Proof Rules for the Programming Language Euclid


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 reprint 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 F44620-76-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 seman-
tically 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
declared 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
other constructs.

Proof Rules for the Programming Language Euclid

In the proof rules, $S$ and $S_i$ 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

$$P \Rightarrow Q, Q \{(S) U, U \Rightarrow R \}

\frac{}{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 declara-
   tions. 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. $(x < y) \land (y < z) \Rightarrow (x < z)$
1.7. $(x \neq c_i) \Rightarrow (x < T. \ succ(x))$
1.8. $x > y \Leftrightarrow y < x$
1.9. \( x \leq y \equiv \neg(x > y) \)
1.10. \( x \geq y \equiv \neg(x < y) \)
1.11. \( x \neq y \equiv \neg(x = y) \)
1.12. \( T.\ first = e_1 \)
1.13. \( T.\ last = e_n \)
1.14. \( T.\ ord(e_i) = i - 1 \) for \( i = 1, \ldots, n \).

The standard scalar type \textit{Boolean} is defined as

\texttt{type Boolean = (False, True).}

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

\[ x \texttt{ and } y \text{ means if } x \text{ then } y \text{ else false, i.e., } x \text{ and } y, \]
\[ x \texttt{ or } y \text{ means if } x \text{ then true } \text{ else } y, \text{ i.e., } x \text{ or } y, \]
\[ x \rightarrow y \text{ means if } x \text{ then } y \text{ else true.} \]

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

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

\[ m \texttt{ mod } n = m - (m \texttt{ div } n) \times n \]

whereas \texttt{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, \texttt{signedInt} or \texttt{unsignedInt}.

**Type Char**

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

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

\[ \$ A \lt \$ B \lt \$ C \lt \$ D \lt \ldots \lt \$ Y \lt \$ Z \]

Axioms 1.5–1.13 apply to the \texttt{char} type. The functions \texttt{char . ord} and \texttt{Chr} are defined by the following additional axioms:

\[ \texttt{Chr(char . ord}(u) = u \]

2.2. \( A \lt B \lt C \lt D \lt \ldots \lt Y \lt Z \)

Axioms 1.5–1.13 apply to the \texttt{char} type. The functions \texttt{char . ord} and \texttt{Chr} are defined by the following additional axioms:

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

\[ \texttt{Chr(char . ord}(u) = u \]

2.4. \( u < v \equiv \texttt{char . ord}(u) < \texttt{char . ord}(v) \).

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

**Subrange Types**

\texttt{type T=\ldots m \ldots 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 \( m \leq i \leq n \), \( i \) is an element of \( T \).
3.4. These are the only elements of \( T \).
3.5. \( T.\ first \leq i \leq T.\ last \Rightarrow T.\ succ(i) = T_0.\ succ(i) \)
3.6. \( T.\ first < i \leq T.\ last \Rightarrow T.\ pred(i) = T_0.\ pred(i) \).

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

**Array Types**

\texttt{type T=\ldots 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 \( m \leq i \leq n \), 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 \( m \leq i \leq n \).

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

\[ (x, i; y) \text{ stands for } T(x_m, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_n). \]

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

\[ (x, \langle i_1, \ldots, i_n \rangle ; y) \text{, where } n \geq 2 \]
to be equal to
\[(x, \langle i_1, \ldots, i_{n-1} \rangle) \cdot (x(i_1) \ldots (i_{n-1}), i_n; y))\]
and the formula
\[P(y/x (i_1) \ldots (i_n)), \text{ where } n \geq 2\]
is equivalent to
\[P((x, \langle i_1, \ldots, i_n \rangle; y)/x)\].

4.4. \(x\). **IndexType** = 1, \(x\). **ComponentType** = \(T_0\).

**Record Types**

\[\textbf{type } T = \textbf{record } s_1 : T_1, \ldots, s_m : T_m \textbf{ end } T\]

where each \(s_i\) must be preceded by \textbf{const} or \textbf{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. \(T(x_1, \ldots, x_m) \cdot s_i = x_i \) for \(i = 1, \ldots, m\).

\((x, s; y)\) 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**

\[\textbf{type } T_n(c) = \textbf{record } s_1 : T_1, \ldots, s_m : T_m ;\]

\[\textbf{case } c \textbf{ of }\]

\[\quad k_1 \Rightarrow s'_1 : T'_1 \textbf{ end } k_1\]

\[\quad k_2 \Rightarrow s'_2 : T'_2 \textbf{ end } k_2\]

\[\vdots\]

\[\quad k_n \Rightarrow s'_n : T'_n \textbf{ end } k_n\]

\[\textbf{end case}\]

where the type of \(c\) is a scalar or subrange type with elements \(k_1, \ldots, k_n\). Note that \(k_n, k_{n-1}, \ldots, k_1 \Rightarrow S\) stands for \(k_n \Rightarrow S; k_{n-1} \Rightarrow S; \ldots; k_1 \Rightarrow S\).

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

5.4a. \(T(x_1, \ldots, x_m, k_j, x'_j)\) is an element of \(T\).

5.5a. Letting \(z\) stand for \(T(x_1, \ldots, x_m, k_j, x'_j)\), the standard Euclid component \(\textit{itsTag}\) is defined by \(z\). \(\textit{itsTag} = k_j\).

A variant record may also contain an \textbf{otherwise} clause

\[\textbf{otherwise } \Rightarrow s'_{n+1} : T'_{n+1}\]

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 \textbf{otherwise} \((n + 1)\) situation:

5.1b. \(T(x_1, \ldots, x_m, k_j, x'_{n+1})\) is also an element of \(T\).

5.4b. \(T(x_1, \ldots, x_m, k_j, x'_n, x'_{n+1}) \cdot s'_{n+1} = x'_{n+1}\)

5.5b. Identical to 5.5a.

The case with a field list containing several fields

\[k_j \Rightarrow s_{j1} : T_{j1}, \ldots, s_{jn} : T_{jn} \textbf{ end } k_j\]

is to be interpreted as

\[k_j \Rightarrow s'_j : T'_j \textbf{ end } k_j\]

where \(s'_j\) is a fresh identifier, and \(T'_j\) is a type defined as

\[\textbf{type } T'_j = \textbf{record } s_{j1} : T_{j1}, \ldots, s_{jn} : T_{jn} \textbf{ end record} .\]

In this case \(x . s'_j\) is interpreted as \(x . s'_j . s_{jn}\).

Now consider type \(T = T_0(\textit{any})\).

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**

\[\textbf{type } T = \textbf{set of } T_0\]

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. \( T : \text{BaseType} = T_0 \).

\( T(\ ) \) denotes the empty set, and \( T(x_0) \) denotes the singleton set containing \( x_0 \).

The operators \(+, *, -\), \text{xor}, and \text{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.

\[
\text{type } T(\text{const } C) = \text{pre } P1; \text{ post } Q1; \text{ module } \nu \text{ invariant } Q0:
\]

\[
\text{imports (var } Y \text{ const } D \text{ readonly } R)
\]

\[
\text{exports (const } (K1, i, f) \text{ var } V1 \text{ readonly } R1)
\]

\[
\text{const } K; \text{ var } V
\]

\[
\text{procedure } p(\text{var } X2 \text{ nonvar } C2) = \text{imports (var } Y2 \text{ nonvar } D2)
\]

\[
\text{pre } P2; \text{ post } Q2; \text{ begin } S2 \text{ end}
\]

\[
\text{function } f(\text{nonvar } C3) \text{ returns } G = \text{imports (nonvar } D3)
\]

\[
\text{pre } P3; \text{ post } Q3; \text{ begin } S3 \text{ end}
\]

\[
\text{initially } \text{imports (var } Y4 \text{ nonvar } D4) \text{ post } Q4; \text{ begin } S4 \text{ end}
\]

\[
\text{abstraction function } A \text{ returns } t_0 = \text{imports (nonvar } D5)
\]

\[
\text{begin } S5 \text{ end}
\]

\[
\text{invariant } Q
\]

\[
\text{finally } \text{imports (var } Y6 \text{ nonvar } D6) \text{ pre } P6; \text{ begin } S6 \text{ end}
\]

\[
\text{end } T
\]

The parts of a module definition are explained in the section on module types in the Euclid report. Nonvar denotes the list of \text{const} and \text{readonly} identifiers. Each of \( c, y, d, r, k1, v1, r1, k, r, x_2, c_2, y_2, d_2, c_3, d_3, y_4, d_4, d_5, y_6, \) and \( d_6 \) 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, \( v, v \) 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

\[
\text{1, 2, 3, 4, 5, 6.}
\]

\[
7.
\]

\[
[8.1, 8.2, 8.3, 8.4] \vdash 8.5
\]

\[
P(\text{var } x: T(a); S; R \land Q1)
\]

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 \text{initially} establishes the postcondition of \text{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: \textbf{finally} establishes the postcondition of the module.

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

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 \textbf{initially}, \textbf{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, \textbf{initially}, and \textbf{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 with a data type (generator) induction.) Formula 8.3 treats \( x \). \textbf{Initially} as a parameterless procedure call that establishes the abstract invariant. Formula 8.4 treats \( x \). \textbf{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 moduless, then \( x \). \textbf{Initially} and \( x \). \textbf{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) \( Q \supseteq Q_0(A/t) \),
(2) \( P_1 \{ \text{const} K; \text{var} V \}; S_1 \} Q_1(A/t) \land Q \),
(3) \( P_2(A/t) \land Q \{ S_2 \} Q_2(A/t) \land Q \),
(4) \( g \{ P_3 \{ A/t \} \land Q \{ S_3 \} Q_3(A/t) \land g = g_1(A, c, d) \} \),
(5) \( g \{ P_3 \{ A/t \} \land Q \Rightarrow Q_3(A/t) \} \),
(6) \( P_6(A/t) \land Q \{ S_6 \} Q_1 \),
(7) \( P \Rightarrow P_1(a/c) \).

(8.1) \( Q_0(a/c, x/t, x'/t') \Rightarrow (P_2(x/t, x'/t', a/2x2, c2/c2, a/c) \land (Q_2(x#2/t, x'/t', a#2/x2, c2/c2, a/c, y#2/y2, a#2/x', y#2/y2) \Rightarrow R_1(x#2/t, a#2/x2, y#2/y2)) \{ x, p(a, c2) \} R_1 \land Q_0(a/c, x/t, x'/t') \).

(8.2) \( Q_0(a/c, x/t) \Rightarrow P_3(x/t, a/3x3, a/c) \Rightarrow (Q_3(x/t, a/3x3, a/c, f(a, d3)/g) \land Q_0(a/c, x/t) \).

(8.3) \( P_1(a/c) \land Q_4(x#4/t, x'/t', a/c, y#4/y4, y#4/y4) \Rightarrow R_4(x#4/t, y#4/y4) \{ x \}. \text{Initially} R_4 \land Q_0(a/c, x/t, x'/t') \).

(8.4) \( Q_0(a/c, x/t, x'/t') \Rightarrow P_6(x/t, x'/t', a/c) \land (Q_1(a/c, y#6/y6, y#6/y6) \Rightarrow R(y#6/y6)) \{ x. \text{Finally} \} R \).

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

\[ \begin{align*}
\text{Proof Rules for the Programming Language Euclid} \\
Q_0(a/c, x/t, x'/t'). \text{Since a module may not import its own name, premise 2 is never "recursively applied" in } T. \text{ In the interest of simplicity, the formulas for recurrence, return, and } H \text{ 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 \textbf{var} variables, i.e., \textbf{initially} and \textbf{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

\[ [8.1, 8.2] \vdash P(x#x) \land Q_4(x/t) \land Q_0(x/t) \{ S \} R(x#x) \]

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.

\textbf{Pointer and Collection Types}

\textbf{type} \( T_0 = \text{collection of } T' \)

\textbf{var} \( C : T_0 \)

\textbf{type} \( T = \uparrow C \)

8.1. \( C. \text{nil} \) 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, \( \neq C. \text{nil} \), 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 \( \pi_i \) denotes the component \( \beta_i \) 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 \( \pi \neq C. \text{nil} \) of type \( T, \pi \uparrow C(\pi) \).

We introduce the following abbreviation:

\[ \text{If } C = T_0(\{ \pi_1 : \beta_1, \ldots, \pi_n : \beta_n \}) \text{ then } (C, \pi_i, y) \text{ stands for } T_0(\{ \pi_1 : \beta_1, \ldots, \pi_n : \beta_n \} - \{ \pi_i : \beta_i \} + \{ \pi_i : y \}) \]

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

8.7. \( C. \text{New}(t) \) means \( i \mapsto \pi \)

where \( \pi \) is an element of \( T, \pi \neq \pi_i \) for \( i = 1, \ldots, n \).
and \( C := T_0(\pi_1: \beta_1, \ldots, \pi_n: \beta_n + \{x: \beta\}) \) where \( \beta \) is undefined (and may not be referenced).

8.8. \( C. \text{Free}(t) \) means
\[
C := T_0(\pi_1: \beta_1, \ldots, \pi_n: \beta_n + \{t: C(i)\}) \quad \text{(recall \( C(t) \) means \( t \uparrow \)}
\]
\[ t := C. \text{nil}. \]

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. \( x\# . \text{itsType} = T, x\# \in T \vdash P[S(x\#/x)] \quad Q \)
\[
P[\text{var } x: T; S] \quad Q
\]
where \( \text{itsType} \) 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 \text{initially} or \text{finally} clauses modify imported \text{var} identifiers. The separate rules required to cover these structured types are omitted.

9.2. \( P(c/x) \{ \text{const } x := c \} P \).

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

9.3a. \( x = y \land P(x\#/y) \{ S \} Q(x\#/x, x/y) \)
\[
P[\text{bind varBindingCondition } x \text{ to } y; S] \quad Q
\]
9.3b. \( i = i_0 \land x = y(i) \land P(x\#/y) \{ S \} Q(x\#/x, y; y:i; x/y) \)
\[
P[\text{bind varBindingCondition } x \text{ to } y(i); S] \quad Q
\]
where, in all cases, \( i_0 \) and \( x\# \) are fresh variables and where \( \text{varbindingCondition} \) is \text{readonly}, \text{var} or empty.

9.4. \( H \vdash P[S] \quad Q \)
\[
P[\text{type } T = \ldots; S] \quad Q \]
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

\textbf{type} \( T(c) = \text{pre } P_1; \text{Defn} \)

9.5. \( P \Rightarrow P_1(a/c), P[\text{const } c = a; \text{type } T' = \text{Defn}; \text{var } x: T'; S] \quad Q \)
\[
P[\text{var } x: T(a); S] \quad Q
\]
\textbf{type} \( T''(\ldots, x, \ldots) = \cdots \)
\textbf{type} \( T_0 = \text{collection of } T'(\ldots, \text{unknown}, \ldots) \)
\textbf{var} \( C: T_0 \)
\textbf{type} \( T = \uparrow C \)
\textbf{var} \( t: T \)

If \( T \) is referenced as the object type of a collection, then one or more of the actual parameters may be specified as \text{unknown}. In such cases the component of \( H \) (in Rule 9.4) associated with \( t \) is the disjunction
\[
\bigwedge_{a \in x + \text{unknown}} [t \in \text{collection of } T'(\ldots, a, \ldots)].
\]

9.6. \( C. \text{New}(t, y) \) means all of Section 8.7 except that the type of \( \beta \) is \( T'(\ldots, y, \ldots) \).

Procedure Declarations and Calls

\textbf{procedure} \( p[\text{var } X, \text{nonvar } C] = \text{imports } (\text{var } Y, \text{nonvar } D) \)
\textbf{pre } P \textbf{; post } Q \textbf{; begin } X \textbf{ end} \)

Nonvar denotes the list of \text{const} and \text{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)
\[
[P(a/l, x, \ell/c) \land (Q(a\#l/x, e\#/y, a\#l/x', y/y') \Rightarrow RI(a\#l, a\#l, y/y')) \Rightarrow R] (a/l, e/l)] \Rightarrow R, Q[\text{return asserting } Q] \text{ false, } H \]
\[
\vdash x = x' \land y = y' \quad P[S] \quad Q
\]
\[
P(a/x, c/e/c) \land (Q(a\#x, c\#e, y/y, a\#x', y/y') \Rightarrow R(a\#a, y/y')) \Rightarrow R \quad P(e/c) \quad R
\]

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
\[
[Q[\text{return asserting } Q] \text{ false, } H] \vdash x = x' \land y = y' \quad P[S] \quad Q.
\]
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 9.2. 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. $R(a_1 \# a_2 \# \ldots \# a_n)$ may evaluate to a formula of the form

$$R(a_{(\# a_1, \ldots, \# a_n)} \# (i_1, \ldots, i_n)),$$

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

$$R((B, \langle i_1, \ldots, i_m \rangle): a_{(\# a_1, \ldots, \# a_n)} \# (i_1, \ldots, i_n)).$$

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

$$R((B, \langle i_1, \ldots, i_m \rangle): a_{(\# a_1, \ldots, \# a_n)} \# (i_1, \ldots, i_n)) = R((B, \langle i_1, \ldots, i_m \rangle): a_{(\# a_1, \ldots, \# a_n)} \# (i_1, \ldots, i_n)).$$

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

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

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. This "recursion" clause note that the symbols are deliberately different from those in the rule's conclusion: $R!$ replaces $R$, and $a_1$ and $a_2$ 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. $R \{\text{procedure } p \ldots \text{begin } S \text{ end }\} R$.

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. $R \{\text{function } f \ldots \text{begin } S \text{ end }\} R$.

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)

$$[P(a_1/c) \Rightarrow Q(a_1/c, f(a_1, d)/g), Q\{\text{return asserting } Q\} \text{ false}, H]$$

$$\vdash [P[S] Q, g \downarrow (P[S] g = g l(c, d))].$$

$$H \Rightarrow g(P \Rightarrow Q)$$

$$P(a/c) \Rightarrow Q(a/c, f(a, d)/g)$$
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 \( P \land Q \) part gives the relation which the functions declared body, \( S \), and its single precondition, \( P \), and single postcondition, \( Q \), must satisfy. The part involving \( \exists g \) 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 \( \exists g \)) are the recursion clause, the definition of the return statement, and the type information for each \( e \in C \) and \( g \), respectively. The return statement is the same as in procedures, including the "asserting \( Q \)" clause. The statements

\[
\text{return } expr \text{ when } B \\
\text{return } expr
\]

are equivalent to

\[
\text{if } B \text{ then } g := expr; \text{return end if}
\]

\[
g := expr; \text{return}
\]

respectively.

In \( \exists g \in P[S] g = g_l(c, d) \), \( g_l \) 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 \( \exists g_l \) 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 \( \exists g_l \) 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. \( P \{ \text{if } x := y \} \ P \)

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. \( P \land Q \{ \text{assert } P \} \ P \land Q \)
11.4. If the checked option is specified, we may use
\[ Q \{\text{assert } B\} \quad Q \wedge B \]
where \( B \) is a Boolean expression.

**Structured Statements**

**Compound Statements**

12.1. \[ P_{i-1} \{S_i\} \quad P_i \text{ for } i = 1, \ldots, n \]
\[ P_n \{S_1; S_2; \ldots; S_n\} \quad P_n \]

**If Statements**

12.2. \[ P \wedge B \{S_i\} \quad Q, \quad P \wedge \neg B \{S_i\} \quad Q \]
\[ P \{\text{if } B \text{ then } S_i \text{ else } S_i \text{ end if}\} \quad Q \]

12.3. \[ P \wedge B \{S\} \quad Q, \quad P \wedge \neg B \Rightarrow Q \]
\[ P \{\text{if } B \text{ then } S \text{ end if}\} \quad Q \]

**Case Statements**

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

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

Note that \( k_1, k_2, \ldots, k_n \Rightarrow S \) stands for \( k_1 \Rightarrow S; \quad k_2 \Rightarrow S; \quad \ldots; \quad k_n \Rightarrow S \). The type of \( x \) is constrained as in the section on variant records.

12.5. \[ P \{\text{var } \text{anyx: } T(k_i); \quad \text{begin var } x: T(k_i):=\text{anyx}; \quad \text{end}\} \quad Q, \text{ for } i = 1, \ldots, n \]
\[ P \{\text{var } \text{anyx: } T(\text{any}); \quad \text{case } x:=\text{anyx} \text{ of } k_1 \Rightarrow S_1; \ldots; k_n \Rightarrow S_n \text{ end case}\} \quad Q \]

There may be other formal parameters in \( T \) besides the single any (see the expansion in the procedure declarations section of the Euclid report). The case
\[ \text{var } \text{anyx: } T(\text{any}); \quad \text{anyx: } = y \]
is already covered by the assignment axiom (Rule 11.1).

**Loop Statements**

12.6. \[ Q \{\text{exit asserting } Q\} \quad \text{false} \quad P \{S\} \quad P \]
\[ P \{\text{loop } S \text{ end loop}\} \quad Q \]

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
\[ \text{if } B \text{ then exit end if.} \]

**For Statements**

For statements may be always 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. \( (T. \text{ first} \leq x \leq T. \text{ last}) \wedge P([-T. \text{ first} \ldots x]) \{S\} \quad P([-T. \text{ first} \ldots x]) \)
\[ P \{[\ldots]\} \quad \{\text{for } x \text{ in } T \text{ loop } S \text{ end loop}\} \quad P([-T. \text{ first} \ldots T. \text{ last}]) \]

12.8. \((T. \text{ first} \leq x \leq T. \text{ last}) \wedge P([x \ldots T. \text{ last}]) \{S\} \quad P([x \ldots T. \text{ last}]) \)
\[ P \{[\ldots]\} \quad \{\text{for } x \text{ decreasing in } T \text{ loop } S \text{ end loop}\} \quad P([-T. \text{ first} \ldots T. \text{ last}]) \]

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

Let \( T \) be a set type.

12.9. \( T_1 \leq T \times x \in T - T_1 \wedge P(T_1) \{S\} \quad P(T_1 + (x)) \)
\[ P \{[\ldots]\} \quad \{\text{for } x \text{ in } T \text{ loop } S \text{ end loop}\} \quad P(T) \]

Recall that \( (\ldots) \) 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 remaining errors. 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

\[
\text{procedure } p(\text{var } a; \text{signedInt}, b; \text{signedInt}) = \text{pre } a \leq 2 \ast b; \text{post } a \leq 2 \ast b; \begin{var} c \colon \text{signedInt}; c = 2 \ast b; \text{if } a > c \text{ then } a := c \text{ end if end.} \end{var}
\]

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

\[\{S\} \ a \leq 2 \ast b.\]

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 \( z \leq 2 \ast w \), and using the procedure-call rule (10.1) we may validly conclude for any \( R \)

\[\text{true} \land (z \leq 2 \ast w \Rightarrow R(z \leq z)) \{p(z, w)\} R.\]

In particular, \( R \) might be just \( z \leq 2 \ast w \) 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 \( a \leq 2 \ast b \).

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

\[\text{true} \land (z \leq \min(z, 2 \ast w) \Rightarrow R(z \leq z)) \{p(z, w)\} R.\]

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 \( d(i) \leq 2 \ast i \) we may validly conclude, using the original postcondition, that

\[\text{true} \land (d(i) \leq 2 \ast i \Rightarrow \exists d(i) \leq 2 \ast i) \{p(d(i), i)\} d(i) \leq 2 \ast i.\]

If the formal parameter \( b \) were also a var parameter, we may validly conclude

\[\text{true} \land (d(i) \leq 2 \ast i \Rightarrow \exists d(i) \leq 2 \ast i) \{p(d(i), i)\} d(i) \leq 2 \ast i,\]

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), \( Q(d(i) \leq a, i \leq b) \) is \( d(i) \leq 2 \ast i \) while \( R(d(i) \leq a, i \leq b) \) is \dagger. In the const \( b \) case, \( Q(d(i) \leq a, i \leq b) \) and \( R(d(i) \leq a, i \leq b) \) both \( d(i) \leq 2 \ast i \). Note that while making the second formal \( b \) 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

\[
\text{function } f(c; \text{signedInt}) \text{ returns } m; \text{signedInt} = \text{pre } P_f(c); \text{post } Q_f(c, m); \text{begin } S_f \text{ end}\]

\[
\text{function } g(c; \text{signedInt}) \text{ returns } m; \text{signedInt} = \text{pre } P_g(c); \text{post } Q_g(c, m); \text{begin } S_g \text{ end}\]

and suppose that we have proved of the two bodies

\[P_f(S_f) \ Q_f \text{ and } P_g(S_g) \ Q_g\]
and proved of the pre- and postconditions

\[ \exists! m (P_1 \Rightarrow Q_1) \quad \text{and} \quad \exists! m (P_2 \Rightarrow Q_2). \]

The axiom for the assignment statement

\[ x := f(g(x)) + 1 \]
leads to

\[ R(\{ f(g(x)) + 1 \} / x) \]

The two function calls \( g(x) \) and \( f(g(x)) \) appear in the expression \( f(g(x)) + 1 \). The function-call rule applied to \( f \) requires that the precondition \( P_2(x) \) be established which in turn yields the postcondition \( Q_2(x, g(x)) \). The latter may be used in applying the function-call rule to \( f \) to establish \( P_2(g(x)) \) which yields \( Q_2(g(x), f(g(x))) \). The two postconditions \( Q_2(g(x), f(g(x))) \) and \( Q_2(x, g(x)) \) 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

\[ P \{ \text{if } f(x) > 0 \text{ then } S_1 \text{ else } S_2 \text{ end if } \} \]

it suffices to show the three premises

\[ P \Rightarrow P_2(x), \]
\[ P \land f(x) > 0 \Rightarrow S_1 \]
\[ P \land \neg f(x) > 0 \Rightarrow S_2 \]
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 \( Q_2(x, f(x)) \) 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

\begin{verbatim}
function power(x: y: signedInt) returns z: signedInt =
  pre y >= 0; post z = x ** y; begin S end
\end{verbatim}

and suppose that we have proved

\[ y \geq 0 \Rightarrow z = x ** y \]
\[ \exists z (y \geq 0 \Rightarrow z = x ** y) \]

We wish to show

\[ b \geq 1 \text{ if } power(a, b) > 0 \text{ then } c := \text{true else } c := \text{false end if } c = (a ** b > 0) \]

Since

\[ b \geq 1 \Rightarrow b \geq 0 \]

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

\[ b \geq 1 \land power(a, b) > 0 \{ c := \text{true} \} c = (a ** b > 0), \]
\[ b \geq 1 \land power(a, b) \leq 0 \{ c := \text{false} \} c = (a ** b > 0), \]

that is,

\[ b \geq 1 \land a ** b > 0 \Rightarrow a \land b \geq 0, \]
\[ b \geq 1 \land a ** b \leq 0 \Rightarrow \text{false} = (a ** b > 0). \]

Note that if the formal parameter \( y \) were of type \textit{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 \( b \geq 0 \). 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 \textit{smallIntSet} example in [8]. Our module \textit{smallIntSet} provides the abstraction of a set of integers in the range \( 1 \ldots 100 \). The abstract operations are insertion and removal of individual elements and a membership test. When a variable of type \textit{smallIntSet} is declared, it is initialized to the empty set. The set will be represented by a \textit{Boolean} array, \( S \), of 100 elements.

\[ S: \text{array} 1 \ldots 100 \text{ of Boolean.} \]

\( S(i) = \text{true iff } i \text{ belongs to the set. In this example \{ \} are used for set brackets.} \]

Comment brackets around non-Euclid \texttt{pre} and \texttt{post} are omitted.

\begin{verbatim}
type smallIntSet =
  pre true
module smallSet
  invariant true
  exports (insert, remove, has, :=)
var S: array 1 \ldots 100 of Boolean
procedure insert (i: integer) =
  pre 1 \leq i \leq 100 \land smallSet = smallSet'
  post smallSet = smallSet' \cup \{i\}
  begin S(i) := true end insert

procedure remove (i: integer) =
  pre 1 \leq i \leq 100 \land smallSet = smallSet'
  post smallSet = smallSet' \setminus \{i\}
  begin S(i) := false end remove
\end{verbatim}
function has(i: integer) returns hasResult: Boolean =
pre 1 ≤ i ≤ 100
post hasResult = (i ∈ 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 | S(j) ∧ 1 ≤ j ≤ 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) true ⊢ true
(2) true {var S: array 1..100 of Boolean
begin for j in S. IndexType
loop S(j) := false end loop end} setValue(S) = emptySet ∧ true
(3) (insert) 1 ≤ i ≤ 100 ∧ setValue(S) = setValue(S') ∧ true {S(i) := true
setValue(S) = setValue(S') ∪ {i} ∧ true
(3) (remove) 1 ≤ i ≤ 100 ∧ setValue(S) = setValue(S') ∧ true {S(i) := false
setValue(S) = setValue(S') ∖ {i} ∧ true
(4) ∃ g1(1 ≤ i ≤ 100 ∧ true {hasResult := S(i)
hasResult = (i ∈ setValue(S)) ∧ hasResult = g1(setValue(S))
(5) ∃ hasResult (1 ≤ i ≤ 100 ∧ true ⊢ hasResult = (i ∈ 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} ∧ y = true ∧ 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 ⊢ 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)
References

15. Wirth, N.: The programming language PASCAL. Acta Informatica 1, 35-63 (1971)

Received June 9, 1977

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

The Algebraic Semantics of Abstract Data Types

J.V. Guttag*
Computer Science Department
J.J. Horning**
Computer Systems Research Center

Summary. The class constructs of many programming languages, e.g., [15], [16], are built by binding together the operations of an abstract data type. Our operation on abstract data types is to offer no representation and to make possible the composition of other operations. The class abstract data type comprises the components of the class, the method to supply the objects of the class, and the list of operations. For example, define a list

```
NEW:  C
ADD:  C
FRONT:  C
REMOVE:  C
EMPTY?:  C
```

* Supported in part by the Joint Services Electronics Program under contract F44620
** Supported in part by the Research Center, 333