Acta Informatica 10, 1–26 (1978)

Acta Informatica

Proof Rules for the Programming Language Euclid

R.L. London[1]*, J.V. Guttag[2]**, J.J. Horning[3]***, B.W. Lampson[4], J.G. Mitchell[4], and G.J. Popek[5]****

[1] USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291, USA
[2] Computer Science Dept., University of Southern California, Los Angeles, CA 90007, USA
[3] Computer Systems Research Group, University of Toronto, Toronto, M5S 1A4, Canada
[4] Xerox Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA
[5] 3532 Boelter Hall, Computer Science Dept., University of California, Los Angeles, CA 90024, USA

Summary. In the spirit of the previous axiomatization of the programming language Pascal, this paper describes Hoare-style proof rules for Euclid, a programming language intended for the expression of system programs which are to be verified. All constructs of Euclid are covered except for storage allocation and machine dependencies.

"*The symbolic form of the work has been forced upon us by necessity: without its help we should have been unable to perform the requisite reasoning.*"
A.N. Whitehead and B. Russell, *Principia Mathematica*, p. vii

"*Rules are rules.*" Anonymous

Introduction

The programming language Euclid has been designed to facilitate the construction of verifiable system programs. Its defining report [11] closely follows the defining report [15] of the Pascal language (see also [10]). The present document, giving Hoare-style proof rules applicable only to legal Euclid programs, owes a great deal to (and is in part identical to) the axiomatic definition of Pascal [9]. Major differences from [9] include the treatment of procedures and functions, declarations, modules, collections, escape statements, binding, parameterized types, and the examples and detailed explanations in Appendices 1–3. Other semantic defini-

* To whom offprint requests should be sent. Supported by the Defense Advanced Research Projects Agency under contract DAHC-15-72-C-0308
** Supported in part by the National Science Foundation under grant MCS-76-86089 and the Joint Services Electronics Program monitored by the Air Force Office of Scientific Research under contract F 44620-76-C-0061
*** Supported in part by a Research Leave Grant from the University of Toronto and a grant from the National Research Council of Canada. *Current address:* Xerox Research Center
**** Supported in part by the Defense Advanced Research Projects Agency under contract DAHC-73-C-0368. The views expressed are those of the authors

tion methods are certainly applicable to Euclid. We have used proof rules for two reasons: familiarity and the existence of the Pascal definition. Readers unfamiliar with proof rules may to read [6, 7].

One may regard the proof rules as a definition of Euclid in the same sense as the Pascal axiomatization defines Pascal. By stating what can be proved about Euclid programs, the rules define the meaning of most syntactically and semantically legal Euclid programs, but they do not give the information required to determine whether or not a program is legal. This information may be found in the language report. Neither do the proof rules define the meaning of illegal Euclid programs containing, for example, division by zero or an invalid array index. Finally, explicit proof rules are not provided for those portions of Euclid defined in the report by translation into other legal Euclid constructs. This includes **pervasive**, implicit imports through **thus**, and some uses of **return** and **exit**. All such transformations must be applied before the proof rules are applicable.

As is the case with Pascal, the Euclid axiomatization should be read in conjunction with the language report, and is an almost total axiomatization of a realistic and useful system programming language. While the primary goal of the Euclid effort was to design a practical programming language (not to provide a vehicle for demonstrating proof rules), proof rule considerations did have significant influence on Euclid [13]. All constructs of the language are covered except for storage allocation (zones and collections that are not referenced-counted) and machine dependencies. In a few instances rules are applicable only to a subset of Euclid; the restrictions are noted with those rules.

Conventions and Notation

In describing these Euclid proof rules we have used as much as possible of the Pascal axiomatization. We have also deliberately followed the same order and style of presentation and tried to use the same terminology.

The notation $P(y/x)$ to denote substitution is used for the formula which is obtained by systematically substituting y for all free occurrences of x in the predicate P. If this introduces conflict between free variables of y and bound variables of P, the conflict is resolved by systematic change of the latter variables. The notation $P(y/x)$ may be read "*P with y for x.*"

$$P(y_1/x_1, \ldots, y_n/x_n)$$

denotes simultaneous substitution for all occurrences of any x_i by the corresponding y_i. Thus occurrences of x_i within any y_j are *not* replaced. The expressions $x_1, \ldots, x_n$ must be distinct; otherwise the simultaneous substitution is not defined. The meaning of substitution for subscripted and qualified expressions is defined in Sections 4 and 5.

Euclid expressions in assertions must be legal Euclid expressions. Assertions may contain, outside of expressions, non-Euclid notations such as quantifiers or [illegible]

In the proof rules, S and S_j denote a sequence of zero or more statements. As is done in the Pascal axiomatization, we refer to values of a type as elements of that type. Also, the rule of consequence

$$\frac{P \supset Q,\ Q\{S\}\ U,\ U \supset R}{P\{S\}\ R}$$

may be used in proofs.

Data Types

The axioms presented in this and the following sections display the relationship between a type declaration and the axioms which specify the properties of values of the type and operations defined over them. The treatment is not wholly formal, and the reader must be aware that:

1. Free variables in axioms are assumed to be universally quantified over the appropriate type.
2. The expression of the "induction" axiom is always left informal.
3. The types of variables used in the axioms have to be deduced either from the section heading or from the more immediate context.
4. The name of a type is used as a transfer function constructing a value of the type (such a use of the type identifier is not available in Euclid).
5. The verifier can assume that everything will be type-checked by the compiler or that the compiler will generate the necessary legality assertions.
6. In defining Euclid's types we will not be presenting proof rules that may be directly applied to Euclid programs. Rather, we will provide a set of assertions, denoted by H, about the values of the type being defined. These assertions are incorporated into proof rules in Section 9 on constant, variable, and type declarations. Rule 9.1, for example, tells us that in the case of an identifier declaration, $x:T$, we may use the fact that $x \in T$, i.e., x has the attributes associated with values of type T.

Parameterized types are covered in Section 9.5. The rules for type compatibility in Euclid are defined in the language report.

Scalar Types

type $T=(c_1, c_2, \ldots, c_n)$

1.1. $c_1, c_2, \ldots, c_n$ are distinct elements of T.
1.2. These are the only elements of T.
1.3. $c_{i+1} = T.succ(c_i)$ for $i=1, \ldots, n-1$
1.4. $c_i = T.pred(c_{i+1})$ for $i=1, \ldots, n-1$
1.5. $\neg(x<x)$
1.6. $(x<y) \wedge (y<z) \supset (x<z)$
1.7. $(x \neq c_n) \supset (x<T.succ(x))$
1.8. $x>y \equiv y<x$

1.9. $x \leqq y \equiv \neg(x > y)$
1.10. $x \geqq y \equiv \neg(x < y)$
1.11. $x \neq y \equiv \neg(x = y)$
1.12. $T.\,first = c_1$
1.13. $T.\,last = c_n$
1.14. $T.\,ord(c_i) = i - 1$ for $i = 1, \ldots, n$.

The standard scalar type *Boolean* is defined as

type *Boolean* = (False, True).

The Boolean operators **not**, **and**, **or**, and → (implication) are those of the conventional logical calculus except that some operands may possibly not be evaluated. Specifically, in terms of conditional expressions,

x **and** y means **if** x **then** y **else** false, i.e., x cand y,
x **or** y means **if** x **then** true **else** y, i.e., x cor y,
$x \rightarrow y$ means **if** x **then** y **else** true.

(Note, however, that conditional expressions are not included in Euclid.)

The standard type *integer* stands for the set of the whole numbers. The arithmetic operators +, −, *, and **div** are those of whole number arithmetic. The modulus operator **mod** is defined by the equation

m **mod** $n = m - (m$ **div** $n) * n$

whereas **div** denotes division with truncated fraction, i.e., move toward zero. Implementations are permitted to refuse the execution of programs which refer to integers outside the appropriate range, *signedInt* or *unsignedInt*.

Type Char

2.1. The elements of type *char* are the 26 (capital) letters, the 10 (decimal) digits, and possibly other characters defined by particular implementations. In programs, a constant of type *char* is denoted by preceding the character with a $.

The sets of letters and digits are ordered, and the digits are coherent, i.e.,

2.2. $\$A < \$B$ $\$a < \$b$ $\$1 = char.\,succ(\$0)$
$\$B < \$C$ $\$b < \$c$ $\$2 = char.\,succ(\$1)$
... ...
$\$Y < \$Z$ $\$y < \$z$ $\$9 = char.\,succ(\$8)$.

Axioms 1.5–1.13 apply to the *char* type. The functions $char \cdot ord$ and *Chr* are defined by the following additional axioms:

2.3. If u is an element of *char*, then $char.\,ord(u)$ is a non-negative integer (called the *ordinal number* of u), and

$Chr(char.\,ord(u)) = u$

2.4. $u < v \equiv char.\,ord(u) < char.\,ord(v)$.

These axioms have been designed to make possible interchange of programs between implementations using different character sets. It should be noted that the function $char.\,ord$ does not necessarily map the characters onto consecutive integers.

Subrange Types

type $T = m\,..\,n$

Let m, n be elements of scalar type T_0.

3.1. $T.\,first = m$
3.2. $T.\,last = n$
3.3. For all i in T_0 such that $m \leqq i \leqq n$, i is an element of T.
3.4. These are the only elements of T.
3.5. $T.\,first \leqq i < T.\,last \supset T.\,succ(i) = T_0.\,succ(i)$
3.6. $T.\,first < i \leqq T.\,last \supset T.\,pred(i) = T_0.\,pred(i)$.

Note that since all elements of T are elements of T_0, all operators of T_0 apply to them.

Array Types

type $T =$ **array** I **of** T_0

Here I is a scalar or subrange type. Let $m = I.\,first$ and $n = I.\,last$.

4.1. If x_i is an element of T_0 for all i such that $m \leqq i \leqq n$, then $T(x_m, \ldots, x_n)$ is an element of T.
4.2. These are the only elements of T.
4.3. $T(x_m, \ldots, x_n)(i) = x_i$ for $m \leqq i \leqq n$.

Letting x stand for $T(x_m, \ldots, x_n)$ we introduce the following abbreviation:

$(x, i: y)$ stands for $T(x_m, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_n)$.

The formula

$P(y/x(i))$,

denoting a substitution for an array reference, is then defined to be

$P((x, i: y)/x)$.

To extend this definition to substitution for a multiply-subscripted array reference, we define

$(x, \langle i_1, \ldots, i_n \rangle: y)$, where $n \geqq 2$

to be equal to

$(x, \langle i_1, \ldots, i_{n-1}\rangle : (x(i_1) \ldots (i_{n-1}), i_n : y))$

and the formula

$P(y/x(i_1) \ldots (i_n))$, where $n \geqq 2$

is equivalent to

$P((x, \langle i_1, \ldots, i_n\rangle : y)/x)$.

4.4. x . *IndexType* $= I$, x . *ComponentType* $= T_0$.

Record Types

type $T =$ **record** $s_1 : T_1, \ldots, s_m : T_m$ **end** T

where each s_i must be preceded by **const** or **var**.

Let x_i be an element of T_i for $i = 1, \ldots, m$.

5.1. $T(x_1, x_2, \ldots, x_m)$ is an element of T.
5.2. These are the only elements of T.
5.3. $T(x_1, \ldots, x_m) . s_i = x_i$ for $i = 1, \ldots, m$.

$(x, s_i : y)$ stands for $T(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_m)$.

The formula $P(y/x . s_i)$ is defined analogously to arrays (see definitions following 4.3).

Variant Records

type $T_0(c) =$ **record** $s_1 : T_1, \ldots, s_m : T_m$;

case c **of**
$k_1 \Rightarrow s_1' : T_1'$ **end** k_1
$k_2 \Rightarrow s_2' : T_2'$ **end** k_2
...
$k_n \Rightarrow s_n' : T_n'$ **end** k_n
end case
end T

where the type of c is a scalar or subrange type with elements $k_1, \ldots, k_n$. Note that $k_a, k_b, \ldots, k_m \Rightarrow S$ stands for $k_a \Rightarrow S$; $k_b \Rightarrow S$; ...; $k_m \Rightarrow S$.

Consider type $T = T_0(k_j)$, and let x_j' be an element of T_j' for $j = 1, \ldots, n$.

5.1a. $T(x_1, \ldots, x_m, k_j, x_j')$ is an element of T.
5.2a. These are the only elements of T.
5.3a. $T(x_1, \ldots, x_m, k_j, x_j') . s_i = x_i$ for $i = 1, \ldots, m$

5.4a. $T(x_1, \ldots, x_m, k_j, x_j') . s_j' = x_j'$ for $j = 1, \ldots, n$
5.5a. Letting z stand for $T(x_1, \ldots, x_m, k_j, x_j')$, the standard Euclid component *itsTag* is defined by z . *itsTag* $= k_j$.

A variant record may also contain an **otherwise** clause

otherwise $\Rightarrow s_{n+1}' : T_{n+1}'$

in which case the type of c may contain elements in addition to $k_1, \ldots, k_n$. If there is such a clause, we have the following additional axioms for the **otherwise** $(n+1)$ situation:

Let k_j be an element of the type of c such that $k_j \neq k_i$ for $1 \leqq i \leqq n$, and let x_{n+1}' be an element of T_{n+1}'.

5.1b. $T(x_1, \ldots, x_m, k_j, x_{n+1}')$ is also an element of T.
5.4b. $T(x_1, \ldots, x_m, k_j, x_{n+1}') . s_{n+1}' = x_{n+1}'$
5.5b. Identical to 5.5a.

The case with a field list containing several fields

$k_j \Rightarrow s_{j1} : T_{j1}, \ldots, s_{jh} : T_{jh}$ **end** k_j

is to be interpreted as

$k_j \Rightarrow s_j' : T_j'$ **end** k_j

where s_j' is a fresh identifier, and T_j' is a type defined as

type $T_j' =$ **record** $s_{j1} : T_{j1}, \ldots, s_{jh} : T_{jh}$ **end record**.

In this case $x . s_{jt}$ is interpreted as $x . s_j' . s_{jt}$.

Now consider type $T = T_0$(**any**).

5.1c. $T(x_1, \ldots, x_m, y, x')$ is an element of T for $(y, x') = (k_1, x_1'), \ldots, (k_n, x_n'), (k_d, x_{n+1}')$ where $k_d \neq k_i$ for $i = 1, \ldots, n$.
5.2c. These are the only elements of T.
5.3c. $T(x_1, \ldots, x_m, y, x') . s_i = x_i$ for $i = 1, \ldots, m$.

There is no 5.4c since access to the variant part of the record may only be done using the discriminating case statement (see Section 12.5).

Machine-dependent records do not change verification properties. The additional information supplied can be ignored (unless the machine interprets certain locations specially).

Set Types

type $T =$ **set of** T_0

Let x_0 be an element of T_0.

6.1. $T(\ )$ is an element of T.
6.2. If x is an element of T, then $x + T(x_0)$ is an element of T (the operator $+$ is defined below).

6.3. These are the only elements of T.
6.4. $T(x_1, x_2, \ldots, x_n)$ means $T([[T(\,) + T(x_1)] + T(x_2)] + \cdots + T(x_n))$.
6.5. $T.BaseType = T_0$.

$T(\,)$ denotes the empty set, and $T(x_0)$ denotes the singleton set containing x_0. The operators $+$, $*$, $-$, **xor**, and **in**, applied to elements whose type is set, denote the conventional operations of set union, intersection, difference, symmetric difference, and membership. The operators $<=$ and $>=$ denote set inclusion. The specific axioms defining these operators are omitted.

Note that Euclid allows implementations to restrict set types to be built only on base types T_0 with a specified maximum number of elements.

Module Types

Because modules are by far the most complex part of Euclid, this section contains a large amount of explanation as well as the technical details of the module rule. The explanatory material can be read now. However, we recommend that Section 10 on procedures and functions be read before reading the rule and the technical details. An example of the use of the module rule appears as Appendix 3. Additional comments on the module construct appear in the Epilogue. The material on modules appears here because modules, like subrange types and array types, are type constructors.

type *T*(**const** *C*) = **pre** *P1*; **post** *Q1*; **module** *t* **invariant** *Q0*;
  imports (**var** *Y* **const** *D* **readonly** *R*)
  exports (**const** (*K1*, *p*, *f*) **var** *V1* **readonly** *R1*)
  const *K*; **var** *V*
  procedure *p*(**var** *X2* nonvar *C2*) = **imports** (**var** *Y2* nonvar *D2*)
    pre *P2*; **post** *Q2*; **begin** S_2 **end**
  function *f*(nonvar *C3*) **returns** *G* = **imports** (nonvar *D3*)
    pre *P3*; **post** *Q3*; **begin** S_3 **end**
  initially imports (**var** *Y4* nonvar *D4*) **post** *Q4*; **begin** S_4 **end**
  abstraction function *A* **returns** t_0 = **imports** (nonvar *D5*)
    begin S_5 **end**
  invariant *Q*
  finally imports (**var** *Y6* nonvar *D6*) **pre** *P6*; **begin** S_6 **end**
end *T*

The parts of a module definition are explained in the section on module types in the Euclid report. We assume that equality is not exported. Nonvar denotes the list of **const** and **readonly** identifiers. Each of *c*, *y*, *d*, *r*, *k1*, *v1*, *r1*, *k*, *v*, *x2*, *c2*, *y2*, *d2*, *c3*, *d3*, *y4*, *d4*, *d5*, *y6*, and *d6* denotes the list of identifiers in the corresponding upper case declaration (*p* and *f* denote declared routines). As the Euclid report requires, the lists *k1*, *v1*, and *r1* are sublists of *k*, *v*, and *v* (yes, *v*, not *r*), respectively. The declarations *K* and *V* may not include identifiers from *y* or *r*. The list *d5* is a sublist of *c*, *d*, *k*, and $v - v1$. The arguments of *P1* and *Q1* may include as free variables only identifiers from *c*, *y*, *d*, and *r*. Similarly, the arguments of *P2*, *Q2*, *P3*, *Q3*, *Q4*, and *P6* may only include *t* and their respective formal parameter and import lists; *Q3* may also include *g*. *Q*, the "concrete invariant," may only include *c*, *d*, *k*, and $v - v1$, that is to say, the same identifiers as *d5*. *Q0*, the "abstract invariant," may include *c*, *d*, and *t*.

The exported identifiers of *k1*, *v1*, and *r1* are treated as if they were fields of a record type named *T*. The abstraction function *A* maps the identifiers in *d5* into a value of type $T(c)$; its body may contain constructs outside Euclid. The function *A* is for verification purposes and is not callable from a Euclid program.

The module is a mechanism for providing encapsulation and the support of data abstractions. There are two points from which it may be viewed: The users of the module see only the abstract pre- and postconditions associated with module routines and the pre- and postcondition of the module itself. The implementor of the module also sees the bodies of the routines and the (concrete) identifiers declared within the module. The connection between the concrete and abstract identifiers is the abstraction function, *A*, which transforms a set of concrete identifiers to an abstract identifier. See [8] for a more complete discussion of the role of *A*.

The module rule given below contains a conclusion and eight premises. We shall explain the structure of the rule, describe the purpose and workings of each premise, and finally give the rule itself.

The conclusion of the rule involves the instantiation of a module identifier and the use of that identifier in a scope. Premises 1–6 are properties required of the module definition; these verifications need be done only once per module definition. Premise 7 discharges the instantiation precondition; this must be proved each time a module is instantiated. Premise 8, itself in five parts, uses the verified definitions (formulas 8.1–8.4 which depend on premises 1–6) to verify the uses, in premise 8.5, of the module identifier in the scope. Thus the module rule has the structure

$$\frac{1, 2, 3, 4, 5, 6,\quad 7,\quad [8.1, 8.2, 8.3, 8.4] \vdash 8.5}{P\{\textbf{var } x;\ T(a);\ S\}\ R \wedge Q1}$$

We now describe each premise in more detail. In premises 1–6 the substitution of a call of the abstraction function, *A*, for the name of the module, *t*, converts a predicate on the abstract identifier *t* to one involving concrete identifiers. Premise 1: concrete invariant implies abstract invariant. Premise 2: module precondition across declaration of module's local variables and body of **initially** establishes the postcondition of **initially** and the concrete invariant. Premise 3: verification of each exported procedure body, i.e., precondition and concrete invariant across body establishes postcondition and preserves concrete invariant. Premises 4–5: these two premises are for each exported function body. Premise 4 is analogous to premise 3 except (a) the concrete invariant is automatically preserved since functions have no side effects, and (b) the single-valued requirement described following Rule 10.2 is included. Premise 5 also concerns functions

and is the consistency clause described after Rule 10.2. Premise 6: **finally** establishes the postcondition of the module.

Premise 7: instantiation environment implies module precondition with actuals substituted for formals.

Premise 8: shows how the instantiated module variable x is used in the scope S. It must be shown in premise 8.5 that the instantiation environment across **initially**, S, and **finally** establishes R. In showing premise 8.5 one may use the four formulas 8.1–8.4, which give the properties of the procedures, functions, **initially**, and **finally**, respectively. Formulas 8.1 and 8.2 correspond to the conclusions of the procedure and function call rules; the only difference is that the abstract invariant may be used in proving the preconditions and is assumed following the calls. (This is the source of much of the utility of the module construct. It allows us to prove theorems using data type (generator) induction.) Formula 8.3 treats x. *Initially* as a parameterless procedure call that establishes the abstract invariant. Formula 8.4 treats x. *Finally* as a parameterless procedure call for which the abstract invariant may be used in establishing its precondition. (If x is declared to be an array of modules or a record containing modules, then x. *Initially* and x. *Finally* must each be replaced in 8.3, 8.4, and 8.5 by a sequence of calls to initialization and finalization routines, respectively. These sequences are defined in the report.)

Here, then, is the full module rule.

7.1. (module rule)

(1) $Q \supset Q0(A/t)$,

(2) $P1\{\textbf{const } K; \textbf{var } V; S_4\}\ Q4(A/t) \wedge Q$,

(3) $P2(A/t) \wedge Q\{S_2\}\ Q2(A/t) \wedge Q$,

(4) $\exists g1(P3(A/t) \wedge Q\{S_3\}\ Q3(A/t) \wedge g = g1(A, c, d))$,

(5) $\exists g(P3(A/t) \wedge Q \supset Q3(A/t))$,

(6) $P6(A/t) \wedge Q\{S_6\}\ Q1$,

(7) $P \supset P1(a/c)$,

(8.1) $[Q0(a/c, x/t, x'/t') \supset (P2(x/t, x'/t', a2/x2, e2/c2, a/c) \wedge (Q2(x2\#/t, x'/t', a2\#/x2, e2/c2, a/c, y2\#/y2, a2/x2', y2/y2') \supset R1(x2\#/x, a2\#/a2, y2\#/y2)))\{x.\ p(a2, e2)\}\ R1 \wedge Q0(a/c, x/t, x'/t')$,

(8.2) $(Q0(a/c, x/t) \supset P3(x/t, a3/c3, a/c)) \supset Q3(x/t, a3/c3, a/c, f(a3, d3)/g) \wedge Q0(a/c, x/t)$,

(8.3) $P1(a/c) \wedge (Q4(x4\#/t, x'/t', a/c, y4\#/y4, y4/y4') \supset R4(x4\#/x, y4\#/y4))\{x.\ Initially\}\ R4 \wedge Q0(a/c, x/t, x'/t')$,

(8.4) $(Q0(a/c, x/t, x'/t') \supset P6(x/t, x'/t', a/c)) \wedge (Q1(a/c, y6\#/y6, y6/y6') \supset R(y6\#/y6))\{x.\ Finally\}\ R]$

$\vdash$

(8.5) $P(x\#/x)\{x.\ Initially; S; x.\ Finally\}\ R(x\#/x)$

$P\{\textbf{var } x: T(a); S\}\ R \wedge Q1$

where the "#" denotes fresh identifiers and the scope of x is exactly S. As noted above, calls in S to module routines use formulas 8.1–8.4. Although not written, calls in S_2, S_3, S_4, and S_6 to module routines are similarly handled, but using the A/t substitution ("x." is missing) and without assuming $Q0(a/c, x/t)$ or $Q0(a/c, x/t, x'/t')$. Since a module may not import its own name, premise 2 is never "recursively applied" in T. In the interest of simplicity, the formulas for recursion, **return**, and H are omitted in premises 3–5. Non-exported routines follow the rules for routines in Section 10.

In the simple case where the module T imports no **var** variables, i.e., **initially** and **finally** can have no side effects outside T, premise 6 and $Q1$ are missing (**finally** can have no visible effect) and premise 8 can be just

$$[8.1, 8.2] \vdash P(x\#/x) \wedge Q4(x/t) \wedge Q0(x/t)\ \{S\}\ R(x\#/x)$$

where 8.1 and 8.2 contain no #'s on identifiers from y.

This formulation of the module rule follows [8]. Other approaches [14, 5], which use different specification methods, might have been substituted and, of course, may be used to verify programs containing modules. With these alternative approaches there would be changes only to the verification information of the module.

Pointer and Collection Types

type $T_0 =$ **collection of** T'
var C: T_0
type $T = \uparrow C$

8.1. C. *nil* is an element of T.

8.2. There are an unbounded number of elements of T: $\pi_1, \pi_2, \ldots$ (see 8.7).

8.3. If $\beta_1, \ldots, \beta_n$ are elements of T' and $\pi_1, \ldots, \pi_n$ are distinct members, $\neq C$. *nil*, of T, then

$T_0(\{\pi_1 : \beta_1, \ldots, \pi_n : \beta_n\})$ is an element of T_0.

Note that $\pi_i : \beta_i$ indicates that π_i denotes the component β_i in the collection.

8.4. These are the only elements of T_0.

8.5. If $C = T_0(\{\pi_1 : \beta_1, \ldots, \pi_n : \beta_n\})$ then $C(\pi_i) = \beta_i$.

8.6. For any element $\pi \neq C$. *nil* of type T, $\pi\uparrow = C(\pi)$.

We introduce the following abbreviation:

If $C = T_0(\{\pi_1 : \beta_1, \ldots, \pi_n : \beta_n\})$ then $(C, \pi_i : y)$ stands for

$$T_0(\{\pi_1 : \beta_1, \ldots, \pi_n : \beta_n\} - \{\pi_i : \beta_i\} + \{\pi_i : y\}).$$

No operations are defined on the elements of T except test of equality, $\uparrow$, and the standard function C. *Index*. The only defined property of C. *Index* is that it is one-to-one. For every collection there are two standard procedures, C. *New*(t) and C. *Free*(t), involving the elements of T. Assume the declaration **var** t: T, and that $C = T_0(\{\pi_1 : \beta_1, \ldots, \pi_n : \beta_n\})$.

8.7. C. *New*(t) means $t := \pi$
where π is an element of T, $\pi \neq \pi_i$ for $i = 1, \ldots, n$,

and $C := T_0(\{\pi_1:\beta_1, \ldots, \pi_n:\beta_n\} + \{\pi:\beta\})$ where β is undefined (and may not be referenced).

8.8. C. *Free*(t) means
$C := T_0(\{\pi_1:\beta_1, \ldots, \pi_n:\beta_n\} - \{t:C(t)\})$ (recall $C(t)$ means $t\uparrow$)
$t := C$. *nil*.

Declarations

The purpose of a declaration is to introduce a named object (constant, type, variable, function, or procedure) and to prescribe its properties. These properties may then be assumed in any proof relating to the scope of the declaration.

Constant, Variable, and Type Declarations

If D is a sequence of variable and type declarations, then

D; S

is called a *scope*, and the following are its rules of inference (some expressed in the usual notation for subsidiary deductions):

9.1. $\dfrac{x\#\,.\,itsType = T, x\#\in T \vdash P\{S(x\#/x)\}\ Q}{P\{\textbf{var}\ x\colon T;\ S\}\ Q}$

where *itsType* is the standard Euclid component. The substitution of the fresh identifier $x\#$ for x expresses the fact that x is a "local" variable. T is any type except a module (see Section 7) or a structured type (record or array) involving modules whose **initially** or **finally** clauses modify imported **var** identifiers. The separate rules required to cover these structured types are omitted.

9.2. $P(e/x)\ \{\textbf{const}\ x := e\}\ P$.

This axiom also applies to structured constants according to the order of the components.

9.3a. $\dfrac{x = y \wedge P(x\#/x)\ \{S\}\ Q(x\#/x, x/y)}{P\{\textbf{bind}\ \text{varBindingCondition}\ x\ \textbf{to}\ y;\ S\}\ Q}$

9.3b. $\dfrac{i = i_0 \wedge x = y(i) \wedge P(x\#/x)\ \{S\}\ Q(x\#/x, (y, i_0\colon x)/y)}{P\{\textbf{bind}\ \text{varBindingCondition}\ x\ \textbf{to}\ y(i);\ S\}\ Q}$

where, in all cases, i_0 and $x\#$ are fresh variables; and where varBindingCondition is **readonly**, **var** or empty.

9.4. $\dfrac{H \vdash P\{S\}\ Q}{P\{\textbf{type}\ T = \cdots;\ S\}\ Q}$

where H is the set of assertions derived from the type declaration of T in the manner described in Sections 1–6 and 8.

Parameterized Types

type $T(c) =$ **pre** $P1$; *Defn*

9.5. $\dfrac{P \supset P1(a/c),\ P\{\textbf{const}\ c := a; \textbf{type}\ T' = Defn; \textbf{var}\ x\colon T';\ S\}\ Q}{P\{\textbf{var}\ x\colon T(a);\ S\}\ Q}$

type $T'(\ldots, x, \ldots) = \cdots$
type $T_0 =$ **collection of** $T'(\ldots, \textbf{unknown}, \ldots)$
var $C\colon T_0$
type $T = \uparrow C$
var $t\colon T$

If T is referenced as the object type of a collection, then one or more of the actual parameters may be specified as **unknown**. In such cases the component of H (in Rule 9.4) associated with t is the disjunction

$\cup_{a\ \text{in}\ x + (\text{any})}[t\in\uparrow \textbf{collection of}\ T'(\ldots, a, \ldots)]$.

9.6. C. *New*(t, y) means all of Section 8.7 except that the type of β is $T'(\ldots, y, \ldots)$.

Procedure Declarations and Calls

procedure p(**var** X, nonvar C) = **imports** (**var** Y, nonvar D)
pre P; **post** Q; **begin** S **end**

Nonvar denotes the list of **const** and **readonly** identifiers. Let x, c, y, and d be the identifiers declared in X, C, Y, and D, respectively. P and Q are each predicates involving as free variables only x, c, y, and d. Q may also involve x' and y', fresh variables which denote the initial values of x and y on entry to the procedure body.

10.1. (procedure-call rule)

$$\frac{\begin{array}{l}[P(a1/x, e1/c) \wedge (Q(a1\#/x, e1/c, y\#/y, a1/x', y/y') \supset \\ \qquad R1(a1\#/a1, y\#/y))\ \{p(a1, e1)\}\ R1,\ Q\ \{\textbf{return asserting}\ Q\}\ \text{false}, H] \\ \vdash \\ x = x' \wedge y = y' \wedge P\{S\}\ Q\end{array}}{P(a/x, e/c) \wedge (Q(a\#/x, e/c, y\#/y, a/x', y/y') \supset R(a\#/a, y\#/y))\ \{p(a, e)\}\ R}$$

This rule, which is similar to the adaptation rules in [7, 3], assumes the above declaration of procedure p. If the procedure p is nonrecursive, the premise of 10.1 can be just

$[Q\ \{\textbf{return asserting}\ Q\}\ \text{false}, H] \vdash x = x' \wedge y = y' \wedge P\{S\}\ Q$.

In 10.1 a and e are the list of actual parameters which correspond respectively to the formal parameters specified as variable and nonvariable parameters. Note that the elements of a and y will, in any legal Euclid program, all be distinct in the sense that none can overlap any other. H is the conjunction of the assertions for each $x \in X$ and $c \in C$, that they are elements of their declared type. (The members of Y and D need not be included in this H, but are covered by 9.1 and 9.4.) The "#" indicates that fresh variables are to be used. Recall that the prime symbol (') denotes initial value of the corresponding formal parameter at procedure body entry. By convention, P and R do not contain primes. An example of the application of Rule 10.1 is contained in Appendix 1. A full discussion of this rule appears in [4] which compares this rule to several other rules, including those contained in [7, 9].

It should be noted that if two or more of the actuals, a, are components of the same array, a slight complication arises in substituting for the actuals. $R(a\#/a, y\#/y)$ may evaluate to a formula of the form

$$R(a1\#/B(i_1) \ldots (i_m), a2\#/B(j_1) \ldots (j_n)).$$

Since the non-overlapping rule guarantees that there exists a k where $1 \leqq k \leqq m \leqq n$ such that $i_k \neq j_k$, the substitution is well-defined. Applying the rule for array element substitution will reduce this to

$$R((B, \langle i_1, \ldots, i_m \rangle : a1\#)/B, (B, \langle j_1, \ldots, j_n \rangle : a2\#)/B).$$

At this point simultaneous substitution is no longer well-defined. We therefore define *extended simultaneous substitution* with the rule

$$R((B, \langle i_1, \ldots, i_m \rangle : a1\#)/B, (B, \langle j_1, \ldots, j_n \rangle : a2\#)/B)$$
$$= R(((B, \langle i_1, \ldots, i_m \rangle : a1\#), \langle j_1, \ldots, j_n \rangle : a2\#)/B).$$

Note that in verifying Euclid programs this situation can only arise in connection with substitutions generated by application of the procedure-call rule. In this environment, we know, because of the non-overlapping restriction, that replacing the simultaneous substitution with sequential substitutions produces identical results regardless of the order in which they are performed.

Note that we do not have an independent rule covering the **return** statement. Rather, we have embedded it in the above rule for procedure calls, which allows us to use the axiom

$$Q\,\{\textbf{return asserting } Q\}\ \text{false}$$

in proving $P\{S\}Q$ for p. Informally the rule states that any **return** causes us to exit the statically enclosing procedure. Although the syntax of Euclid is just "**return**," we have added the "**asserting** Q" clause in order to state succinctly the axiom for **return**. We assume a preprocessor, if necessary, that determines the statically enclosing procedure associated with each **return** and adds to each **return** the corresponding Q. This addition is necessary to ensure against making an unsound inference about a **return** from an internally nested procedure with a different postcondition. The statement **return when** B may be replaced (as

specified in the Euclid report) by the statement

if B **then return end if**.

Beware, the axiom involving **return** may not be used immediately if the procedure p contains an instantiation of a module whose **finally** clause falsifies Q. In such cases, the expansion described in the Euclid report for moving the **finally** clause must be first applied.

Rule 10.1 may be used in proving assertions about calls of the procedure p, including those occurring within S itself or in other declarations in the same scope. The rule is applicable to all recursive calls because of the clause in the premise to the left of the turnstile, $\vdash$. In this "recursion" clause note that the symbols are deliberately different from those in the rule's conclusion: $R1$ replaces R, and $a1$ and $e1$ replace a and e to allow different formulas and actual parameters to be used for recursive calls. The entire premise of Rule 10.1 need be proved only once for each procedure declaration, not once for each call.

For a procedure declaration itself we have

10.1a. $R\,\{\textbf{procedure } p \ldots \textbf{begin } S \textbf{ end}\}\ R$.

Function Declarations and Calls

function f(nonvar C) **returns** G = **imports** (nonvar D)
pre P; **post** Q; **begin** S **end**

The same notation is used as in procedures. Nonvar denotes the list of **const** and **readonly** identifiers; P is a predicate involving c and d; Q involves c, d, and g. A rule similar to 10.1a applies to function declarations:

10.2a. $R\,\{\textbf{function } f \ldots \textbf{ begin } S \textbf{ end}\}\ R$.

Function calls, unlike procedure calls, appear in expressions which are part of statements. There is no function-call statement corresponding to a procedure-call statement. The proof rule for functions depends crucially on the fact that Euclid functions have no side effects, a consequence of the absence of **var** in a function declaration. Therefore, the order of evaluation of functions within an expression does not matter.

Suppose in an expression, possibly within S itself or in other declarations in the same scope, there is a call $f(a)$ of the function f with actual parameters a. The rule

10.2. (function-call rule)

$$\frac{\begin{array}{l}[P(a1/c) \supset Q(a1/c, f(a1, d)/g), Q\,\{\textbf{return asserting } Q\}\ \text{false}, H] \\ \quad \vdash [P\{S\}\ Q, \exists\, g1(P\{S\}\ g = g1(c, d))], \\ H \supset \exists\, g(P \supset Q)\end{array}}{P(a/c) \supset Q(a/c, f(a, d)/g)}$$

may be used in verifying the properties of the expression involving $f(a)$. Since the term $f(a, d)$, rather than $f(a)$, occurs in the conclusion of the rule, applying this rule to an assertion R will first require the verifier to apply the substitution $f(a, d)\ f(a)$ to R. This rule is due to David Musser; a full discussion is in [12].

The second premise, called the consistency clause, ensures that the lemma in the conclusion of the rule will not be inconsistent. In the first premise, the $P\{S\}Q$ part gives the relation which the function's declared body, S, and its single precondition, P, and single postcondition, Q, must satisfy. The part involving $\exists g1$ is a requirement that the function be single-valued; it is discussed below. These, like the second premise, need be proved only once per function declaration. The other three parts of the premise (before the $\vdash$) are the recursion clause, the definition of the **return** statement, and the type information for each $c \in C$ and g, respectively. The **return** statement is the same as in procedures, including the "**asserting** Q" clause. The statements

return *expr* **when** B
return *expr*

are equivalent to

if B **then** $g := expr$; **return end if**
$g := expr$; **return**.

respectively.

In $\exists g1(P\{S\}\ g = g1(c, d))$, $g1$ is a mathematical function of c and d. The premise is thus equivalent to requiring that S defines a mathematical function, i.e., that it be single-valued. Note that the implicit universal quantifiers associated with formulas in the Hoare logic go inside the existential quantifier in this formula. If the function contains no module variables in its parameter or import lists, the $\exists g1$ part is automatically true because Euclid is a deterministic language.

The standard equality of Euclid modules (if equality is exported) is, informally, component-by-component (bitwise) equality of the modules' concrete representations. With respect to this equality, Euclid functions of modules are also single-valued and thus the $\exists g1$ part is again true. However, other equality relations may be needed in the verification of programs which use Euclid modules. In particular, the abstraction function of a module, A, may be used to induce an equality relation on the concrete objects, a relation that is different from the standard equality. For example, suppose a stack module uses for its concrete representation an array and a top-of-stack pointer. The stack operations push, a second push, and then a pop ought to yield the same stack as does just the first push. Using an abstraction function that ignores the "unused" part of the array (where the second pushed element remains), the single push will give a stack equal to that of push-push-pop; using the standard equality, this will not be true. Thus always using the standard equality will not suffice to verify certain programs. As another example, consider sets represented by arrays. Equal sets, by a useful abstraction function, contain identical elements although not necessarily in the same order within the array. The abstract operation of choosing an arbitrary element from the set can be implemented by returning the first element

from the array. According to set equality defined by the abstraction function, this operation is not single-valued. In such a situation, the standard algebraic simplification rules may fail since $f(s) = f(s)$ is not necessarily true. Accordingly, before using the function-call rule on Euclid functions of modules, it is necessary to prove that the function is single-valued with respect to the equality relation induced by A.

A pseudo-function type-converter is treated as a function with appropriate precondition and postcondition as defined in the Euclid report. Examples involving function calls are in Appendix 2.

Statements

Statements are classified into simple statements and structured statements. The meaning of all simple statements (except procedure calls) is defined by axioms, and the meaning of structured statements (and procedure calls) is defined in terms of rules of inference permitting the properties of the structured statement to be deduced from properties of its constituents. However, the rules of inference are formulated so as to facilitate the reverse process of deriving necessary properties of the constituents from postulated properties of the composite statement. The reason for this orientation is that in deducing proofs of properties of programs it is most convenient to proceed in a "top-down" direction.

Simple Statements

Assignment Statements

11.1. $P(y/x)\ \{x := y\}\ P$

The substitution definitions given in Sections 4, 5, and 8 apply here.

Procedure Statements

Procedure statements are explained in Section 10 on procedure declarations and calls.

Escape Statements

Return statements are explained in Section 10. Exit statements are explained in Section 12.6.

Empty Statements

11.2. $P\{\ \}\ P$

Assertion Statements

11.3. $P \wedge Q\ \{\textbf{assert}\ P\}\ P \wedge Q$

11.4. If the **checked** option is specified, we may use

$$Q\{\textbf{assert } B\}\ Q \wedge B$$

where B is a *Boolean* expression.

Structured Statements

Compound Statements

12.1. $$\frac{P_{i-1}\{S_i\}\ P_i \text{ for } i=1,\ldots,n}{P_0\{S_1; S_2; \ldots; S_n\}\ P_n}$$

If Statements

12.2. $$\frac{P \wedge B\{S_1\}\ Q,\ P \wedge \neg B\{S_2\}\ Q}{P\{\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end if}\}\ Q}$$

12.3. $$\frac{P \wedge B\{S\}\ Q,\ P \wedge \neg B \supset Q}{P\{\textbf{if } B \textbf{ then } S \textbf{ end if}\}\ Q}$$

Case Statements

12.4a. $$\frac{P \wedge (x = k_i)\{S_i\}\ Q, \text{ for } i=1,\ldots,n}{P\{\textbf{case } x \textbf{ of } k_1 \Rightarrow S_1; \ldots; k_n \Rightarrow S_n \textbf{ end case}\}\ Q}$$

12.4b. $$\frac{P \wedge (x = k_i)\{S_i\}\ Q \text{ for } i=1,\ldots,n,\ P \wedge x \textbf{ not in } (k_1,\ldots,k_n)\{S_{n+1}\}\ Q}{P\{\textbf{case } x \textbf{ of } k_1 \Rightarrow S_1; \ldots; k_n \Rightarrow S_n; \textbf{otherwise} \Rightarrow S_{n+1} \textbf{ end case}\}\ Q}$$

Note that $k_a, k_b, \ldots, k_m \Rightarrow S$ stands for $k_a \Rightarrow S;\ k_b \Rightarrow S; \ldots; k_m \Rightarrow S$. The type of x is constrained as in the section on variant records.

12.5. $$\frac{P\{\textbf{var } anyx\colon T(k_i);\ S;\ \textbf{begin var } x\colon T(k_i) := anyx;\ S_i \textbf{ end}\}\ Q, \text{ for } i=1,\ldots,n}{P\{\textbf{var } anyx\colon T(\textbf{any});\ S;\ \textbf{case } x := anyx \textbf{ of } k_1 \Rightarrow S_1; \ldots; k_n \Rightarrow S_n \textbf{ end case}\}\ Q}$$

There may be other formal parameters in T besides the single **any** (see the procedure declarations section of the Euclid report). The case expansion in the procedure declarations section of the Euclid report). The case

var $anyx$: T(**any**); S; $anyx := y$

is already covered by the assignment axiom (Rule 11.1).

Loop Statements

12.6. $$\frac{Q\{\textbf{exit asserting } Q\}\ \text{false} \vdash P\{S\}\ P}{P\{\textbf{loop } S \textbf{ end loop}\}\ Q}$$

Note that **exit** plays the same role with respect to loops that **return** plays with respect to procedures and functions (among other things, it is associated with the nearest enclosing loop and a corresponding exit assertion; and the axiom involving **exit** may not be used directly with certain module instantiations). Like **return when** B, the statement **exit when** B may be replaced by the statement

if B **then exit end if**.

For Statements

For statements may always be expanded as explained in the Euclid report. However, for simplified cases the following rules are available, where the loop body S may not contain an escape statement:

Let T be a subrange type.

12.7. $$\frac{(T.\mathit{first} \leqq x \leqq T.\mathit{last}) \wedge P([T.\mathit{first}\ldots x))\ \{S\}\ P([T.\mathit{first}\ldots x])}{P([\,])\ \{\textbf{for } x \textbf{ in } T \textbf{ loop } S \textbf{ end loop}\}\ P([T.\mathit{first}\ldots T.\mathit{last}])}$$

12.8. $$\frac{(T.\mathit{first} \leqq x \leqq T.\mathit{last}) \wedge P((x\ldots T.\mathit{last}])\ \{S\}\ P([x\ldots T.\mathit{last}])}{P([\,])\ \{\textbf{for } x \textbf{ decreasing in } T \textbf{ loop } S \textbf{ end loop}\}\ P([T.\mathit{first}\ldots T.\mathit{last}])}$$

$[u\ldots v]$ denotes the closed interval $u, \ldots, v$, i.e., the set $\{i \mid u \leqq i \leqq v\}$, and $[u\ldots v)$ denotes the half-open interval $u, \ldots, v$, i.e., the set $\{i \mid u \leqq i < v\}$. Similarly, $(u\ldots v]$ denotes the set $\{i \mid u < i \leqq v\}$. Note that $[u\ldots u) = (u\ldots u]$ is the empty set. Since x, $T.\mathit{first}$, and $T.\mathit{last}$ are constants, S cannot change x, $T.\mathit{first}$, or $T.\mathit{last}$.

Let T be a set type.

12.9. $$\frac{T_1 \leqq T \wedge x \textbf{ in } T - T_1 \wedge P(T_1)\ \{S\}\ P(T_1 + (x))}{P(( ))\ \{\textbf{for } x \textbf{ in } T \textbf{ loop } S \textbf{ end loop}\}\ P(T)}$$

Recall that ( ) are used for set brackets. Since x and T are constants, S cannot change x or T.

Epilogue

We would like to note a few points about our experiences in constructing these proof rules.

We began to axiomatize Euclid in early 1976, and essentially ended over a year later. At the start we expected an interesting but not terribly challenging project. We were half-right – it was for us interesting *and* challenging. We learned a great deal about both proof rules and Euclid – two topics we were not nearly so well versed in as we thought.

Our increased understanding of Euclid paid a clear and immediate dividend. A number of improvements to the language were made as a direct result of facts discovered during the axiomatization. (This in turn meant the need for new proof rules.) The long-term payoff of our increased understanding of proof rules is less certain. We would like to believe that were we to begin a similar project

today, we would find it considerably less painful. It is, however, far from clear to us that it would be the routine task it must eventually become if rigorous definitions of new programming languages are to be the norm. We would surely try to write the proof rules in parallel with the design rather than afterward, as we did in some instances with Euclid. We spent an inordinate amount of time and effort on problems related to modules. Had we felt free to make substantial changes to this part of Euclid, much of this time and effort could have been avoided, and the proof rules themselves simplified substantially.

A somewhat disturbing aspect of these proof rules is our lack of complete confidence in them. There are some rules with which we are still unhappy, although we know of no errors in them. Nevertheless, it would be naive for us to believe that there are no remaining errors; many bugs were found in earlier versions, and we have too much programming experience to interpret this as a good sign. Our approach to "verifying" these proof rules has been to study each rule in (as much as possible) isolation. We have informally presented (to ourselves and others) the reasons we believe each rule to be appropriate, and have tested each rule on as many distinct cases as we had the energy to look at. The inadequacies of this approach have been cogently argued in the literature on programming. What is needed is a more formal approach to validating proof rules. The work by Donahue [2] on "verifying" soundness via a complementary definition and by Clarke [1] on completeness seems a step in this direction. We would very much encourage and be happy to talk with anyone who wishes to examine rigorously the soundness and completeness of these proof rules.

Acknowledgments. We are greatly indebted to C.A.R. Hoare and Niklaus Wirth for their axiomatization of Pascal. We are grateful to them, and to Springer-Verlag, for permission to use sections of the Pascal axiomatization [9] in this paper. Suggestions, comments, and criticisms by numerous colleagues and the referees have greatly aided us; David Musser has been especially helpful. As always, responsibility for errors and problems remains with us. We appreciate the efforts of Betty Randall and Lisa Moses in typing and formatting the various versions of this work.

Appendix 1

Procedures

Consider the trivial procedure, with only one **var** parameter, one **const** parameter, and no imported variables

procedure p(**var** a: *signedInt*, b: *signedInt*) = **pre** true; **post** $a \leqq 2 * b$;
begin var c: *signedInt*; $c := 2 * b$; **if** $a > c$ **then** $a := c$ **end if end**.

Letting S stand for the body of this procedure, it is easy to prove

$$\text{true}\ \{S\}\ a \leqq 2 * b.$$

The invocation of this procedure will (in general) change the value of a in some manner dependent on the initial values of a and b (and, if present and referenced from within S, on values of imported variables).

Now the effect of any call of $p(z, w)$ is to change z such that $z \leqq 2 * w$, and using the procedure-call rule (10.1) we may validly conclude for any R that

$$\text{true} \wedge (z\# \leqq 2 * w \supset R(z\#/z))\ \{p(z, w)\}\ R.$$

In particular, R might be just $z \leqq 2 * w$ or R might involve variables other than z and w if the call $p(z, w)$ followed statements involving those other variables. The properties of a call of p are contained solely in the postcondition $a \leqq 2 * b$.

The given postcondition is not the only property that can be proved about p. For example, if the postcondition $a = \min(a', 2 * b)$, where the prime denotes the initial value of a, had been supplied with p, we could validly conclude that

$$\text{true} \wedge (z\# = \min(z, 2 * w) \supset R(z\#/z))\ \{p(z, w)\}\ R.$$

It is important to see how the rule accommodates a structured actual variable in a **var** position. For a call $p(d(i), i)$ where d is an array and R is $d(i) \leqq 2 * i$ we may validly conclude, using the original postcondition, that

$$\text{true} \wedge (d\#(i) \leqq 2 * i \supset d\#(i) \leqq 2 * i)\ \{p(d(i), i)\}\ d(i) \leqq 2 * i.$$

If the formal parameter b were also a **var** parameter, we may validly conclude

$$\text{true} \wedge (d\#(i) \leqq 2 * i\# \supset d\#(i\#) \leqq 2 * i\#)\ \{p(d(i), i)\}\ d(i) \leqq 2 * i$$

which uses $i\#$ in place of i in *three* places. In the **var** b case, by the rules of simultaneous substitution (see discussion following Rule 10.1), $Q(d\#(i)/a, i\#/b)$ is $d\#(i) \leqq 2 * i\#$ while $R(d\#(i)/d(i), i\#/i)$ is $d\#(i\#) \leqq 2 * i\#$. In the **const** b case, $Q(d\#(i)/a, i/b)$ and $R(d\#(i)/d(i))$ are both $d\#(i) \leqq 2 * i$.

Note that while making the second formal **var** has no effect on the behavior of the program, it does reduce the number of things we can prove. Though the procedure does not change the second parameter, this cannot be deduced from the postcondition associated with p.

Appendix 2

Functions

Suppose we have the function declarations

function $f(c$: *signedInt*) **returns** m: *signedInt* =
pre $P_f(c)$; **post** $Q_f(c, m)$; **begin** S_f **end**

function $g(c$: *signedInt*) **returns** m: *signedInt* =
pre $P_g(c)$; **post** $Q_g(c, m)$; **begin** S_g **end**

and suppose that we have proved of the two bodies

$$P_f\{S_f\}\ Q_f \quad \text{and} \quad P_g\{S_g\}\ Q_g$$

and proved of the pre- and postconditions

$$\exists m(P_f \supset Q_f) \quad \text{and} \quad \exists m(P_g \supset Q_g).$$

The axiom for the assignment statement

$$x := f(g(x)) + 1$$

leads to

$$R([f(g(x)) + 1]/x)\ \{x := f(g(x)) + 1\}\ R.$$

The two function calls $g(x)$ and $f(g(x))$ appear in the expression $f(g(x)) + 1$. The function-call rule applied to g requires that the precondition $P_g(x)$ be established which in turn yields the postcondition $Q_g(x, g(x))$. The latter may be used is applying the function-call rule to f to establish $P_f(g(x))$ which yields $Q_f(g(x), f(g(x)))$. The two postconditions $Q_f(g(x), f(g(x)))$ and $Q_g(x, g(x))$ may be used to establish the substituted R term. Having done all the above, we may conclude that R holds after the assignment.

As another simpler example, to show

$$P\ \{\textbf{if } f(x) > 0 \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end if}\}\ Q,$$

it suffices to show the three premises

$$P \supset P_f(x),$$

$$P \wedge f(x) > 0\ \{S_1\}\ Q,$$

$$P \wedge \neg f(x) > 0\ \{S_2\}\ Q.$$

The first premise shows that it is legal to call $f(x)$ and, by the function-call rule applied to f, to use the postcondition $Q_f(x, f(x))$ as an additional hypothesis in establishing the second and third premises of the if rule. As a concrete example of this schema, suppose we have the function declaration

function *power*(x, y: *signedInt*) **returns** z: *signedInt* =
  pre $y \geqq 0$; **post** $z = x ** y$; **begin** S **end**

and suppose that we have proved

$$y \geqq 0\ \{S\}\ z = x ** y,$$

$$\exists z(y \geqq 0 \supset z = x ** y).$$

We wish to show

$$b \geqq 1\ \{\textbf{if } power(a, b) > 0 \textbf{ then } c := \text{true} \textbf{ else } c := \text{false} \textbf{ end if}\}\ c = (a ** b > 0).$$

Since

$$b \geqq 1 \supset b \geqq 0,$$

the function-call rule applied to $power(a, b)$ yields the conclusion $power(a, b) = a ** b$. Using this equation it is easy to show

$$b \geqq 1 \wedge power(a, b) > 0\ \{c := \text{true}\}\ c = (a ** b > 0),$$

$$b \geqq 1 \wedge power(a, b) \leqq 0\ \{c := \text{false}\}\ c = (a ** b > 0),$$

that is,

$$b \geqq 1 \wedge a ** b > 0 \supset a ** b > 0,$$

$$b \geqq 1 \wedge a ** b \leqq 0 \supset \text{false} = (a ** b > 0).$$

Note that if the formal parameter y were of type *unsignedInt*, the precondition of power could be just the constant true. In this case the compiler, rather than the verifier, would have to be satisfied that $b \geqq 0$. The compiler might, of course, produce a legality assertion for the verifier.

Appendix 3

Modules

As an example of the use of the module rule, we shall use a variant of the *smallIntSet* example in [8]. Our module *smallIntSet* provides the abstraction of a set of integers in the range 1 . . 100. The abstract operations are insertion and removal of individual elements and a membership test. When a variable of type *smallIntSet* is declared, it is initialized to the empty set. The set will be represented by a *Boolean* array, S, of 100 elements,

S: **array** 1 . . 100 **of** *Boolean*.

$S(i)$ = true iff i belongs to the set. In this example { } are used for set brackets. Comment brackets around non-Euclid **pre** and **post** are omitted.

```
type smallIntSet =
    pre true
    module smallSet
    invariant true
    exports (insert, remove, has, :=)
    var S: array 1 . . 100 of Boolean

    procedure insert (i: integer) =
        pre 1 ≦ i ≦ 100 ∧ smallSet = smallSet'
        post smallSet = smallSet' ∪ {i}
        begin S(i) := true end insert

    procedure remove (i: integer) =
        pre 1 ≦ i ≦ 100 ∧ smallSet = smallSet'
        post smallSet = smallSet' ~ {i}
        begin S(i) := false end remove
```

function *has* (*i*: *integer*) **returns** *hasResult*: *Boolean* =
pre $1 \leqq i \leqq 100$
post $hasResult = (i \in smallSet)$
begin $hasResult := S(i)$ **end** *has*

initially
post $smallSet = emptySet$
begin for *j* **in** *S*. *IndexType* **loop** $S(j) :=$ false **end loop end**

abstraction function *setValue* **returns** *resultSet* = **imports** {*S*}
begin $resultSet = \{j \mid S(j) \wedge 1 \leqq j \leqq 100\}$ **end**

invariant true
end *smallIntSet*

At the point where we encounter this module definition program, we verify the asserted properties of the definition. The necessary formulas to be verified here are derived from the module definition and the first six premises of the module rule:

(1) $\text{true} \supset \text{true}$

(2) $\text{true}\{$**var** *S*: **array** 1 . . 100 **of** *Boolean*
begin for *j* **in** *S*. *IndexType*
loop $S(j) :=$ false **end loop end**$\}$ $setValue(S) = emptySet \wedge \text{true}$

(3) (insert) $1 \leqq i \leqq 100 \wedge setValue(S) = setValue(S') \wedge \text{true}\{S(i) := \text{true}\}$
$setValue(S) = setValue(S') \cup \{i\} \wedge \text{true}$

(3) (remove) $1 \leqq i \leqq 100 \wedge setValue(S) = setValue(S') \wedge \text{true}\{S(i) := \text{false}\}$
$setValue(S) = setValue(S') \sim \{i\} \wedge \text{true}$

(4) $\exists\, g1 (1 \leqq i \leqq 100 \wedge \text{true}\{hasResult := S(i)\}$
$hasResult = (i \in setValue(S)) \wedge hasResult = g1(setValue(S)))$

(5) $\exists\, hasResult\ (1 \leqq i \leqq 100 \wedge \text{true} \supset hasResult = (i \in setValue(S)))$

(6) there is no **finally**, so this premise is missing.

In the above formulas the abstraction function *setValue* has its imported identifier *S* included as an argument. We shall not go through the exercise of proving these. They are all trivially verifiable.

Now, to show

$x = 2\{$**var** *x*: *smallIntSet*; *x*. *insert*(3); *x*. *insert*(5); $y := x$. *has*(3);
x. *delete*(3); $z := x$ **end**$\}$ $z = \{5\} \wedge y = \text{true} \wedge x = 2$,

where *y* and *z* have been declared prior to *x*, we can use the remainder of the module rule.

First, to verify the declaration **var** *x*: *smallIntSet*, we need only show

(7) $x = 2 \supset \text{true}$.

Having verified this, and premises 1–6, we may now derive and, by premise 8 in the module rule, assume and use the formulas (where *x1*, *x2*, *x3*, and *x4* are fresh variables)

(8.1 a) $\text{true} \supset (1 \leqq 3 \leqq 100 \wedge (x1 = x \cup \{3\} \supset x1 = \{3\} \wedge x\# = 2))$
$\{x$. *insert*(3)$\}$ $x = \{3\} \wedge x\# = 2$

(8.1 b) $\text{true} \supset (1 \leqq 5 \leqq 100 \wedge (x2 = x \cup \{5\} \supset x2 = \{3, 5\} \wedge x\# = 2))$
$\{x$. *insert*(5)$\}$ $x = \{3, 5\} \wedge x\# = 2$

(8.1 c) $\text{true} \supset (1 \leqq 3 \leqq 100 \wedge (x3 = x \sim \{3\} \supset x3 = \{5\} \wedge x\# = 2 \wedge y = \text{true}))$
$\{x$. *delete*(3)$\}$ $x = \{5\} \wedge x\# = 2 \wedge y = \text{true}$

(8.2) $(\text{true} \supset 1 \leqq 3 \leqq 100) \supset x$. *has*(3) $= 3 \in x$

(8.3) $\text{true} \wedge (x4 = emptySet \supset x4 = emptySet \wedge x\# = 2)$ $\{x$. *Initially*$\}$
$x = emptySet \wedge x\# = 2$, i.e., $x\# = 2\{x$. *Initially*$\}$ $x = emptySet \wedge x\# = 2$.

(8.4) This premise is missing because there is no **finally**.

We now instantiate premise 8.5 by setting *S* to be

x. *insert*(3); *x*. *insert*(5); $y := x$. *has*(3); *x*. *delete*(3); $z := x$ **end**

and *R* to be

$x = \{5\} \wedge y = \text{true} \wedge x = 2$

obtaining

(8.5) $x\# = 2\{x$. *Initially*; *S*$\}$ $z = \{5\} \wedge y = \text{true} \wedge x\# = 2$.

We now prove premise 8.5 using the formulas 8.1 a–8.3. Using (8.3) we reduce the formula to be proved to

$x\# = 2 \wedge x = emptySet\ \{S\}\ z = \{5\} \wedge y = \text{true} \wedge x\# = 2$.

Next, using (8.1 a), we get

$x\# = 2 \wedge x = \{3\}\ \{x$. *insert*(5); $y := x$. *has*(3); *x*. *delete*(3); $z := x\}$
$z = \{5\} \wedge y = \text{true} \wedge x\# = 2$.

Then, by (8.1 b), we get

$x\# = 2 \wedge x = \{3, 5\}\ \{y := x$. *has*(3); *x*. *delete*(3); $z := x\}$
$z = \{5\} \wedge y = \text{true} \wedge x\# = 2$.

By (8.2) and the assignment axiom, the problem reduces to

$x\# = 2 \wedge x = \{3, 5\} \wedge y = \text{true}\{x$. *delete*(3); $z := x\}$ $z = \{5\} \wedge y = \text{true} \wedge x\# = 2$.

By (8.1 c) we get

$x\# = 2 \wedge x = \{5\} \wedge y = \text{true}\{z := x\}$ $z = \{5\} \wedge y = \text{true} \wedge x\# = 2$

which, by the assignment axiom, is true.

Since the module *smallIntSet* imports no **var** variables, the instantiation of premise 8.5 could have been from the simpler form of premise 8. If this were done, there would be no formula 8.3 to be used.

References

1. Clarke, E.M. Jr.: Programming language constructs for which it is impossible to obtain good Hoare-like axiom systems. Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, pp. 10–20. New York: ACM 1977
2. Donahue, J.E.: Complementary definitions of programming language semantics. In: Lecture Notes in Computer Science, Vol. 42. Berlin-Heidelberg-New York: Springer 1976
3. Ernst, G.W.: Rules of inference for procedure calls. Acta Informat. **8**, 145–152 (1977)
4. Guttag, J.V., Horning, J.J., London, R.L.: A proof rule for Euclid procedures. In: Formal Description of Programming Concepts (E. Neuhold, ed.), pp. 211–220. Amsterdam-New York-Oxford: North-Holland 1978. Also USC Information Sciences Institute, Technical Report ISI/RR-77-60, May 1977
5. Guttag, J.V., Horowitz, E., Musser, D.R.: Abstract data types and software validation. Comm. ACM (to appear). Also: USC Information Sciences Institute, Technical Report ISI/RR-76-48, August 1976
6. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. ACM **12**, 576–580, 583 (1969)
7. Hoare, C.A.R.: Procedures and parameters: An axiomatic approach. In: Symposium on Semantics of Algorithmic Languages (E. Engeler, ed.), Lecture Notes in Mathematics, Vol. 188, pp. 102–116. Berlin-Heidelberg-New York: Springer 1971
8. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informat. **1**, 271–281 (1972)
9. Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language PASCAL. Acta Informat. **2**, 335–355 (1973)
10. Jensen, K., Wirth, N.: PASCAL user manual and report. Lecture Notes in Computer Science, Vol. 18, 2nd ed. Berlin-Heidelberg- New York: Springer 1975
11. Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., Popek, G.J.: Revised report on the programming language Euclid. Xerox Research Center, Technical Report CSL 78-2, 1978. An earlier version appeared in: SIGPLAN Notices **12**, No. 2 (February 1977)
12. Musser, D.R.: A proof rule for functions. USC Information Sciences Institute, Technical Report ISI/RR-77-62, October 1977
13. Popek, G.J., Horning, J.J., Lampson, B.W., Mitchell, J.G., London, R.L.: Notes on the design of Euclid. Proceedings of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina. SIGPLAN Notices **12**, No. 3, 11–18 (1977)
14. Spitzen, J., Wegbreit, B.: The verification and synthesis of data structures. Acta Informat. **4**, 127–144 (1975)
15. Wirth, N.: The programming language PASCAL. Acta Informat. **1**, 35–63 (1971)

Received June 9, 1977

Note Added in Proof

There may be some inconsistence between these proof rules and the latest version of the Euclid report [11]. The latter was recently revised.