Technical Report CS/E 84-005 *September, 1984*

FUNCTIONAL SEMANTICS OF MODULES

John Gannon
University of Maryland

Dick Hamlet
Oregon Graduate Center

Harlan Mills
University of Maryland

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 the functional method of Mills, and contrasts this with the Alphard formalism based on Hoare logic.

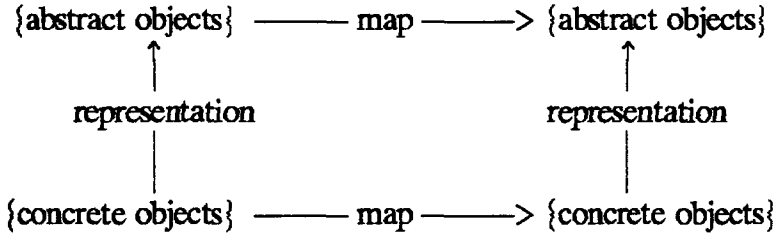
Authors' addresses: Dr. Hamlet, Department of Computer Science, Oregon Graduate Center, Beaverton, OR 97006; Drs. Gannon and Mills, Department of Computer Science, University of Maryland, College Park, MD 20742. Research of Drs. Gannon and Hamlet was partially supported by the Air Force Office of Scientific Research under contract F49620-80-C-0004.

1. 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 the functional semantics of Mills [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 concrete maps programmed 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.

2. Functional Semantics of Modules

A *denotational semantics* associates a mapping with each fragment of a program, as the meaning of that fragment. 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 4.

The most fundamental meaning function is the *state,* mapping program identifiers to their value sets. This function may be undefined when an identifier has no value; the situation can arise for syntactically correct programs only in the execution interval between

declaration and assignment of the first value.

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 its value, that is, S(V). On this base 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

$\boxed{c}$ 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.

$\boxed{X + Y}(S) = \boxed{X}(S) + \boxed{Y}(S)$

(and similarly for subtraction, multiplication, and integer division).

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

$\boxed{X > Y}(S)$ is *true* iff $\boxed{X}(S) > \boxed{Y}(S)$ and *false* iff $\boxed{X}(S) \leq \boxed{Y}(S)$.

Since it is possible for the value functions on identifiers to be undefined, expression functions may inherit this property.

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

$V := E$

is

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

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

$$\llbracket A;\ B \rrbracket = \llbracket A \rrbracket \circ \llbracket B \rrbracket ,$$

where $\circ$ is functional composition, written in the order the functions are applied. (Again, the parsing necessary to isolate the compound statement is ignored.)

A more complex example is

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

for the conditional statement with Boolean expression B and nested statement S.

The loop has a less obvious definition:

$$\llbracket \text{WHILE } B \text{ DO } D \rrbracket = \{(T, U): \exists k \geq 0, \text{ such that } \forall\ 0 \leq i < k$$
$$(\llbracket B \rrbracket (\llbracket D \rrbracket^{1}(T)) \wedge \neg \llbracket B \rrbracket (\llbracket D \rrbracket^{k}(T)) \wedge \llbracket D \rrbracket^{k}(T) = U)\}.$$

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

THEOREM (WHILE statement Verification): Let W be the program fragment

WHILE B DO D.

Then

$$f = \llbracket W \rrbracket$$

if and only if:

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

(The proof is given in [7].) 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. Then use the three conditions of the if-part of the theorem to check that the trial function is correct.

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, it cannot merely be sufficiently strong to capture necessary properties of the loop. This is both

the strength and weakness of the Mills method, because exact functions are sometimes easier to find than assertions, yet sometimes much harder to work with than weak assertions.

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 and identifier conflicts. When there is one VAR parameter X in the declaration of procedure P, whose body is T, the meaning of a call on P passing parameter A is:

$$\lfloor\!\lceil P(A) \rceil\!\rfloor = \lfloor\!\lceil T{:}X{\leftarrow}A \rceil\!\rfloor$$

where $T{:}X{\leftarrow}A$ means that each occurrence of X in T is 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: ...)
```

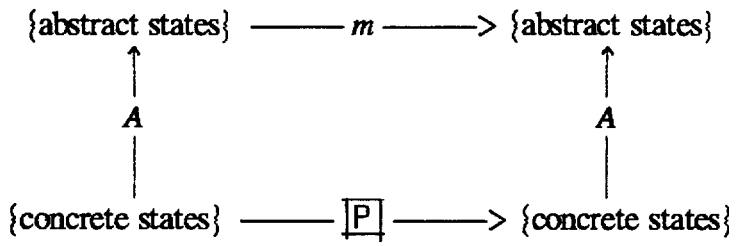
and extracting the declared body.

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 $\lfloor\!\lceil P(A) \rceil\!\rfloor = \lfloor\!\lceil T{:}X{\leftarrow}A \rceil\!\rfloor$, instead calculate $\lfloor\!\lceil T \rceil\!\rfloor :X{\leftarrow}A$.

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 imposed for simplicity can be lifted (and call-by-value parameters handled) in the Mills 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.

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 type 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* iff beginning in the lower left corner and passing in both possible directions always gives the same result, that is $A \circ m = \boxed{P} \circ A$.

3. *Proof Method*

When using a module, a programmer begins with objects that are not of the module's type. These may have come from the external world, or may have been 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. 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). 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

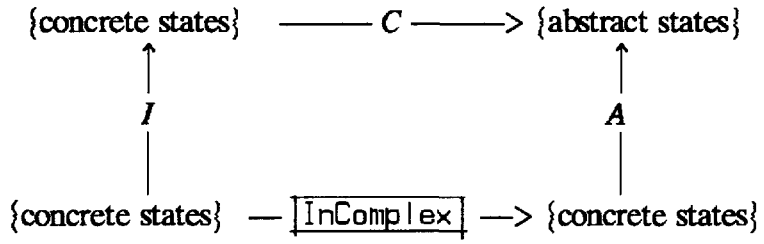
conversions.

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, 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

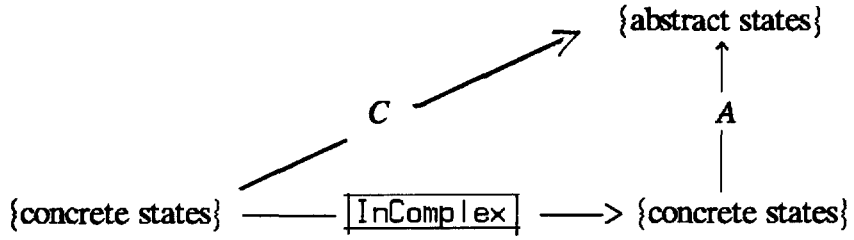
```
InComplex(1.0, pi/2, Eye)
```

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.

The objects whose values are the raw data from which type values can be constructed, exist in the concrete world, which for these objects is also the abstract world. That is, the representation function for such objects is required to be identity. If the abstract function for the input conversion of COMPLEX is *C*, the diagram is



showing identity on the left instead of the representation mapping. Or, the left side could be collapsed to identify the two worlds, producing a triangular diagram. Here for example:



Thus the programmer has in mind abstract functions for each operation of a module. These map between values of the module's type, and other values that may be built in, or defined by other modules. In reasoning about the program using a module, the programmer will employ these abstract functions. Intuitively, the module implementation is correct if and 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_1T, m_2T, ... and finally converted back to known values by (another triangular diagram) OutT. The abstract view of this sequence of diagrams is that non-module values are transformed to other non-module values by the function

$$InT \circ m_1T \circ m_2T \circ \ldots \circ OutT$$

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

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

Correctness then means that any extended diagram, a sequence with triangular diagrams at the extremes, commutes. That is, in the general example above,

$$InT \circ m_1T \circ m_2T \circ \ldots \circ OutT = \boxed{PInT} \circ \boxed{Pm1} \circ \boxed{Pm2} \circ \ldots \circ \boxed{POutT} .$$

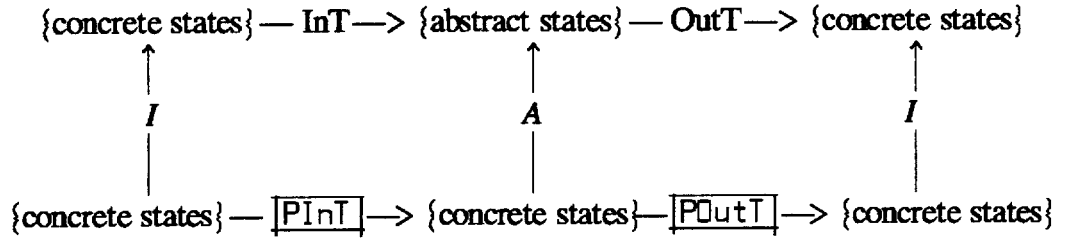
The strange feature of this defining equation is that the representation function does not appear!

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 the identity *I* on built-in types.

Proof. Without loss of generality, assume that the module in question makes no use of other modules. (This must be true of the lowest-level module, and its use by others can be thought of as adding "hidden" operations to them.) The proof is by induction on the number of operations in a sequence between the input- and output-conversion operations.

Base case. If there are none, the extended diagram consists of the input-conversion function immediately followed by the output-conversion function:



In the notation above, we must show that

$$InT \circ OutT = \boxed{PInT} \circ \boxed{POutT} .$$

Suppose it were not so, for the point x, i.e.,

$$OutT(InT(x)) \neq \lfloor POutT \rfloor (\lfloor PInT \rfloor (x)).$$

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

$$InT(x) = A(\lfloor PInT \rfloor (x)),$$

which substituted on the left side above gives:

$$OutT(A(\lfloor PInT \rfloor (x))) \neq \lfloor POutT \rfloor (\lfloor PInT \rfloor (x)).$$

That is, there exists a $y = \lfloor PInT \rfloor (x)$ such that

$$OutT(A(y)) \neq \lfloor POutT \rfloor (y).$$

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, as required.

Induction step. Suppose then that for all diagrams with less than $k > 0$ operations between input and output conversions, the component diagrams commuting implies that the overall diagram commutes. Consider a diagram with k operations between conversions. Reasoning similar to that used in the base case shows that if the extended diagram fails for some point x, then the diagram formed by stripping off its last operation would also fail for x. But that contradicts the inductive hypothesis. QED.

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.

4. 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): $T = S$ except that identifiers of the form x.Num and x.Den are replaced by x, with the corresponding rational value if x.Den $\neq 0$}

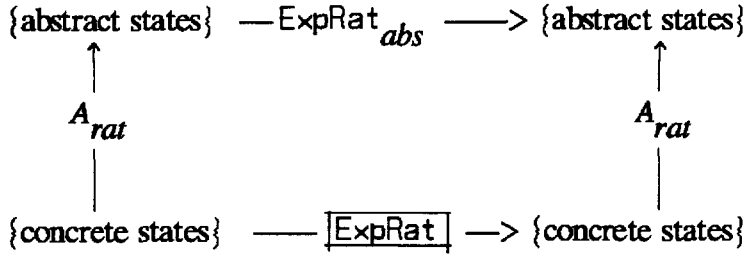
The procedure ExpRat given below is intended to raise a rational number R to the power N. The comment describes this intention in the abstract ("abs") and concrete ("con") worlds. The comment notation combines concurrent assignments with alternative relational guards to describe functions in the syntax of programs. For example, the "abs" part would be more conventionally expressed:

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

Similarly, the "con" comment describes $\boxed{\mathsf{ExpRat}}$.

```
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 $\boxed{\mathsf{ExpRat}}$ (see Appendix), and prove that the following diagram commutes:



That is, on domain(A_{rat}) :

$$A_{rat} \circ \mathsf{ExpRat}_{abs} = \boxed{\mathsf{ExpRat}} \circ A_{rat}.$$

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

(R.Den ≠ 0 → <R> := <R.Num/R.Den>) o
((N ≥ 1 → <R> := <R**N>) | (N < 1 → <> := <>))

The *trace table* [8] 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}	R.Den≠0	R.Num/R.Den		
ExpRat_{abs}	N ≥ 1	(R.Num/R.Den)**N		

part	condition	R	R.Num	R.Den
A_{rat}	R.Den≠0	R.Num/R.Den		
ExpRat_{abs}	N < 1			

The resulting function is:

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

The composition of $\boxed{\mathrm{ExpRat}}$ with A_{rat} is:

((N≥1 → <R.Num, R.Den> := <R.Num**N, R.Den**N>) |
(N<1 → <> := <>)) o (R.Den ≠ 0 → <R> := <R.Num/R.Den>)

Two trace tables are also used to compute this composition:

part	condition	R	R.Num	R.Den
$\boxed{\mathrm{ExpRat}}$	N≥1		R.Num**N	R.Den**N
A_{rat}	R.Den**N≠0	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	N<1			
A_{rat}	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.

5. Comparison with Related Work

Just as the Mills method of program proof is closest in spirit to that of Hoare, so this treatment 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 which 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 things do not work quite so well. For example, consider the procedure ExpRat proved in Section 3. 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′. Proof of a usage requires

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

where C is the concrete invariant and A the representation function. With the invariant R.Den ≠ 0 this is

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

which cannot be proved, since the concrete representation is not unique. Nor can the invariant be strengthened to allow the proof. The trouble is that the correspondence between abstract and concrete state is not precise enough to pull implications about the latter from facts about the former.

References

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

2. C. A. R. Hoare, Proof of correctness of data representations, *Acta Informatica* 1 (1972), pp. 271-281.

3. W. A. Wulf, R. L. London, and M. Shaw, An introduction to the construction and verification of Alphard programs, *IEEE Trans. Software Engineering* SE-2 (1976), pp. 253-265.

4. J. Guttag and J. Horning, The algebraic specification of abstract data types, *Acta Informatica* 10 (1978), 27-52.

5. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, Initial algebra semantics and continuous algebras, *J. of the Assoc. for Comp. Mach.* 24 (1977), pp. 68-95.

6. R. G. Hamlet and H. D. Mills, Functional semantics, University of Maryland Computer Science Technical Report 1238, 1983.

7. *Ibid.*, Functional Analysis of Programs, in preparation.

8. Linger, R.C., Mills, H.D., and Witt, B.I., *Structured Programming: Theory and Practice*, Addison-Wesley, 1979.

Appendix

To determine $\boxed{ExpRat}$, 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>) o
((I<N → <I, T.Num, T.Den> :=
                    <N, T.Num*R.Num**(N-I), T.Den*R.Num**(N-I)>) |
  (I>=N → <> := <>)) o
(<R.Num, R.Den> := <T.Num, T.Den>)
```

The result of the compositions 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 ⌈ExpRat⌉ :

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

since for N=1, R.Num = R.Num**N.

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;
```

Using the WHILE Statement Verification Theorem, 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 ¬ ⌈I < N⌉ (T)
3. F = ⌈IF I < N THEN S⌉ ∘ 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 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 → <I, T.Num, T.Den> := <I+1, T.Num*R.Num, T.Den*R.Den>) |
          (I≥N → ◇ := ◇ )
```

The composition [IF I < N THEN S] o F is:

```
((I◁N → <I, T.Num, T.Den> := <I+1, T.Num*R.Num, T.Den*R.Den>) |
   (I≥N  → ◇ := ◇ )) o
 (I◁N → <I, T.Num, T.Den> :=
                  <N, T.Num*R.Num**(N-I), T.Den*R.Den**(N-I)>) |
   (I≥N → ◇ := ◇)
```

There are four cases to consider.

Execution 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*R.Num **(N-(I+1))	T.Den*R.Den*R.Den **(N-I+1))

Simplifying some of these expressions yields:

```
I◁N 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 → <I, T.Num, T.Den> := <N, T.Num*R.Num**(N-I), T.Den*R.Den**(N-I)>
```

Execution 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:

$$I \lhd N \text{ AND } I+1 \geq N \;=\; I+1=N$$

For I+1=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:

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

Execution 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.

Execution Table 4

Part	Condition	I	T.Num	T.Den
IF	I≥N			
F	I≥N			

Thus this part of the function is:

$$(I \geq N \rightarrow \diamond := \diamond)$$

Putting the four part functions together:

```
(I+1≤N → <I, T.Num, T.Den> :=
                    <N, T.Num*R.Num**(N-I), T.Den*R.Den**(N-I)>  |
  (I≥N → <> := <>)
```

Since I+1≤N = I<N, the composition of the four part functions is identical to F, establishing the third condition.