7-1987

Theory of Modules

J. D. Gannon

Richard G. Hamlet

Harlan D. Mills

Theory of Modules

JOHN D. GANNON, RICHARD G. HAMLET, MEMBER, IEEE,
AND HARLAN D. MILLS, SENIOR MEMBER, IEEE

Abstract—**Because large-scale software development is a struggle against internal program complexity, the modules into which programs are divided play a central role in software engineering. A module encapsulating a data type allows the programmer to ignore both the details of its operations, and of its value representations. It is a primary strength of program proving that as modules divide a program, making it easier to understand, so do they divide its proof. Each module can be verified in isolation, then its internal details ignored in a proof of its use. This paper describes proofs of module abstractions based on functional semantics, and contrasts this with the Alphard formalism based on Hoare logic.**

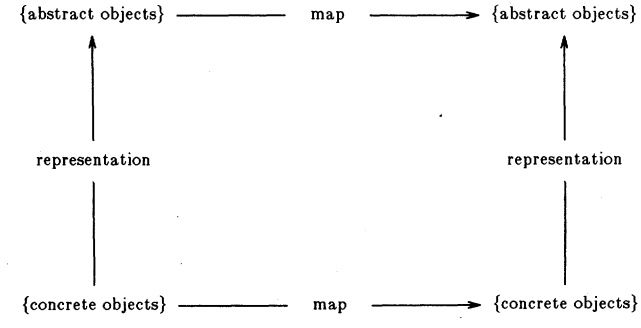
Index Terms—**Abstract data types, functional semantics, modules, programming methodology, program specifications, program verification.**

I. Introduction

MODULES that encapsulate complex data types are perhaps the most important sequential programming-language idea to emerge since the design of ALGOL 60. Such a module serves two purposes. First, in its abstraction role, it allows the programmer to ignore the details of operations (procedural abstraction) and value representations (data abstraction) in favor of a concise description of their meaning. Second, encapsulation is a protection mechanism isolating changes in one module from the rest of a program. The first role helps people to think about what they are doing; the second allows program changes to be reliably made with limited effort.

Modules have their source in practical programming languages beginning with SIMULA [1], and their theory has developed in two directions, based on program proving by Hoare [2], Wulf, London, Shaw [3], and others; and on many-sorted algebras by Guttag [4], Goguen, Thatcher, Wagner, Wright [5], and others. This paper reports on a new proving theory using functional semantics [6].

The essence of data-abstraction is captured by a diagram showing the relationship between a *concrete* world, the objects manipulated directly by a conventional programming language, and an *abstract* world, objects that the programmer chooses to think about instead of the more detailed program objects. Within each world, the items of interest are mappings among the objects. The two worlds are connected by a *representation* function that maps from concrete to abstract.



A data-abstraction theory must define *correctness*, intuitively the property that the programmed concrete maps do properly mirror the abstract maps in our minds. A theory following Hoare's example also defines a *proof method*, a means of establishing the correctness of any particular module.

The abstract world may also be viewed as *specifying* what the program in the concrete world must do. Program specifications are not always functional, however. Sometimes the program behavior should be constrained but not uniquely determined, as when a result should come from a stored set of values, but any of the values is acceptable. The theory presented here is essentially unchanged for specification relations that are not necessarily functions, but it will be presented for simplicity in the functional case.

Section II outlines the formal semantics of modules (and the Appendix gives an example of calculating a program function). Section III presents the module proof theory, with an example in Section IV. In Section V the theory is compared to its primary competitor.

II. Functional Semantics of Modules

A *denotational* or *functional semantics* associates a meaning with certain fragments of a program. Denotational definitions are mathematically precise, but do not always obviously capture the intuitive meaning of programs. In this paper we do not demonstrate that our denotational definitions agree with operational intuition, although that argument can be given [7]. We treat only a subset of Pascal needed for the example of Section IV.

Manuscript received October 31, 1984. The work of J. D. Gannon and R. G. Hamlet was supported in part by the Air Force Office of Scientific Research under Contract F49620-83-K-0018.

J. D. Gannon is with the Department of Computer Science and the Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742.

R. G. Hamlet is with the Department of Computer Science and Engineering, Oregon Graduate Center, Beaverton, OR 97006.

H. D. Mills is with the Department of Computer Science, University of Maryland, College Park, MD 20742 and IBM Corporation, 6600 Rockledge Drive, Bethesda, MD 20817.

IEEE Log Number 8714459.

The most fundamental meaning is the *state*, which associates program identifiers with values. Except for the interval between declaration and assignment of the first value, a state is a function mapping variable names to their current values. There is no clear choice for how to handle "uninitialized variables" (in [6] the state is taken to be an all-inclusive relation), and since this point is far from central to module theory, we will treat states as mappings.

Expressions have as meaning mappings from states to values. The meaning of an integer constant in state S is the (mathematical) integer whose representation in base 10 the constant is (as a string). The meaning of an identifier V in state S is the value that S assigns to V, that is, $S(V)$. On this base of constants and variables the meaning of integer expressions can be defined inductively. If the expression is $X + Y$, then in state S its value is the value of X in state S plus (integer addition) the value of Y in state S. It is convenient to have a notation for meaning functions, and we adopt a convention similar to one used by Kleene: the meaning function corresponding to a programming object is denoted by a box around that object. Using this notation, we have the definitions:

$\boxed{c}$ for integer constant c is the constant function for which c represents the base-10 value.

$\boxed{V}(S) = S(V)$ for identifier V and state S.

$\boxed{X + Y}(S) = \boxed{X}(S) + \boxed{Y}(S)$ for expressions X, Y and state S.

(and similarly for subtraction, multiplication, etc.).

For Boolean expressions it is almost the same. For example,

$\boxed{X >= Y}(S)$ is *true* if $\boxed{X}(S) \geq \boxed{Y}(S)$,
false if $\boxed{X}(S) < \boxed{Y}(S)$,

for expressions X, Y, and state S. The other Boolean operations are treated in the same way.

This inductive definition hides the parsing that must actually be done to assign a meaning function to an expression. In an expression with more than one operation, the operator precedence must be followed in applying the definition. The use of the mathematical operations in these definitions ignores the possibility of overflow. A precise definition could be given for any particular Pascal implementation, but it would complicate our proofs.

Program statements are given meanings of state-to-state mappings. The meaning of assignment $V := E$, where V is a variable and E an expression, is:

$$\boxed{V := E} = \{(S, T): T = S \text{ except that } \boxed{V}(T) = \boxed{E}(S)\}.$$

That is, the input state to the assignment statement and its output state are the same except that the expression value has been attached to the variable.

The meanings of other program constructions are inductively defined; for example

$$\boxed{\text{BEGIN } A; B \text{ END}} = \boxed{A} \circ \boxed{B},$$

for statements A and B, where $\circ$ is functional composition, written in the order the functions are applied. (Again, the parsing necessary to isolate the compound statement and its parts is ignored.)

A more complex definition is

$$\boxed{\text{IF } B \text{ THEN } S} = \{(u, \boxed{S}(u)): \boxed{B}(u)\} \cup \{(u, u): \neg \boxed{B}(u)\}$$

for the conditional statement with Boolean expression B and nested statement S. The two sets cover the cases in which the condition is *true* in the input state (and the statement acts like S), and where it is *false* so the statement means an identity mapping.

Iteration has a less obvious definition:

$$\boxed{\text{WHILE } B \text{ DO } D} = \{(T, U): \exists\, k \geq 0, \text{ such that } \forall\, 0 \leq i < k\ (\boxed{B}(\boxed{D}^i(T)) \wedge \neg \boxed{B}(\boxed{D}^k(T)) \wedge \boxed{D}^k(T) = U)\}.$$

In words, the loop function is undefined for state T unless there is a natural number k (the number of times the loop body is executed) for which the test first fails following k iterations. Then T is transformed to the k-fold composition of D on T. This definition is not constructive, so a characterizing theorem is needed for practical proofs to be carried out. It is:

Theorem (WHILE statement verification): Let W be the program fragment

WHILE B DO D.

Then $f = \boxed{W}$ if and only if:

1) domain(f) = domain($\boxed{W}$),
2) $f(T) = T$ whenever $\neg \boxed{B}(T)$,
3) $f = \boxed{\text{IF } B \text{ THEN } D} \circ f$.

(The proof is given in [6, Chapter 8].)

This theorem implies a proof method for loop W as follows: First, guess or work out a trial function f, say by reading program documentation, or by examining representative symbolic executions of W. (f would be given if W is the code that implements a stepwise refinement of a design.) Then use the three conditions of the if-part of the theorem to check that the trial function is correct. (The Appendix includes an example of using the theorem.)

A comparison between this method and that of Floyd/Hoare is revealing. The function f corresponds to the Floyd/Hoare loop assertion, but unlike an assertion, it must be exact, not merely strong enough to capture necessary properties of the loop. This is both the strength and weakness of the functional method, because exact functions are often easier to find and state accurately than are assertions, yet sometimes the exact function is harder to work with than the weak assertions that suffice when the loop initialization provides a strong precondition.

The definition of statement meaning culminates with the procedure-call statement: the meaning function of a call is the function for the declared body, after textual substitutions (based on the ALGOL 60 copy rule) have been made to accommodate parameters. When there is one VAR parameter X in the declaration of procedure P,

whose declared body is T, the meaning of a call on P passing parameter A is:

$$\boxed{P(A)} = \boxed{T \leftarrow A \backslash X}$$

where $T \leftarrow A \backslash X$ means T with each occurrence of X replaced by A. Students of ALGOL 60 will recognize the semantics of call-by-name; in the absence of arrays this is the same as Pascal's strict call-by-reference. A similar copy-rule substitution can be used to define the meaning of call-by-value parameters. This definition hides a great deal of parsing: to find the meaning of $P(A)$ actually requires locating the definition

```
PROCEDURE P(VAR X: . . .)
```

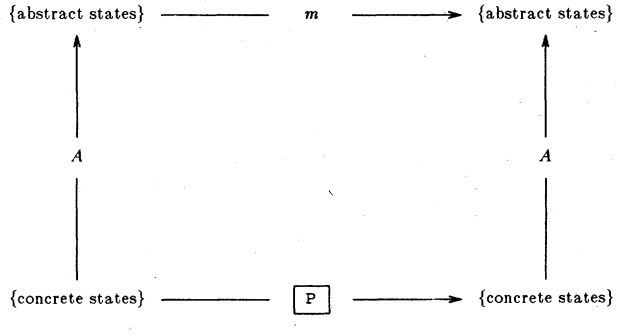
and extracting the declared body.

In practice it is convenient to calculate the meaning of a procedure in terms of its formal parameter, and for each call later substitute the actual parameter identifier. That is, to calculate $\boxed{P(A)} = \boxed{T \leftarrow A \backslash X}$, instead calculate $\boxed{T} \leftarrow A \backslash X$.

The definition assumes there are no conflicts between local and global identifiers; its generalization to multiple parameters is straightforward if there is no aliasing. Each restriction here imposed for simplicity can be lifted in this theory, in contrast to the Floyd/Hoare theory. When there is recursion, the definition leads to a fixed-point equation whose least solution is the defined meaning, and a theorem similar to the WHILE verification theorem is needed for practical proofs. (For details, see [6, Chapters 9, 11].)

The meaning function for a procedure call gives precise form to the concrete portion of the diagram for a data abstraction. The concrete objects are states, and the concrete mapping is the meaning function for a procedure call. The abstract level is more difficult to capture. Its objects and transformations are mental constructions, things a programmer finds convenient to think about. A mathematical theory is seldom available to describe them. There are, however, well defined identifiers and states in the abstract world, formed using aggregate identifiers in place of their component identifiers. The final element in the picture is the correspondence between a typical concrete object and its abstract counterpart, the representation function. This mapping is often many-to-one, because the concrete realization is not unique.

In the data-abstraction diagram:



the abstract mapping is m, the representation mapping is A, and the concrete mapping is the meaning of some procedure P. We say that the diagram *commutes* if and only if beginning in the lower left corner and passing in both possible directions gives the same result whenever the abstract is defined; that is, $A \circ m \subseteq \boxed{P} \circ A$. In the view that the abstract function is a specification, a commuting diagram corresponds to a correct implementation with "don't care" cases: when the abstract function m is undefined, the program function $\boxed{P}$ may take any value.

III. Proof Method

When using a module, a programmer begins with objects that are not of the module's type. These may come from the external world, or may be created internally. They cannot be of the module's type because details of the representation are the module's secret. What the programmer possesses is raw information necessary to construct a value of the module type, and the first call on a module is therefore a conversion call: the calling program passes the component information, and within the module it is placed in the secret internal form. Succeeding invocations of the module make use of the value thus stored, transforming it according to the operations defined within the module. Finally, the transformed value must again be communicated to the world outside the module, converted back to externally usable form. The process is a familiar one: from the very beginning programming languages have had secret representations for integers, reals, characters, etc., and compilers have performed conversions from external to internal forms and back.

For example, in a module implementing complex numbers, the raw data might take the form of two REAL values, one for magnitude and the other for angle. The COMPLEX module's input conversion routine would have a declaration like

```
PROCEDURE InComplex(Mag, Ang: REAL;
    VAR Val: COMPLEX)
```

and a programmer might begin by reading in the pair of REAL values, or by creating them (e.g., for the constant i with:

```
InComplex(1.0, pi/2, Eye)
```

to place the result in the variable Eye of TYPE COMPLEX). Similarly, a routine declared

```
PROCEDURE OutComplex(VAR Mag, Ang: REAL;
    Val: COMPLEX)
```

would be called to obtain answers, while ones like

```
PROCEDURE AddComplex(A, B: COMPLEX;
    VAR Result: COMPLEX)
```

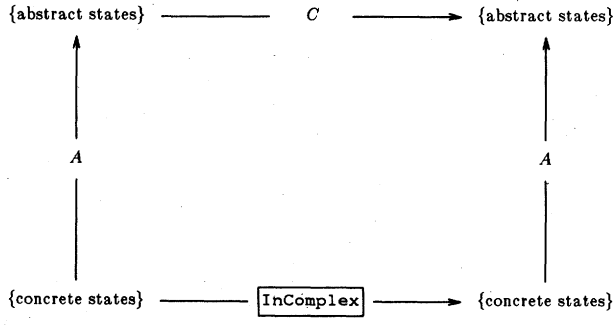
would implement operations of the type. Of course, if the implementor chose the radix form for complex numbers internally, the code for InComplex and OutComplex would be trivial; however, if there is a great deal of addition and not much conversion, an implementation using real and imaginary parts would be better, and in that case these routines make actual conversions. If aggregate types

were first-class objects in Pascal, these procedures could be written as functions to better correspond to the diagrams.

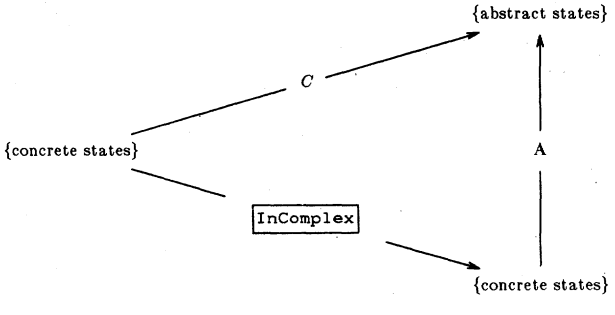
In any application of a module, its users will reason about its actions "in the abstract." That is, they will imagine it performing a mapping involving objects that do not really exist in the computer, those of the intuitive type it implements. For example in COMPLEX, they will think of AddComplex as performing the mathematical operation of complex addition, etc. Here the input- and output-conversion operations have a special role: they are thought of as maps between the built-in language values and the intuitive values of the type being defined. Thus

```
InComplex(1.0, pi/2, Eye)
```

intuitively gives Eye the value $1.0 \times e^{i\pi/2} = i$. The reasoning represented by this equality is an example of "in the abstract:" it in no way depends on the implementation of the module, only on mathematical properties of complex numbers. If the abstract function mapping a pair of REAL values to COMPLEX is C, the diagram for input conversion is:



The raw-data values from which abstract values are constructed all lie in the concrete state, and these values must be preserved by the representation mapping. Thus at the left of the input-conversion diagram the mappings C and InComplex take input values only from the concrete state. The diagram might therefore be collapsed to a triangular one using this common domain, for example:



The programmer has in mind abstract functions for each operation within a module. In reasoning about the program using a module, the programmer will employ these abstract functions. Intuitively, the module implementation is correct if an only if such reasoning is safe. In terms of the operation diagrams, a sequence of operations is thought of on the top: beginning with a triangular diagram whose left side does not involve objects of the module's type T, an object of type T is created by the abstract operation *InT*, then used by abstract operations $m_1 T$, $m_2 T$, $\cdots$ and finally converted back to known values (another triagular diagram) by *OutT*. The abstract view of this sequence of diagrams is that values from outside the module are transformed by the function

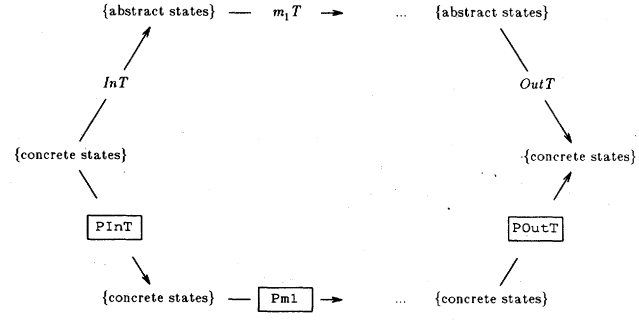
$$InT \circ m_1 T \circ m_2 T \circ \cdots \circ OutT$$

with the intermediate values being the abstract ones of the module's type.

Of course, the actual calculation proceeds across the bottom of the diagrams. The implementation begins with values and successively transforms them, at no time leaving the built-in types of the language. If the procedures for the example functions above are PInT, Pm1, Pm2, ..., POutT, the actual function computed in the sequence is

$$\boxed{\text{PInT}} \circ \boxed{\text{Pm1}} \circ \boxed{\text{Pm2}} \circ \ldots \circ \boxed{\text{POutT}}.$$

We call such a composite an *extended diagram*, here:



Correctness then means that any extended diagram commutes. That is, in the general example above, the implementation is *correct* if and only if

$$InT \circ m_1 T \circ m_2 T \circ \cdots \circ OutT \subseteq \boxed{\text{PInT}} \circ \boxed{\text{Pm1}} \circ \boxed{\text{Pm2}} \circ \cdots \circ \boxed{\text{POutT}}$$

for any sequence of operations. The virtue of this definition is that the representation function does not appear!

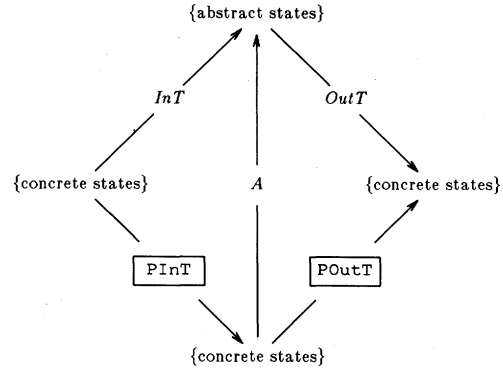
To be useful in software development, however, proofs must apply to operations in isolation, not to sequences of operations. The following theorem allows such proofs to be given.

Theorem: A module's implementation is correct if there is a representation function A such that each operation's diagram commutes using A, and A is identity on built-in types.

Proof: We establish a stronger result than the theorem, namely that every extended diagram commutes, and that the diagram formed by stripping the final output conversion and connecting the newly exposed states with the representation function A also commutes. Proceed by induction on the number of operations between the input- and output-conversion operations.

Base Case: If there are no internal operations, the extended diagram consists of the input-conversion func-

tion immediately followed by the output-conversion function:



We must show that this diagram commutes:

$$InT \circ OutT \subseteq \boxed{\text{PInT}} \circ \boxed{\text{POutT}}.$$

Suppose it were not so, for the concrete point x (for which the abstract maps are defined), i.e.,

$$OutT(InT(x)) \neq \boxed{\text{POutT}}(\boxed{\text{PInT}}(x)).$$

The diagram for the input-conversion function commutes, and a special case is

$$InT(x) = A(\boxed{\text{PInT}}(x)),$$

which substituted on the left side above gives:

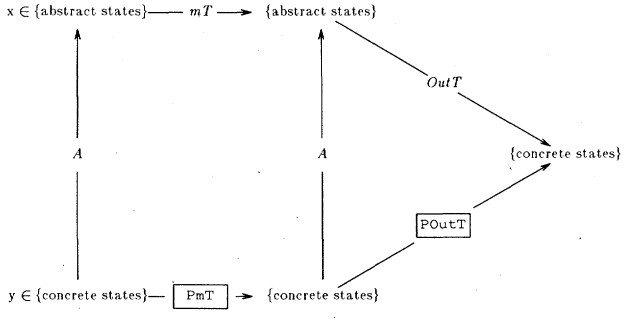
$$OutT(A(\boxed{\text{PInT}}(x))) \neq \boxed{\text{POutT}}(\text{PInT}(x)).$$

That is, there exists a (concrete) $y = \boxed{\text{PInT}}(x)$ such that

$$OutT(A(y)) \neq \boxed{\text{POutT}}(y).$$

But this violates the assumption that the diagram for the output-conversion function commutes. Hence the two diagrams commuting imply that the extended diagram commutes. The additional requirement that the diagram stripped of the output conversion commute, is trivial in this case, since that reduces it to the commuting input diagram.

Induction Step: Suppose then that for all extended diagrams with $n \geq 0$ operations between input and output conversions, the component diagrams commuting implies that the extended diagram commutes, and that the diagram stripped of its output conversion also commutes. Consider a diagram with $n + 1$ operations between conversions, and let the last abstract operation by mT implemented by procedure PmT. Consider only the right end of the extended diagram with the representation mapping inserted:



Let x be any abstract value that enters this fragment from the omitted part of the diagram, that is, beginning with a concrete value and mapping by InT followed by the omitted abstract operations. Similarly let y be the corresponding concrete value that results from $\boxed{\text{PInT}}$ followed by the omitted concrete operations. Then since the omitted part of the diagram commutes by the inductive hypothesis (it is of size n), $x = A(y)$. Because the component diagram for mT commutes, we have

$$mT(A(y)) = A(\boxed{\text{PmT}}(y)).$$

Substituting $x = A(y)$,

$$mT(x) = A(\boxed{\text{PmT}}(y)),$$

which is the statement that the original diagram commutes with its output operation removed. Similarly, that the output operation itself has a commuting diagram gives

$$OutT(A(\boxed{\text{PmT}}(y)) = \boxed{\text{POutT}}(\boxed{\text{PmT}}(y)),$$

and substituting $mT(x)$ from above gives

$$OutT(mT(x)) = \boxed{\text{POutT}}(\boxed{\text{PmT}}(y)),$$

which is the statement that the original extended diagram commutes. Q.E.D.

The verification of a module may therefore be accomplished in isolation by selecting a proper representation function, calculating the meaning of each procedure, and then showing that each operation's diagram commutes for the intended abstract function, calculated meaning, and chosen representation function.

IV. An Example: Rational Numbers

A Pascal TYPE declaration is an implicit form of the representation mapping. For example,

```
TYPE Rational =
    RECORD Num, Den: INTEGER
    END
```

suggests the abstract world of *rational numbers*, where concrete states contain pairs of integer values (N, D), and the corresponding rational value is the fraction with numerator N and denominator D, defined only if N and $D \neq 0$ are defined. The representation mapping A_{rat} from concrete state S to abstract state T is thus

$$A_{rat} = \{(S, T): T = S \text{ except that all identifiers of the form } x.\text{Num} \text{ and } x.\text{Den} \text{ are replaced by } x, \text{ with value } \boxed{x}(T) = \boxed{x.\text{Num}}(S)/\boxed{x.\text{Den}}(S) \text{ if } \boxed{x.\text{Den}}(S) \neq 0\}.$$

This notation is cumbersome, and can be replaced by a *conditional assignment* [6, Chapter 7] in which the state mapping is given by a guarded assignment statement. If the only variable in the abstract state is R:

$$A_{rat} = (\text{R.Den} < > 0 \rightarrow \text{R} := \text{R.Num/R.Den}).$$

The procedure ExpRat given below is intended to raise a rational number R to the power N. The comment de-

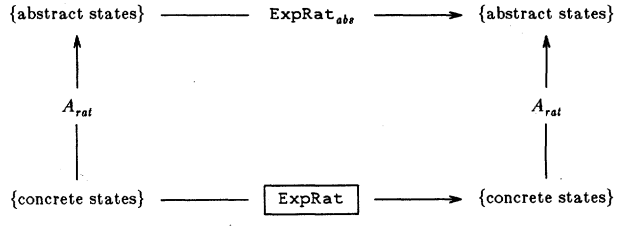
scribes this intention in the abstract ("abs") and concrete ("con") worlds. The comments are written as conditional assignments (the notation uses program operations, so it can be made part of program listings). The vertical bar gives guarded alternatives. For example, the "abs" comment describes the function that would be conventionally expressed as

$$\mathrm{ExpRat}_{abs} = \{(S, T): \boxed{N}(S) \geq 1 \wedge T = S \text{ except that } \boxed{R}(T) = \boxed{R}(S)^{\boxed{N}(S)}\} \cup \{(S, S): \boxed{N}(S) < 1\}.$$

(An empty assignment is written for the identity function.) Similarly, the "con" comment describes ExpRat.

```
PROCEDURE ExpRat(VAR R: Rational;
N: INTEGER);
{abs: (N>=1 → R:= R**N) | (N<1 → )
 con: (N>=1 → R.Num,R.Den := R.Num**N,
          R.Den**N)
      | (N<1 → ) }
VAR
      T: Rational;
      I: INTEGER;
   BEGIN {ExpRat}
      T.Num := R.Num; T.Den := R.Den;
      I := 1;
      WHILE I < N
      DO
        BEGIN
          I := I + 1;
          T.Num := T.Num * R.Num;
          T. Den := T.Den * R.Den
        END;
      R.Num := T.Num; R.Den := T.Den
   END  {ExpRat}
```

To demonstrate the correctness of this procedure, we must calculate ExpRat (see Appendix), and prove that the following diagram commutes:



That is

$$A_{rat} \circ \mathrm{ExpRat}_{abs} \subseteq \boxed{\mathrm{ExpRat}} \circ A_{rat}.$$

The composition of A_{rat} with ExpRat_{abs} is:

```
(R.Den < > 0 → R := R.Num/R.Den) ∘
    ((N >= 1 → R := R**N ) | (N < 1 → )).
```

The *trace table* [6, Chapter 6, 7] is a device for organizing the calculation of program meanings, particularly useful when there are many cases introduced by conditional statements. It is essentially a symbolic execution of the program. Two trace tables, corresponding to the two cases of ExpRat_{abs}, are used to compute the composition:

part	*condition*	R	R.Num	R.Den
A_{rat} ExpRat$_{abs}$	R.Den< >0 N >= 1	R.Num/R.Den (R.Num/R.Den)**N		

part	*condition*	R	R.Num	R.Den
A_{rat} ExpRat$_{abs}$	R.Den< >0 N < 1	R.Num/R.Den		

The resulting function is:

```
(R.Den< >0 AND N>=1 →
      R := (R.Num/R.Den)**N) |
  (R.Den< >0 AND N<1 →
      R:= R.Num/R.Den).
```

The composition of ExpRat with A_{rat} is:

```
((N>=1 → R.Num,R.Den := R.Num**N,R.Den**N) |
  (N<1 → )) ∘ (R.Den < > 0 → R := R.Num/R.Den).
```

Two trace tables are also used to compute this composition. First:

part	*condition*	R	R.Num	R.Den
ExpRat A_{rat}	N>=1 R.Den**N< >0	R.Num**N/R.Den**N	R.Num**N	R.Den**N

Since R.Den**N< >0 implies R.Den< >0, this part of the composition can be rewritten as:

```
N<=1 AND R.Den< >0 →
      R := R.Num**N/R.Den**N.
```

Turning to the second case, we have the following table:

part	*condition*	R	R.Num	R.Den
ExpRat A_{rat}	N<1 R.Den< >0	R.Num/R.Den		

Thus the result of the second function composition is:

```
(N>=1 AND R.Den< >0 →
     R := R.Num**N/R.Den**N) |
  (N<1 AND R.Den< >0 →
      R := R.Num/R.Den)
```

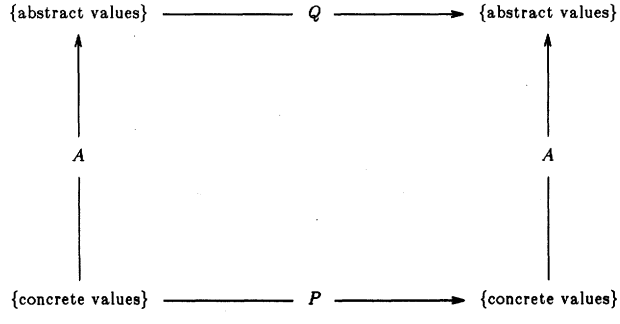
which is identical to the first composition. Hence the diagram commutes, and ExpRat is correct.

V. Comparison to Related Work

Just as the functional method of program proof is close in spirit to the Floyd/Hoare method, so the treatment of modules given here is little more than the application of denotational-semantic definitions to Hoare's initial formalization of SIMULA classes. However, we believe that the choice of the concrete and abstract domains as sets of *states* containing variables connected by the representation mapping is an improvement over the Alphard methodology that is also based on Hoare's work. The states

allow the representation to include not only the value correspondence, but an identifier correspondence as well. When a data abstraction is used, the calls on its operations occur in states that include the abstract variables, and our proof method allows the abstract function whose correctness has been established by the proof of a module to be used directly in such a state.

In the Alphard methodology [3], the commuting diagram is not the central idea. Instead, Hoare-style proofs and invariants are used to factor the correctness problem. The two worlds contain only values, not states that include variable names. There is a program in each world, which manipulates these values. The diagram looks something like:



where A is the representation mapping, Q is a program manipulating abstract values, and P operates on concrete values. These two programs must be proved with respect to input–output assertions in the appropriate world, two Hoare proofs:

$$\beta_{pre}\{Q\}\beta_{post},$$
$$\beta_{in}\{P\}\beta_{out}.$$

Given these proofs, the diagram is forced to commute using a concrete invariant $I_c(x)$, factored from the concrete input and output assertions for procedures β_{in} and β_{out}, and an abstract invariant, $I_a(A(x))$, from the abstract input and output assertions β_{pre} and β_{post}. Sufficient conditions for a commuting diagram are that if the abstract input assertion holds on mapped concrete values, then the concrete input assertion holds on those concrete values:

$$I_c(x) \wedge \beta_{pre}(A(x)) \supset \beta_{in}(x);$$

and, the result of the concrete operation guarantees the result of the abstract operation:

$$I_c(x) \wedge \beta_{pre}(A(x')) \wedge \beta_{out}(x) \supset \beta_{post}(A(x)).$$

Establishing the first condition can present problems since premises about abstract objects are used to establish facts about concrete objects, and the many-to-one nature of representation functions may prevent these demonstrations.

For example, consider the procedure ExpRat proved in Section IV. In Alphard terms, its abstract pre- and postconditions would be

$$\beta_{pre} \equiv R = R' \quad \text{and} \quad \beta_{post} \equiv R = R'^{N},$$

where the ghost variable R′ has been introduced to represent the initial value of the parameter. The concrete input and output assertions are similarly:

$$\beta_{in} \equiv R.Num = R.Num' \wedge R.Den = R.Den'$$
$$\beta_{out} \equiv R.Num = R.Num'^{N} \wedge R.Den = R.Den'^{N}$$

with ghost variables R.Num′ and R.Den′. To show

$$I_c(x) \wedge \beta_{pre}(A(x)) \supset \beta_{in}(x),$$

that is,

$$I_c(x) \wedge R = R' \supset R.Num = R.Num' \wedge R.Den = R.Den',$$

we need to pick a stronger concrete invariant than R.Den $\neq 0$.

$$R.Den \neq 0 \wedge R = R' \supset R.Num = R.Num' \wedge R.Den = R.Den'$$

cannot be proved since the correspondence between abstract and concrete states is not precise enough to pull implications about the latter from facts about the former. We need to add clauses to the concrete invariant that assure that the numerator and denominator do not contain a common factor greater than one and that both positive and negative rationals are both uniquely represented. Thus we might pick

$$R.Den > 0 \wedge gcd(R.Num, R.Den) = 1$$

as the concrete invariant so that

$$R.Den > 0 \wedge gcd(R.Num, R.Den) = 1 \wedge R.Den' > 0 \wedge gcd(R.Num', R.Den') = 1 \wedge R = R' \supset R.Num = R.Num' \wedge R.Den = R.Den'.$$

Much of the difficulty in these proofs comes from including ghost variables in concrete input assertions of operations. We could adopt a proof technique (like that for procedures in Euclid [8]) where such assertions are added to input assertions in the proof rule rather than being part of the input assertions themselves. If this were the case in the example above, the abstract and concrete input assertions could be written as

$$\beta_{pre} \equiv true, \beta_{in} \equiv true,$$

and the verification

$$R.Den \neq 0 \wedge true \supset true$$

could be carried out with a relatively weak concrete invariant. While this approach solves many problems, it may still be necessary to assert that objects have certain

values before an operation is invoked. For example an absolute value operation for rational numbers might be written as follows:

```
R.Num < 0 {R.Num := −R.Num}
    R.Num = |R.Num'|
```

since when the value of N is 1, R.Num and R.Num**N have the same value.

The functions for the sequences of assignment statements were obviously chosen correctly. However, we still must establish the correctness of the function chosen for the WHILE statement:

```
WHILE I < N
DO { (I<N → I,T.Num,T.Den :=
                N,T.Num*R.Num**(N−I),T.Den*R.Den**(N−I)) |
     (I>=N → ) }
  BEGIN
    I := I + 1;
    T.Num := T.Num * R.Num;
    T.Den := T.Den * R.Den
    END;
```

After applying the assignment axiom, we could verify

$$R.Num = R.Num' \wedge R.Num < 0 \supset -R.Num = |R.Num'|$$

However, given the abstract pre- and postconditions

$$\beta_{pre} \equiv R < 0 \quad \text{and} \quad \beta_{post} \equiv R = |R'|$$

we cannot verify

$$I_c(R.Num, R.Den) \wedge R < 0 \supset R.Num < 0$$

without a strong concrete invariant assuring us that the denominator is represented by a positive integer.

In expression-based (functional) programming languages, values do not necessarily require names. But assignment-based (procedural) languages manipulate distinct named values, and their abstract data types require these names.

Appendix

To calculate ExpRat for the procedure of Section IV, we compose the functions computed by the three initial assignment statements, the WHILE statement, and the two final assignment statements:

```
(I,T.Num,T.Den := 1,R.Num,R.Den) ∘
((I<N → I,T.Num,T.Den := N,T.Num*R.Num**(N−I),T.Den*R.Den**(N−I)) |
 (I>=N → )) ∘
(R.Num,R.Den := T.Num,T.Den).
```

The result is:

```
(1<N → I,T.Num,T.Den,R.Num,R.Den :=
                N,R.Num*R.Num**(N−1),R.Den*R.Den**(N−1),
                R.Num*R.Num**(N−1),R.Den*R.Den**(N−1)) |
(1>=N → I,T.Num,T.Den := 1,R.Num,R.Den)
```

Simplifying and ignoring the effects on local variables yields the function:

```
(1<N → R.Num,R.Den := R.Num**N,R.Num**N) | (1>=N → )
```

This is identical to the function given in the "con:" comment of ExpRat:

```
(N>=1 → R.Num,R.Den := R.Num**N,R.Den**N) | (N<1 → )
```

Let the body of the loop be S. Using the WHILE statement verification theorem of Section II, the intended function F, which appears as a comment on the WHILE statement, and [WHILE I < N DO S], are identical if:

1) domain(F) = domain([WHILE I < N DO S]),
2) $F(T) = T$ whenever $\neg$ [I < N](T),
3) F = [IF I < N THEN S] $\circ F$.

The domain of F is:

```
I<N OR I>=N = true
```

If I>=N, the WHILE statement is skipped so termination is assured. If I<N, the WHILE statement is executed, I is incremented, and the eventual termination of the statement is assured because the value of I approaches that of N. Thus the first condition is satisfied.

The second condition requires F to be the identity if the WHILE condition does not hold. This is exactly the final case in the definition of F.

Finally, we can work out the right side of the third condition. The function of the IF statement

```
IF I < N DO S
```

is

$$(I<N \rightarrow I,T.Num,T.Den := I+1,T.Num*R.Num,T.Den*R.Den) \mid (I>=N \rightarrow).$$

The composition IF I < N THEN S $\circ$ F is:

$$((I<N \rightarrow I,T.Num,T.Den := I+1,T.Num*R.Num,T.Den*R.Den) \mid (I>=N \rightarrow)) \circ ((I<N \rightarrow I,T.Num,T.Den := N,T.Num*R.Num**(N-I),T.Den*R.Den**(N-I)) \mid (I>=N \rightarrow)).$$

There are four cases to consider.

Trace Table 1:

Part	*Condition*	I	T.Num	T.Den
IF	I<N	I+1	T.Num*R.Num	T.Den*R.Den
F	I+1<N	N	T.Num*R.Num**(N-(I+1))	T.Den*R.Den**(N-I+1))

Simplifying yields:

$$I<N \text{ AND } I+1<N = I+1<N$$
$$T.Num*R.Num*R.Num**(N-(I+1)) = T.Num**R.Num(N-I)$$
$$T.Den*R.Den*R.Den**(N-(I+1)) = T.Den**R.Den(N-I)$$

Thus this part of the composition is:

$$I+1<N \rightarrow I,T.Num,T.Den := N,T.Num*R.Num**(N-I),T.Den*R.Den**(N-I).$$

Trace Table 2:

Part	*Condition*	I	T.Num	T.Den
IF	I<N	I+1	T.Num*R.Num	T.Den*R.Den
F	I+1>=N			

The condition is:

$$I<N \text{ AND } I+1>=N = I+1=N.$$

When the value of I+1 is the same as the value of N, we observe:

$$T.Num*R.Num**(N-I) = T.Num*R.Num,$$
$$T.Den*R.Den**(N-I) = T.Den*R.Den.$$

Thus this part of the function is:

$$I+1=N \rightarrow I,T.Num,T.Den := N,T.Num*R.Num**(N-I),T.Den*R.Den**(N-I).$$

Trace Table 3:

Part	*Condition*	I	T.Num	T.Den
IF	I>=N			
F	I<N	N	T.Num*R.Num**(N-I)	T.Den*R.Den**(N-I)

The condition I>=N AND I<N cannot be satisfied, so this part contributes nothing to the composition.

Trace Table 4:

Part	*Condition*	I	T.Num	T.Den
IF	I>=N			
F	I>=N			

Thus this part of the function is:

$$(I >= N \rightarrow).$$

Putting the four part functions together:

$$(I+1 <= N \rightarrow I, T.Num, T.Den := N, T.Num * R.Num ** (N-I), T.Den * R.Den ** (N-I)) \mid$$
$$(I >= N \rightarrow).$$

Because the conditions $I+1 <= N$ and $I < N$ have the same value, the composition of the four part functions is identical to F, establishing the third condition of the verification theorem.

Acknowledgment

The authors would like to thank G. Johnson and R. London for helpful discussion of the ideas in this paper. A preliminary version was presented at the 1985 TAPSOFT conference in Berlin, where the comments of J. Wing, J. Horning, and D. Parnas were particularly helpful.

References

[1] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, "The SIMULA 67 common base language," Norwegian Computing Center, Oslo, Publ. S-22, 1970.

[2] C. A. R. Hoare, "Proof of correctness of data representations," *Acta Inform.*, vol. 1, pp. 271–281, 1972.

[3] W. A. Wulf, R. L. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 253–265, 1976.

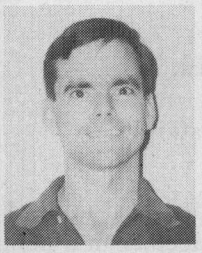
[4] J. Guttag and J. Horning, "The algebraic specification of abstract data types," *Acta Inform.*, vol. 10, pp. 27–52, 1978.

[5] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Initial algebra semantics and continuous algebras," *J. ACM*, vol. 24, pp. 68–95, 1977.

[6] H. D. Mills, V. R. Basili, J. D. Gannon, and R. G. Hamlet, *Principles of Computer Programming: A Mathematical Approach*. Boston, MA: Allyn & Bacon, 1987.

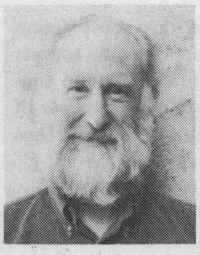
[7] R. G. Hamlet and H. D. Mills, "Functional analysis of programs," Dep. Comput. Sci. Eng., Oregon Graduate Center, Beaverton, Tech. Rep. CS/E 84-006, Oct. 1984.

[8] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek, "Proof rules for the programming language Euclid," *Acta Inform.*, vol. 10, pp. 1–26, 1978.



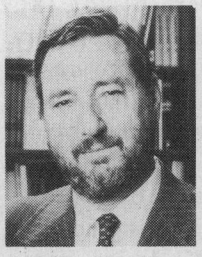
John D. Gannon received the A.B. degree in mathematical economics and the M.S. degree in applied mathematics from Brown University, Providence, RI, in 1970 and 1972, respectively, and the Ph.D. degree in computer science from the University of Toronto, Toronto, Ont., Canada, in 1975.

He is currently an Associate Professor in the Department of Computer Science and the Institute for Advanced Computer Studies at the University of Maryland. His research centers on language and compiler design to increase the reliability of programs. Initially his work focused on the design of less error-prone programming languages. His interests in program proving and testing have lead him to investigate formal specifications, test oracles, and test coverage metrics. He has also studied atomic remote procedure call as a primitive for distributed and fault-tolerant computing.



Richard G. Hamlet (M'81) received the B.S. degree in electrical engineering from the University of Wisconsin in 1959, the M.S. degree in engineering physics from Cornell University, Ithaca, NY, in 1964, and the Ph.D. degree in computer science from the University of Washington, Seattle, in 1971.

He is a Professor in the Department of Computer Science and Engineering at the Oregon Graduate Center, Beaverton. His research interests include theory of software engineering (particularly program testing), tools for software engineering, programming language semantics, computability theory, and text processing by computer. He has worked as an operating-systems programmer and systems programming supervisor for a computer service bureau, and was a faculty member in the University of Maryland Department of Computer Science for 12 years.



Harlan D. Mills (SM'82) received the Ph.D. degree in mathematics from Iowa State University, Ames, in 1952.

He is an IBM Fellow in the Federal Systems Division of the IBM Corporation, Bethesda, MD, and Professor of Computer Science at the University of Maryland, College Park. He has been employed with the IBM Corporation since 1964, where he has served on the Corporate Technical Committee and as FSD Director of Software Engineering and Technology. Before 1964, he was employed in technical staffs at GE and RCA, and as President of Mathematica. He has taught at the University of Maryland since 1975. Before 1975, he taught at Iowa State, Princeton, New York, and Johns Hopkins Universities. He has served on academic visiting committees at the Universities of Toronto and Waterloo and at Catholic and Carnegie-Mellon Universities.

Dr. Mills served as a Governor of the IEEE Computer Society and as a Regent of the DPMA Education Foundation. He was named an Honorary Fellow by Wesleyan University in 1962. He was the recipient of DPMA's 1985 Distinguished Information Sciences Award.