $P^{v.r}_e = P^v_{(v;\ r:e)}$ for a record v with field r.

This is a recursive definition because the rules must be applied again if b (say) is itself of the form a[j] (say).

To define multiple assignment we first extend the definition of (b; i:e) slightly to allow <u>any</u> selector to appear where i is and not just a subscript or field name. Thus (b; s:e) is defined by three simple rules:

(3.1) Definition

(a) $(b;\ \epsilon{:}e) = e$

(b) $(b;\ [i]\circ s{:}e)[j] = \begin{cases} j = i \rightarrow (b[j];\ s{:}e) \\ j \neq i \rightarrow b[j] \end{cases}$ for array b

(c) $(b;\ r\circ s{:}e).t = \begin{cases} t \equiv r \rightarrow (b.r;\ s{:}e) \\ t \not\equiv r \rightarrow b.t \end{cases}$ for record b and field names r, t. $\equiv$ denotes syntactic equality.

When there is no chance of misunderstanding, we omit brackets and periods, writing, for example, $(b;\ [i]{:}e)$ as $(b;\ i{:}e)$. We also omit extra parentheses in expressions such as $(((b;\ s1{:}e1);\ s2{:}e2);\ s3{:}e3)$; this one is written as $(b;\ s1{:}e1;\ s2{:}e2;\ s3{:}e3)$.

Subsequently, the following property will be used:

(3.2) $(b;\ s{:}b\circ s) = b$

Let $\bar{e}$ be a list of expressions and $\bar{x}$ a corresponding list of references, where each x_i has the form $id_i\circ s_i$ for an identifier id_i and selector s_i. Definition (3.3), given below, defines textual substitution $P^{\bar{x}}_{\bar{e}}$ in a predicate P so that the proof rule for an assignment $\bar{x}{:=}\bar{e}$ is

$$\{P^{\bar{x}}_{\bar{e}}\}\ \bar{x}{:=}\ \bar{e}\ \{P\}$$

and the model of execution of the assignment statement is

(1) determine the components $\bar{v}$ referenced by the $\bar{x}$;
(2) evaluate the expressions $\bar{e}$ to yield values $\bar{w}$;
(3) assign values $\bar{w}$ to the $\bar{v}$, in left to right order.

(For simplicity, we have omitted the requirement that the x_i be valid references and the e_i well defined at the point of execution.)

(3.3) Definition. $P^{\bar{x}}_{\bar{e}}$ is given by the following three rules.

(a) Provided $\bar{x}$ is a list of different identifiers, $P^{\bar{x}}_{\bar{e}}$ is the result of conventional simultaneous substitution in

P of the e_i for the x_i, with suitable prior replacement of bound identifiers to avoid conflict.

(b) $$P^{b\circ s_1,\ \ldots,\ b\circ s_m,\ \bar{x}}_{u_1,\ \ldots,\ u_m,\ \bar{e}} = P^{b,\bar{x}}_{(b;\ s_1:u_1;\ \ldots;\ s_m:u_m),\ \bar{e}}$$

provided that identifier b does not begin any of the x_i. This rule indicates how multiple assignments to an object b can be viewed as a single assignment to b.

(c) $$P^{\bar{x},\ b\circ s, c\circ t,\ \bar{y}}_{\bar{e},\ u_1,\ u_2,\ \bar{f}} = P^{\bar{x},\ c\circ t, b\circ s,\ \bar{y}}_{\bar{e},\ u_2,\ u_1,\ \bar{f}}$$

provided that b and c are different identifiers. This rule indicates that adjacent reference-selector pairs may be permuted as along as they begin with different identifiers and are thus disjoint.

It is reassuring to see that this definition of textual substitution enjoys the following property.

(3.4) Lemma. Let $\bar{u}$ be a sequence of fresh, different identifiers. Then $P = (P^{\bar{x}}_{\bar{u}})^{\bar{u}}_{\bar{x}}$ for any list of references $\bar{x}$.

Proof. The lemma is obvious when $\bar{x}$ consists only of different identifiers, and we need only consider the case $\bar{x} = b\circ s_1, \ldots, b\circ s_n$. In this case,

$$\begin{aligned}(P^{\bar{x}}_{\bar{u}})^{\bar{u}}_{\bar{x}} &= \left(P^{b}_{(b;\ s_1:u_1;\ \ldots;\ s_n:u_n)}\right)^{\bar{u}}_{\bar{x}} \\ &= P^{b}_{(b;\ s_1:b\circ s_1;\ \ldots;\ s_n:b\circ s_n)} \\ &= P^{b}_{b} \qquad \text{(by n applications of (3.2))} \\ &= P\end{aligned}$$

We give no examples of the use of the new assignment statement rule; turn to [4] if you need them. Note that the rule allows assignments such as

```
a[i],a[j].r,a[j].s:= e1,e2,e3
```

even if $i = j$. While such assignments should rarely, if ever, occur, the concept of state change through multiple assignment is important enough in programming that it is reassuring to have a simple proof rule that covers all cases.

4. Procedure calls without aliasing

The procedure definition

(4.1) <u>proc</u> p(<u>var</u> $\bar{x}$; <u>value</u> $\bar{y}$); <u>global</u> $\bar{z}$; $\{P\}$ B $\{Q\}$

defines a procedure p with <u>var</u> (i.e. call by reference) parameters $\bar{x} = x_1, \ldots, x_n$, <u>value</u> parameters $\bar{y}$, global variables $\bar{z}$ and body B. The z_i may be arbitrary references. Furthermore, it has been proved that $\{P\}$ B $\{Q\}$ is true for all states. The program variables in predicates P and Q may only be $\bar{x}$, $\bar{y}$, $\bar{z}$ and $\bar{x}$, $\bar{z}$, respectively. Note that Q may not contain the <u>value</u> parameters, since nothing can be said about them once execution of the procedure call has terminated.

For the rest of this report, we assume that procedure p has been declared as in (4.1), and that $\{P\}$ B $\{Q\}$ has been proven, with the restriction mentioned on their program variables. We are ignoring problems of scope in order to concentrate on more essential matters.

Now consider a call $p(\bar{b},\bar{c})$, where the references $\bar{b}$ correspond to the <u>var</u> parameters and the expressions $\bar{c}$ to the <u>value</u> parameters. We want to arrive at a proof rule that allows us to prove

(4.2) $\{R'\}$ $p(\bar{b},\bar{c})$ $\{R\}$

where R' is some -- still to be determined -- predicate that depends on R, the arguments and the procedure definition.

First, since the proof $\{P\}$ B $\{Q\}$ of the procedure body is written under the assumption that the <u>var</u> parameters and globals are disjoint (not aliased), it is reasonable to require the same of the <u>var</u> arguments and globals. Hence, R' will contain a conjunct disjoint$(\bar{b},\bar{z})$.

Secondly, in order to ensure that, after parameter-argument correspondence, it makes sense to execute the procedure body, R' will contain a conjunct $P^{\bar{x},\bar{y}}_{\bar{b},\bar{c}}$, where P is the precondition of the procedure body.

Finally, R' must contain a conjunct that ensures that R will be true <u>after</u> execution of the call. The key to the development of this conjunct is the following:

> any assignment of values $\bar{u},\bar{v}$ to the <u>var</u> parameters and globals $\bar{x},\bar{z}$ must also be an assignment to the <u>var</u> arguments and globals $\bar{b},\bar{z}$.

Suppose particular values $\bar{u},\bar{v}$ are assigned to $\bar{x},\bar{z}$ by execution of the body B. Then the body may be viewed as equivalent, in this case, to the assignment $\bar{x},\bar{z}:= \bar{u},\bar{v}$, in which case

$$Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}} \quad = \quad wp(\text{"}\bar{x},\bar{z}:= \bar{u},\bar{v}\text{"},\ Q)$$

holds and $Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}}$ must be true before the call. But then the call itself can be viewed as an assignment $\bar{b},\bar{z}:= \bar{u},\bar{v}$, and we have

$$R^{\bar{b},\bar{z}}_{\bar{u},\bar{v}} \quad = \quad wp(\text{"}\bar{b},\bar{z}:= \bar{u},\bar{v}\text{"},\ R)$$

Hence, we can ensure that R will be true <u>after</u> execution of the call by requiring that

$$(\underline{A}\bar{u},\bar{v}:\ Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}}:\ R^{\bar{b},\bar{z}}_{\bar{u},\bar{v}}) \qquad \text{(or } (\underline{A}\bar{u},\bar{v}:\ Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}} \Rightarrow R^{\bar{b},\bar{z}}_{\bar{u},\bar{v}}))$$

We therefore have developed the third conjunct of R' and propose the proof rule

(4.3)
$$\frac{\text{p declared as in (4.1) and its subsequent text}}{\{\text{disjoint}(\bar{b},\bar{z})\ \underline{\text{and}}\ P^{\bar{x},\bar{y}}_{\bar{b},\bar{c}}\ \underline{\text{and}}\ (\underline{A}\bar{u},\bar{v}:\ Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}}:\ R^{\bar{b},\bar{z}}_{\bar{u},\bar{v}})\}\ \ p(b,c)\ \{R\}}$$

<u>Remark 1</u> We are of the opinion that, if

$$P = wp(B,Q),$$

then

$$wp(p(\bar{b},\bar{c}),R) = \text{disjoint}(\bar{b},\bar{z})\ \underline{\text{and}}\ P^{\bar{x},\bar{y}}_{\bar{b},\bar{c}}\ \underline{\text{and}}\ (\underline{A}\bar{u},\bar{v}:\ Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}}:\ R^{\bar{b},\bar{z}}_{\bar{u},\bar{v}})$$

We hesitate to claim this without further investigation. <u>end of remark</u>

<u>Remark 2</u> One could write (4.3) as

(4.4) $\begin{array}{l} p \text{ declared as in (4.1) and its subsequent text;} \\ S \Rightarrow (\text{disjoint}(\bar{b},\bar{z}) \ \underline{\text{and}}\ P^{\bar{x},\bar{y}}_{\bar{b},\bar{c}}); \\ (\underline{A}\bar{u},\bar{v}\text{: } S \ \underline{\text{and}}\ Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}}\text{: } R^{\bar{b},\bar{z}}_{\bar{u},\bar{v}}), \quad \text{where } S \text{ does not contain } \bar{u},\bar{v} \\ \hline \{S\}\ \ p(\bar{b},\bar{c})\ \ \{R\} \end{array}$

With the help of the rule of consequence, (4.4) can be proved from (4.3) and, if (4.4) is taken as the inference rule, (4.3) can be proved. _end of remark_

Remark 3 Remembering that a statement $\{P\}$ S $\{R\}$ is really an abbreviation for $P \Rightarrow wp(S,R)$, we see that the conclusion of (4.3) can be rewritten to have the form

(4.5) $(\underline{A}\bar{g}\text{: } N(\bar{g})) \Rightarrow R1$

where predicate RR1 does not contain the free logical variables $\bar{g}$ and N does. Cartwright and Oppen [2] noticed that the Euclid procedure call definition [5] mistakenly wrote this as

(4.6) $N(g) \Rightarrow R1.$

The reason this is a mistake had to be explained by Cartwright and the second author to the first author. We can write (4.5) as

$$\begin{aligned} (4.5) &= \underline{\text{not}}\ (\underline{A}\bar{g}\text{: } N(\bar{g}))\ \underline{\text{or}}\ R1 \\ &= (\underline{E}\bar{g}\text{: } \underline{\text{not}}\ N(\bar{g}))\ \underline{\text{or}}\ R1 \end{aligned}$$

Predicate (4.6), on the other hand, has the free logical variables $\bar{g}$, which by convention are universally quantified over the whole predicate:

$$\begin{aligned} (4.6) &= (\underline{A}\bar{g}\text{: } N(\bar{g}) \Rightarrow R1) \\ &= (\underline{A}\bar{g}\text{: } \underline{\text{not}}\ N(\bar{g})\ \underline{\text{or}}\ R1) \\ &= (\underline{A}\bar{g}\text{: } \underline{\text{not}}\ N(\bar{g}))\ \underline{\text{or}}\ R1 \end{aligned}$$

since R1 does not contain any of the g_i as free variables. Hence (4.5) and (4.6) are different. The moral of the story is that we computer scientists should be more proficient in the use of our basic mathematical tools, such as the predicate calculus. _end of remark_

Example 1 Consider the procedure

$$\underline{\text{proc}}\ \text{swap}(\underline{\text{var}}\ x1,x2);\quad \begin{array}{l}\{P:\ x1 = c1\ \underline{\text{and}}\ x2 = c2\}\\ x1,x2 := x2,x1\\ \{Q:\ x1 = c2\ \underline{\text{and}}\ x2 = c1\}\end{array}$$

Predicates P and Q contain logical variables c1,c2 to describe initial values of x1,x2, and what is really meant is

(4.7) $(\underline{A}c1,c2:\ \{P\}\ x1,x2:= x2,x1\ \{Q\})$.

We want to prove the following about a call swap(i,b[i]), where we have left out the subscript range for array b:

$$(4.8)\quad \begin{array}{l}\{i = I\ \underline{\text{and}}\ (\underline{A}j:\ b[j] = B[j])\}\\ \text{swap}(i,b[i])\\ \{R:\ i = B[I]\ \underline{\text{and}}\ b[I] = I\ \underline{\text{and}}\ (\underline{A}j:\ I \neq j:\ b[j] = B[j])\}\end{array}$$

Logical variables I and B denote initial values of i and array b, respectively. Because (4.7) holds, we can identify c1 with I and c2 with B[I] and write $P \equiv (x1 = I\ \underline{\text{and}}\ x2 = B[I])$, $Q \equiv (x1 = B[I]$ and $x2 = I)$. Using proof rule (4.3) we have {R'} swap(i, b[i]) {R} where R is given in (4.8) and R' is

$$\begin{array}{l}\text{disjoint}(i,b[i])\ \underline{\text{and}}\ (i = I\ \underline{\text{and}}\ b[i] = B[I])\ \underline{\text{and}}\\ (\underline{A}u1,u2:\ u1 = B[I]\ \underline{\text{and}}\ u2 = I:\ u1 = B[I]\ \underline{\text{and}}\ (b;\ i{:}u2)[I] = I\\ \qquad\qquad\qquad \underline{\text{and}}\ (\underline{A}j:\ I \neq j:\ (b;\ i{:}u2)[j] = B[j]))\end{array}$$

The first conjunct of R' is always true, and it is easily seen that the other two are implied by the precondition of (4.5).

This example shows the use of the proof rule whe arguments are not identifiers -- which is not allowed by Donahue [3] or Alagic and Arbib [1]. end of examole 1

Example 2 Consider the procedure

$\underline{\text{proc}}$ p($\underline{\text{var}}$ x1,x2; $\underline{\text{value}}$ y);
$\{P: y = c\}$ x1,x2:= y,y $\{Q: x1 = c \ \underline{\text{and}}\ x2 = c\}$

which assigns the value parameter to both var parameters. Note that Q does not contain the value parameter y. We want to prove the following about a call p(a[i+1],i,a[i]):

(4.9) $\{a[i] = A \ \underline{\text{and}}\ i = I\}$ p(a[i+1],i,a[i]) $\{R: a[I] = a[I+1] = i = A\}$

We identify logical variable c with A and write $P \equiv (y = A)$ and $Q \equiv (x1 = x2 = A)$. Using proof rule (4.3) we find precondition R':

$$\begin{aligned}
&\text{disjoint}(a[i+1],i)\ \underline{\text{and}}\ a[i] = A\ \underline{\text{and}}\\
&(\underline{A}u1,u2\colon u1 = u2 = A\colon (a;\ i+1:u1)[I] = (a;\ i+1:u1)[I+1] = u2 = A)\\
= \ &\text{disjoint}(a[i+1],i)\ \underline{\text{and}}\ a[i] = A\ \underline{\text{and}}\\
&((a;\ i+1:A)[I] = (a;\ i+1:A)[I+1] = A)
\end{aligned}$$

The first conjunct is always true, and the second two conjuncts are implied by the precondition of (4.9), and hence (4.9) holds.

Most proof rules (Euclid's, and Cartwright and Oppen's are exceptions) require that no variables in a var argument occur in a value argument. We don't need to require this because the postcondition Q of the procedure body is not allowed to contain var parameters. end of example

The reader should not be deceived by the fact that the bodies of the procedures of these examples are so short and simple, for the application of the procedure call proof rule has nothing to do with the body of the procedure -- it relies only on the pre- and postconditions of the procedure body. The two examples were chosen to illustrate the use of the proof rule in situations that are banned in other, simpler proof rules.

5. <u>A simpler proof rule</u>

Consider proof rule (4.3). It would be nice not to have the complicated conjunct $(\underline{A}\bar{u},\bar{v}\colon Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}}\colon R^{\bar{b},\bar{z}}_{\bar{u},\bar{v}})$ in the conclusion, so let us try to eliminate it. Suppose we write $R^{\bar{b},\bar{z}}_{\bar{u},\bar{v}}$ as

(5.1) $R^{\bar{b},\bar{z}}_{\bar{u},\bar{v}} \equiv Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}}$ <u>and</u> I

where the program variables of I are disjoint from $\bar{b},\bar{z}$. Then

$$(\underline{A}\bar{u},\bar{v}\colon Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}}\colon R^{\bar{b},\bar{z}}_{\bar{u},\bar{v}}) = (\underline{A}\bar{u},\bar{v}\colon Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}}\colon Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}}\ \underline{\text{and}}\ I)$$
$$= I$$

The problem, then, is to find R so that (5.1) holds. We can assume that $\bar{u},\bar{v}$ are syntactically distinct, in which case, using lemma (3.4), we have

$$R = (R^{\bar{b},\bar{z}}_{\bar{u},\bar{v}})^{\bar{u},\bar{v}}_{\bar{b},\bar{z}}$$
$$= (Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}}\ \underline{\text{and}}\ I)^{\bar{u},\bar{v}}_{\bar{b},\bar{z}}$$
$$= Q^{\bar{x}}_{\bar{b}}\ \underline{\text{and}}\ I$$

Hence we can use $Q^{\bar{x}}_{\bar{b}}$ <u>and</u> I for R <u>if</u> (5.1) holds for this R. That is, we must have

$$R^{\bar{b},\bar{z}}_{\bar{u},\bar{v}} = (Q^{\bar{x}}_{\bar{b}}\ \underline{\text{and}}\ I)^{\bar{b},\bar{z}}_{\bar{u},\bar{v}} = Q^{\bar{x},\bar{z}}_{\bar{u},\bar{v}}\ \underline{\text{and}}\ I$$

But this holds <u>only</u> if $\bar{b}$ satisfies some rather stringent properties -- stronger than disjointness. One such property is that $\bar{b}$ consist only of different identifiers. This leads us to the rule

(5.2) p declared as in (4.1) and its subsequent text
disjoint($\bar{b},\bar{z}$) <u>and</u> pdisjoint($\bar{b},\bar{z}$; ref(I))

$$\frac{}{\{P^{\bar{x},\bar{y}}_{\bar{b},\bar{c}}\ \underline{\text{and}}\ I\}\ p(\bar{b},\bar{c})\ \{Q^{\bar{x}}_{\bar{b}}\ \underline{\text{and}}\ I\}}$$

provided $\bar{b}$ consists of different identifiers

The term ref(I) denotes the set of references in I. This rule includes a "rule of invariance" -- predicates I that don't refer to globals of the procedure or <u>var</u> arguments remain invariantly true.

6. Procedure calls with aliasing

Often, a procedure call with aliasing among the var parameters and globals leads to an error. There are times, however, when such aliasing is useful. For example, consider a procedure add(b1,b2,b3), which accepts two arrays b1 and b2 and stores their element-wise sum in array b3. A call add(a,b,a) seems reasonable. However, most current languages and machines require, for economy of storage, that all three parameters be var parameters. Whether this call works or not depends entirely on how the procedure body is written.

In order to convey our idea for handling calls with aliasing, which is all we want to do, let us make some simplifications. Assume procedures have no global variables, and that only array subscripts (no record field names) appear in var parameters. We will give one proof rule, which handles the simplest kind of aliasing, and outline another.

Rule (6.1) is for a call $p(\bar{b},\bar{c})$ of a procedure with body B, in which two of the var arguments, b_i and b_j for $i \neq j$, are the same identifier. The first premise indicates that procedure p must have been defined. The second premise indicates that from p one can derive a second procedure p' whose body is the same as B except that all references to parameter x_i are replaced by x_j, and $\{P\}\ B^{x_i}_{x_j}\ \{Q\}$ has been proved. This illustrates our basic idea: in order to use a certain kind of aliasing in a call, one must have proved that execution of the procedure body with this particular aliasing does indeed perform as expected. And, one should handle the aliasing as a special case.

(6.1) <u>proc</u> $p($<u>var</u> $\bar{x}$; <u>value</u> $\bar{y})$; B defined or derived

<u>proc</u> $p'($<u>var</u> $\bar{x}/i$; <u>value</u> $\bar{y})$; $\{P\}\ B^{x_i}_{x_j}\ \{Q\}$ derived from p

$\{R\}\ p'(\bar{b}/i,\bar{c})\ \{S\}$

$\{R\}\ p(\bar{b},\bar{c})\ \{S\}$

where $\bar{x}/i$ is the list $\bar{x}$ with the i^{th} element removed,
b_i and b_j are the same for some i,j, $i \neq j$,
P and Q may not contain x_i.

Consider a procedure and call

<u>proc</u> p(<u>var</u> x); B and p(b[i]).

Because <u>var</u> parameters are call by reference, this is obviously equivalent to the tranformed procedure and call

<u>proc</u> p'(<u>var</u> x; <u>value</u> I); $B^{x}_{x[I]}$ and p'(b,i).

In the transformation the subscript i becomes a <u>value</u> argument and the corresponding parameter is appended as a subscript to each occurrence of x in the procedure body B.

Now, if we have a call in a situation where aliasing might occur, it may be prudent to (mentally) apply the above transformation as often as necessary, apply rule (6.1), and thus show that the transformed procedure call is correct. As an example, the procdure

<u>proc</u> swap(<u>var</u> x1,x2); x1,x2:= x2,x1

and call p(b[i],b[j]) where i may be equal to j would be tranformed into

<u>proc</u> swap'(<u>var</u> x1; <u>value</u> I,J); x1[I],x1[J] := x1[J],x1[I]

and

swap'(b,i,j).

This notion could be formalized in a language definition to yield a definition in which the simpler rule (4.3) could be used except in cases where aliasing could occur, in which case the more complicated rule would be called for.

7. Brief notes on other procedure call rules

A number of proof rules for procedure calls have been proposed by others. Here we would like to comment briefly on some of them. Hoare's definition [7] showed very nicely how, with some simplifications, one could arrive at a really simple proof rule in which the method of argument-parameter correspondence would be left to the implementor. While it is a nice idea, and the paper should be read by all, it has not caught on because people are not willing to restrict procedure calls enough in their languages. The Pascal report [6] contains a proof rule for handling most of Pascal's procedure calls, including global variables. It requires the programmer to develop functions that describe how each of the globals and var parameters are defined in terms of their initial values. It seems difficult to use and, moreover, cannot be used in a nondeterminstic language.

The text by Arbib and Alagic [1] contains a rule that is too simplistic to be of much use in any real situation. Donahue [3] has developed a rule that is similar to (5.2), with its rule of invariance. In fact, the idea for attempting to derive the simpler rule came from feading his thesis. Rule (5.2) is more general than Donahue's in that it makes no restrictions on the value parameters and allows global variables.

If one reads the Euclid [5] procedure call proof rule carefully enough, one finds it is quite close to (4.3). One finds three differences. First, they made the mistake discussed in section 4. More importantly, the authors elected to include in the rule a definition of initial values of arguments. This complicated their rule tremendously; we prefer to keep the proof rule simple and leave that complication to the language in which assertions are written. Finally, their definition of multiple assignment or textual substitution, which is important for understanding procedure calls, is not as clean.

Actually, the rule that inspired us to develop our new rule was found in a paper by Cartwright and Oppen [2]. Their rule, which says essentially the same thing, was unfortunately buried in a mass of notation.

Acknowledgements This research was started by the first author. After a while Gary Levin had made so many contributions that it seemed only fair to include him as an author. It is a pleasure to also gratefully acknowledge the help of Corky Cartwright and Jim Donahue.

References

[1] Alagić, S. and M. A. Arbib. The Design of Well-Structured and Correct Programs. Springer Verlag, New York, 1978.

[2] Cartwright, R. and D. Oppen. Unrestricted procedure calls in Hoare's logic. Proc. ACM Symp. on Principles of Programming Languages, Jan. 1978, 131-140.

[3] Donahue, J. E. Complementary Definitions of Programming Language Semantics. LNCS 42, Springer Verlag, New York, 1976.

[4] Gries, D. The multiple assignment statement. IEEE Trans. Software Eng. SE-4 (March 1978), 89-93.

[5] Guttag, J. V., J. J. Horning and R. L. London. A proof rule for Euclid procedures. ISI/RR-77-60, May 1977. (Arpa Order No. 2223).

[6] Hoare, C. A. R. and N. Wirth. An axiomatic definition of the programming language Pascal. Acta Informatica 2 (1973), 335-355.

[7] Hoare, C. A. R. Procedures and parameters: an axiomatic approach. Symp. on the Semantics of Algorithmic Languages (E. Engeler, ed.), Springer Verlag Notes in Mathematics 188, 1971, 102-116.