SOME EXTENSIONS TO ALGEBRAIC SPECIFICATIONS

John V. Guttag
USC Computer Sciences Department

Ellis Horowitz
USC Computer Sciences Department

David R. Musser
USC Information Sciences Institute

Abstract: Algebraic specifications of abstract data types are beginning to gain wide currency. In this paper we discuss an extension to this specification technique which allows the specification of procedures which alter their parameters, and various ways of handling the specification of error conditions.

Key Words and Phrases: abstract data type, data structures, programming languages, procedure specification, error specification, correctness.

CR Categories: 4.22, 4.34, 5.34.

This research was supported in part by the Defense Research Projects Agency under contract No. DAHC15 72 C 0308 and by the National Science Foundation under contract No. MCS76-06089. The views expressed are those of the authors.

1. INTRODUCTION

That abstract data types have a useful role to play in the development of reliable software, is now a widely accepted premise. That a formal technique for specifying abstract data types is an important adjunct to their use, is also widely accepted. No specification technique, however, has emerged as dominant. One leading (at least we like to think of it that way) candidate is the algebraic specification technique espoused in [Guttag75,76a], [Zilles75], [Goguen75] and [Horowitz76].

The theory behind this technique has been explored to some depth by the above authors. Zilles and [Guttag76b] have also devoted considerable attention to studying the application of algebraic specifications to software design and validation. Nevertheless, a great deal of work remains to be done before algebraic specifications can become a truly practical tool for software developers. Little or no attention has been devoted to the problems that occur when one tries to integrate the kind of isolated examples occurring in the literature into real software written in real programming languages. This paper addresses two such "practical problems." The first of these is the development of a notation that, while preserving the virtues of a conventional algebraic specification, allows us to specify abstract data types that include argument altering procedures. The second problem is how best to handle operations that may result in runtime errors.

2. PROCEDURES

The most common example in the literature on algebraic specification is undoubtedly type Stack:

type Stack
interface
    NEWSTACK → Stack,
    PUSH(Stack, Integer) → Stack,
    POP(Stack) → Stack,
    TOP(Stack) → Integer ∪ {UNDEFINED}.
axioms
  declare s:Stack, i:Integer;
    POP(NEWSTACK) = NEWSTACK,
    POP(PUSH(s,i)) = s,
    TOP(NEWSTACK) = UNDEFINED,
    TOP(PUSH(s,i)) = i.

From a pedagogical point of view this example has much to recommend it: it is consistent, complete (by almost any definition), concise, and easy to understand. It has one major drawback--the operations axiomatized are not those most programmers associate with stacks. The operators defined above are purely functional, that is to say they are mappings from a cross-product of values to a value. To preserve the value generated by applying one of the operators to values, one must use an assignment operator defined outside type Stack. A typical program segment might look like

```
declare s:Stack, i:Integer;
s := NEWSTACK;
s := PUSH(s,3);
i := TOP(s);
s := POP(s);
```

Though the discipline inherent in such a restrictive form of inter-module communication often leads to the narrow program interfaces that are so vital in the construction of reliable software, this complete dependence on pure functions and assignment is foreign to the way most people program. There seems to be a perceived need for procedures that directly alter at least some of their parameters. This perception is reflected in the fact that almost all modern programming languages, including those designed with the reliability of programs written in them as an explicit design goal (e.g., Alphard [Wulf76] and Euclid [Lampson76]), permit the definition of procedures that alter their parameters. Thus we find program segments of the form

```
declare s:Stack, i:Integer
s := NEWSTACK
PUSHON(s,3);
i := POPTOP(s);
```

where PUSHON alters its first parameter and POPTOP not only returns a value but also has an effect on its parameter. The existence of parameter altering procedures cannot be ignored in the development of specification techniques. One function of a specification is to define rigorously and formally interfaces among program segments. A representation must, therefore, depict an interface as it will actually occur; not some idealized version of that interface.

We begin by changing the interface specifications so that they distinguish between variable and constant parameters. The interface specification for type Stack will then include

```
PUSHON(var Stack, Integer) →
POPTOP(var Stack) → Integer
```

The first line tells us that PUSHON is a pure procedure (it does not have a value) that alters its first argument but not its second (*const* is the default). The second line tells us that POPTOP is a function procedure that accepts a stack as its only parameter, modifies that parameter, and also has a value of its own. These lines do not, of course, tell us anything about how the parameters are altered or what values POPTOP(s) is to have. Our approach to providing this crucial information is closely related to work on the axiomatic specification of the meaning of procedures in programming languages.

If one assumes no implicit parameters (i.e., global variables), Hoare and Wirth's discussion [Hoare73] of the meaning of the procedure declaration *procedure* p(L):S may be paraphrased:

Let x be the list of explicit parameters declared in L; let $x_1,...,x_m$ be those parameters declared in L as variable parameters. Given the assertion Q{S}R we may deduce the existence of functions F_i satisfying the implication: $Q \supset R$ with each x_i occurring free in R replaced by F_i. The functions F_i may be regarded as those which map the initial values of x on entry to the procedure onto the final values of $x_1,...,x_m$ on completion of the execution of S. What the above says is that the meaning of the procedure p may be expressed in terms of the simultaneous assignment to its *var* parameters of the values obtained by applying the functions F_i to the parameters passed to p. The programs computing these functions can be deduced from S, the body of p.

Taking our cue from this rule, we have extended our specification technique by allowing operators that alter their parameters to be defined in terms of functions that do not. We thus have axioms such as

```
PUSHON(s,i) ≔ s ← PUSH(s,i),
POPTOP(s) ≔ s ← POP(s); TOP(s).
```

where ← is a (simultaneous) assignment operator and the (optional) expression to the right of the semicolon (TOP(s) in the second axiom) is the value returned by the procedure. It is important to note that the expression to the right of the semicolon is evaluated at the same time as the simultaneous assignments. Thus the assignments have no effect on the value returned. As part of a specification that also includes sufficient axioms to define PUSH, POP, and TOP the above axioms are sufficient to fully define PUSHON and POPTOP. A sufficiently complete axiomatization of a type Stack with operations PUSHON and POPTOP follows:

```
type Stack
interface
    NEWSTACK → Stack,
   *PUSH(Stack,Integer) → Stack,
   *POP(Stack) → Stack,
   *TOP(Stack) → Integer ∪ {UNDEFINED},
    PUSHON(var Stack, Integer),
    POPTOP(var Stack) → Integer.
axioms
 declare s:Stack, i:Integer
    POP(NEWSTACK) ≔ NEWSTACK,
    POP(PUSH(s,i)) ≔ s,
    TOP(NEWSTACK) ≔ UNDEFINED,
    TOP(PUSH(s,i,)) ≔ i,
    PUSHON(s,i) ≔ s ← PUSH(s,i),
    POPTOP(s) ≔ s ← POP(s); TOP(s).
```

The asterisk preceding some of the functions in the interface specification indicates that those operators are "hidden" functions which facilitate the definition of the other operators. They are not available to users of the abstract type Stack, nor need they be implemented. Note that these hidden functions, unlike those of [Parnas72], do not involve the introduction of "new" information. That is to say, they do not provide information that is not attainable through the non-hidden operations.

Our approach to proving the correctness of implementations of procedures such as PUSHON and POPTOP derives from the proof rule for procedure declarations. The first step is to derive from the body of the procedure, programs

implementing the functions F_i. From an implementation for PUSHON, for example, we derive one F_i corresponding to PUSH. Once this has been done we convert these programs to recursively defined functions using techniques described in [McCarthy63] and [Manna74]. We now need only show that all of the derived F_i are correct implementations of the axiomatized operations they correspond to. We would, for example, have to show that the derived implementations of PUSH, POP, and TOP combine with the programmer supplied implementation of NEWSTACK to form a model for the first four axioms. How to do this has been discussed at length in [Guttag76b].

3. ERRORS

It is a rare data type that has nothing but "total" operators defined for it. Both the traditionally built-in types (e.g., Integer) and most user-defined types (e.g., Symboltable) have operators that are not well defined over all of the values of the type. Programmers are all too familiar with such unwelcome messages as "attempt to divide by zero" and "symbol table overflow." Throughout our work on abstract data types we have consistently dealt with such "partial" functions by explicitly supplying distinguished values, e.g., UNDEFINED for those instances in which one might normally think of the function as having no value. This allows us to treat all operators as total, an assumption that simplifies the theory underlying some applications. One could cut down the length of specifications by allowing unspecified values to default to some distinguished value, thus insuring completeness. This, however, seems to invite errors of omission. By forcing the authors of a specification to explicitly deal with all syntactically legal uses of the operators we cut down the incidence of these errors.

In practice, we often wish to define types in which errors play a prominent role. Among the most common examples are data types of limited size (as opposed to those we have used in most of our earlier work, which grow without bound). Our experience with types such as these has led us to revise our specifications to include a special mechanism for handling error conditions. The following simple example illustrates some of the inadequacies of the approach to errors taken in some of the earlier work, e.g., [Guttag75], on algebraic specifications.

```
type Bstack
interface
        NEWSTACK(Integer) → Bstack,
        PUSH(Bstack,Integer) → Bstack ∪ {ERROR},
        POP(Bstack) → Bstack ∪ {ERROR},
        TOP(Bstack) → Integer ∪ {UNDEFINED} ∪ {ERROR},
        SIZE(Bstack) → Integer,
        LIMIT(Bstack) → Integer.
axioms
    declare i:Integer,s:Bstack;
        POP(NEWSTACK(i)) = NEWSTACK(i),
        POP(PUSH(s,i)) = IF SIZE(s) < LIMIT(s)
                         THEN s
                         ELSE ERROR,
        TOP(NEWSTACK(i)) = UNDEFINED,
        TOP(PUSH(s,i)) = IF SIZE(s) < LIMIT(s)
                         THEN i
                         ELSE ERROR,
        LIMIT(NEWSTACK(i)) = i,
        LIMIT(PUSH(s,i)) = LIMIT(s),
        SIZE(NEWSTACK(i)) = 0,
        SIZE(PUSH(s,i)) = SIZE(s)+1.
```

Notice that the ranges of some of the operations include the singleton sets {UNDEFINED} or {ERROR}. One could avoid doing this by assuming that these distinguished values are not separate types, but rather are implicitly included in all types, but then one can no longer always assume that the axioms are universally quantified over the types. A more fundamental problem associated with this specification is that it does not accurately parallel most people's concept of a bounded stack. Stack overflow occurs not when we try to push one too many items onto the stack, but rather when we attempt to perform a POP, TOP, or SIZE operation upon a stack onto which too many items have been pushed. An obvious inference to be drawn from the above specification is that the implementations of POP, TOP and SIZE should include a check for stack overflow, but the implementation of PUSH need not. The ludicrousness of this inference need not be explored here.

This problem cannot be cured by the simple expedient of adding the axiom beginning

PUSH(s,i) = IF SIZE(s) > LIMIT(s) THEN ERROR

First, it is not clear that there is any meaningful value we can use in the ELSE clause. We might consider merely repeating the left-hand side in the ELSE clause; this would serve to indicate that PUSH can generate an error. It would not, however, allow us to eliminate the test in axioms 2 and 4, for if we did that it would leave us with an inconsistent axiom set. The introduction of such "circular" axioms would also serve to complicate the processes of automatic verification and the generation of direct implementations.

One solution to this problem is to introduce intermediate functions. Rather than let the user directly invoke functions that may not always be well-defined, we mark these functions as hidden and interpose some sort of access operation between them and the user. The purpose of this access operation is to ensure that any hidden function is invoked only with arguments for which it will be well-defined. The hidden function may then be assumed to be "total" in the sense that it is well-defined for all values to which it can be applied. The following specification of type Bstack, for example, may be construed as sufficiently complete despite the fact that TOP and PUSH are not everywhere defined.

```
type Bstack
interface
        NEWSTACK(Integer) → Bstack,
       *PUSH(Bstack,Integer) → Bstack,
        POP(Bstack) → Bstack,
       *TOP(Bstack) → Integer,
        SIZE(Bstack) → Integer,
        LIMIT(Bstack) → Integer,
        PUSHON(Bstack,Integer) → Bstack ∪ {ERROR},
        TOPOF(Bstack) → Integer ∪ {UNDEFINED}.
```

```
axioms
    declare i:Integer,s:Bstack;
        POP(NEWSTACK(i)) = NEWSTACK(i),
        POP(PUSH(s,i)) = s,
        TOP(PUSH(s,i)) = i,
        LIMIT(NEWSTACK(i)) = i,
        LIMIT(PUSH(s,i)) = LIMIT(s),
        SIZE(NEWSTACK(i)) = 0,
        SIZE(PUSH(s,i)) = SIZE(s)+1,
        PUSHON(s,i) = IF SIZE(s) < LIMIT(s)
                          THEN PUSH(s,i)
                          ELSE ERROR,
        TOPOF(s) = IF SIZE(s) = 0
                          THEN UNDEFINED
                          ELSE TOP(s).
```

This approach seems to present no technical problems. Nevertheless, we do not feel that it is entirely adequate in all situations. While the extra level of nesting in the last two axioms and the introduction of the intermediary operations PUSHON and TOPOF does not seem to have an overly severe effect on the clarity of this specification, this is not always the case. If a large number of operations can cause errors, the specification can become large and unwieldy. In order to understand how the operations behave under normal circumstances one must first wade through a specification of what is to happen in the exceptional cases involving errors. One must try to understand too much at once. This problem has led us to allow for factoring out the exceptional conditions from the main body of the procedure. One way to do this is to associate with each operator a pre-condition defining those values to which it is permissible to apply that operator. We have adopted a related approach in which every specification has, in addition to an interface and a semantic specification, a *restriction specification* that explicitly tells us when the value of an operation will not be well-defined. This leads to the following specification of type Bstack:

```
type  Bstack
interface
        NEWSTACK(Integer) → Bstack,
        PUSH(Bstack,Integer) → Bstack ∪ {ERROR},
        POP(Bstack) → Bstack,
        TOP(Bstack) → Integer ∪ {UNDEFINED},
        SIZE(Bstack) → Integer,
        LIMIT(Bstack) → Integer.
axioms
    declare i:Integer,s:Bstack;
        POP(NEWSTACK(i)) = NEWSTACK(i),
        POP(PUSH(s,i)) = s,
        TOP(PUSH(s,i)) = i,
        LIMIT(NEWSTACK(i)) = i,
        LIMIT(PUSH(s,i)) = LIMIT(s),
        SIZE(NEWSTACK(i)) = 0,
        SIZE(PUSH(s,i)) = 1+SIZE(s),
restrictions
 SIZE(s) ≥ LIMIT(s) ⊃ PUSH(s,i) = ERROR,
 s = NEWSTACK(i) ⊃ TOP(s) = UNDEFINED.
```

Note that the main body of this specification is somewhat simpler than our earlier specification of this type. Note too that the restriction specification is quite short, which reflects the fact that most operations are permissible; a "permissible specification" would be longer. The implications in the restriction specification have been augmented by a value that is to be returned if the predicate is true; this allows the author of a specification to distinguish among the various types of errors that may occur.

If this specification notation were embedded in a programming language, it would be the responsibility of the compiler to establish that all invocations of the operations were legal (cf. the legality assertions of Euclid [Lampson76]). It may be possible to prove, at compile time, that the restriction specification will always be false at the time the operation in question is invoked. If this is not the case, the compiler must generate a runtime check. In either case, a verifier attempting to prove something about an operation can rely upon the restriction condition being false on entry to the operation. That is to say, in proving that an implementation of an abstract data type is consistent with an axiom, one need not consider those cases where the restriction specification doesn't hold.

If the specification is not actually part of the program, the programmer must check the restriction conditions himself. Again, he may be able to construct a proof at compile time; failing this, he will have to program the check himself. Such a check may be synthesised from the operations of the type, or one may define a new primitive operation of the type to make this check. One could, for example, add a new operation to the type, e.g., PUSH-OK, that corresponds to the restriction condition (or its inverse). The interface specification of type Bstack might then be augmented by

PUSH-OK(Bstack) → B in

and the axioms by

PUSH-OK(s) = (SIZE(s) < LIMIT(S)).

4. CONCLUSIONS

This paper discusses two problems encountered while trying to use algebraic specifications of abstract data types in the development of software. The solutions posed to these problems are tentative ones. While we have found them useful in our work, and believe that others will also find them useful, it is not clear that they represent optimal solutions to the problems they were designed to circumvent.

Of the two problems attacked in this paper, the specification of error conditions is the more fundamental one. The introduction of a mechanism for the specification of operations that alter their parameters, is merely a reaction to an unpleasant fact of life. Today, people do program in languages that support such facilities, and people do use them. We are currently in the process of designing a programming language based on data abstraction that will not include procedures that alter their parameters.

We have long felt that the use of pure functions and explicit assignment leads to clearer programs. We have, however, been unable to reconcile this belief with the obvious

inefficiencies involved in the use of multiple function calls (which often involve the duplication of a significant amount of computation) and call by value/result. It was clear that the former problem could be resolved by allowing functions to return tuples and including in our language a facility for assigning members of that tuple to different identifiers, e.g., x,y ← f(x,w,z).

The second problem proved a thornier one. It was obvious that we wanted a language that could be implemented with a call by reference mechanism, but with call by value semantics. That is to say a language in which call by value/result and call by reference are semantically indistinguishable. This is the case if there are no global variables, but this seemed like an intolerably severe restriction. A less severe, and sufficient, restriction is to insure that no function can refer to the object named by a formal parameter except through the formal. The Euclid language, which allows no aliasing whatsoever, has demonstrated that such a restriction is neither prohibitively expensive to enforce nor confining to a programmer. Therefore, we believe that by combining non-aliasing with multiple assignment and effect-free tuple-returning functions as described above, the need for procedures that alter their parameters can be completely eliminated.

ACKNOWLEDGEMENTS

We would like to thank our colleagues at ISI and the paper's referees for their helpful advice.

REFERENCES

[Boyer75] Boyer, R. S., and J S. Moore, "Proving theorems about LISP functions," *J. ACM*, 22, 1, January 1975, 129-144.

[Goguen75] Goguen, J. A., J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Abstract data-types as initial algebras and correctness of data representations, *Proceedings*, Conference on Computer Graphics, Pattern Recognition and Data Structure, May 1975.

[Guttag75] Guttag, J. V., "The specification and application to programming of abstract data types," Ph. D. Thesis, University of Toronto, Department of Computer Science, 1975, available as Computer System Research Report CSRG-59.

[Guttag76a] Guttag, J. V., "Abstract data types and the development of data structures," Supplement to the Proceedings of the SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure, March 1976, pp. 37-46 (to appear *CACM*).

[Guttag76b] Guttag, J. V., E. Horowitz, and D. Musser, "Abstract Data Types and Software Validation," *USC Information Sciences Institute Research Report ISI/RR-76-48*, August 1976 (to appear *CACM*).

[Hoare73] Hoare, C.A.R., and N. Wirth, "An axiomatic definition of the programming language Pascal," *Acta Informatica*, 2, 1973, pp. 335-355.

[Horowitz76] Horowitz, E., and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, June 1976.

[Lampson76] Lampson, B. W., J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek, "Report on the programming language Euclid," 1976.

[Manna74] Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[McCarthy63] McCarthy, J., "Basis for a mathematical theory of computation," in *Computer Programming and Formal Systems*, P. Braffort and D. Hirchberg (eds.), North-Holland Publishing Company, 1963, pp. 33-70.

[Parnas72] Parnas, D. L., "A Technique for Software Module Specifications with Examples," *CACM* 15,5, May 1972, pp. 330-336.

[Wulf76] Wulf, W. A., R. L. London, and M. Shaw, "Abstraction and verification in Alphard: introduction to language and methodology," *Carnegie-Mellon University* and *USC Information Sciences Institute Technical Reports*, 1976.

[Zilles75] Zilles, S. N., "Abstract specifications for data types," IBM Research Laboratory, San Jose, California, 1975.