Data Structure Definition and Access Control
Facilities for Languages Designed for the
Development of Reliable Software,

1. TITLE (and Subtitle)
Data Structure Definition and Access Control
Facilities for Languages Designed for the
Development of Reliable Software,

2. AUTHOR(s)
Billy G. Claybrook

3. REPORT DATE
Feb 87

4. PERIOD COVERED
20 Oct 79 - 19 Oct 80

5. TYPE OF REPORT
Final Report

6. PREPARING ORG. REPORT NUMBER
DAAG29-86-C-0022

7. CONTROLLING OFFICE NAME AND ADDRESS
U.S. Army Research Office
Post Office Box 12211
Research Triangle Park, NC 27709

8. CONTRACT OR GRANT NUMBER(S)
DAAG29-86-C-0022

9. PROGRAM ELEMENT, PROJECT, TASK
AREA & WORK UNIT NUMBERS

10. NUMBER OF PAGES
21

11. MONITORING AGENCY NAME & ADDRESS D (if different from Controlling Office)

12. REPORTING PERIOD COVERED
Feb 87 -

13. SECURITY CLASS. (of this report) Unclassified

14. SECURITY CLASS. (of this report) Unclassified

15. DECLASSIFICATION/DOWNGRADING
SCHEDULE

16. DISTRIBUTION STATEMENT (of the report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
NA

18. SUPPLEMENTARY NOTES
The view, opinions, and/or findings contained in this report are those of the
author(s) and should not be construed as an official Department of the Army
position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
The objective of the research reported here was to develop a specification method
for the specification of abstract data types and an access control facility suitable
for inclusion in high-level programming languages. The research was not
intended to include the design of a complete language but instead involved the
development of programming language features that aid in the development of
languages designed for producing reliable software. A constructive specification
method was developed for specifying abstract data types. Abstract data types
20. ABSTRACT CONTINUED

A constructive specification consists of two parts: a logical structure specification and a semantics of operations specification. The constructive specification method is described in the following pages.
Final Report

U. S. Army Research Office Contract No. DAAG29-80-C-0022

entitled

Data Structure Definition and Access Control Facilities for Languages Designed for the Development of Reliable Software

University of South Carolina

October 20, 1979 – October 19, 1980

Billy G. Claybrook
Principal Investigator
1.0 Introduction

The concept of abstraction has become an important part of problem solving and in particular software development. Through the use of abstract data types, programs can be written that manipulate objects of a given type without being concerned with how the primitive operations on the objects are implemented.

In this paper we describe a constructive (or operational) specification method for specifying abstract data types. We limit our discussion to the specification of data abstractions and do not treat procedural abstractions. A constructive method specifies how a type's operations affect instances of the type. By this definition, the abstract model ([BERZV79]) approach can be considered a constructive specification method while the algebraic method ([GUTT75], [GOGU78]) is nonconstructive.

The work described in this paper includes some important changes in notation to the constructive specification method described by the author in [CLAY79] and [CLAY80]. The improvements in notation permit more concise and more readable specifications to be written. This paper gives a more detailed description of the method, especially with respect to proofs of implementation correctness.

This paper is organized as follows. In the next section we present some preliminary concepts dealing with the algebraic and abstract model specification methods. In Section 3 we describe the constructive specification method by specifying stack, mapping and symboltable data types. Then in Sections 4 and 5, we discuss implementing and proving implementation correctness for abstract data types specified constructively using the symboltable data type. The symboltable data type example was chosen because it is a nontrivial example and because the reader can readily compare the constructive specification given for it in this paper with Guttag's algebraic...
Problem Studied

The proposed research was to develop a specification method for the specification of abstract data types and an access control facility suitable for inclusion in high-level programming languages. The research was not intended to include the design of a complete language but instead involved the development of programming language features that aid in the development of languages designed for producing reliable software.

Results of the Research

To meet the objectives of this research, a constructive specification method was developed for specifying abstract data types. Abstract data types are specified using the module encapsulation mechanism. A constructive specification consists of two parts: a logical structure specification and a semantics of operations specification. The constructive specification method is described in the following pages.
specification (in [GUTJ78]). In Section 6 we treat a database view as a data abstraction and realize it as an abstract data type and specify it using the constructive specification method. This particular example differs from the other data types specified in this paper because the representation data is shared by all instances of the database view. In addition, it illustrates at least one important difference between the constructive method and the algebraic method. Finally, in Section 7 we present a rather detailed comparison of specification methods.

2.0 Preliminaries

An abstract data type can be viewed as a set of values (or objects) and a set of operations applicable to the values. The definition of an abstract data type consists of a specification of the type and an implementation of the type. A specification of a type is a representation-independent description of all properties of the type. This includes specifying the syntax and semantics of a set of primitive operations applicable to the type. The implementation of a type involves assigning a representation for the set of objects defined and then implementing the type's primitive operations on the representation selected. A stack data type, for example, is often represented by a vector and the stack operations, e.g. push, pop, etc., are implemented using the vector and its operations. In turn, the vector type itself has a representation, such as a contiguous block of memory cells, and a set of operations implemented in terms of this representation.

A substantial amount of work has been done to develop specification methods and languages. These approaches to specification have usually been classified as algebraic specifications ([ZILLS75], [GUTJ78], [GOUJ78]) and abstract model specifications ([BERZV79], [WULFW76], [HOARA72]). Each approach has its advantages and disadvantages ([GUTJ78], [BERZV79]). Presently, the most popular method for specifying abstract data types is the algebraic specification method. The reason for this appears to be
twofold: (1) the algebraic method has a mathematical basis in algebra and has been formalized, and (2) the algebraic method has been widely publicized.

When an abstract data type is specified algebraically, it is viewed as an algebra. One then writes the syntax of the operations and the axioms of the corresponding algebra. The syntax specification defines the names, domains and ranges of the type's primitive operations. Writing the axioms is considered to be the semantic specification. The axioms are written in the form of equations (or rewrite rules) which relate the primitive operations of the type to each other. To prove implementation correctness, one must show that an implementation satisfies all the axioms.

With the abstract model approach, an abstract representation (or abstract object) is selected for objects of the type being defined. It is important that the users of a type know what its abstract representation is since the semantics of the type's operations are specified with respect to the abstract representation. For example, the abstract representation of a stack is usually a mathematical sequence. The syntax of a type's operations is defined just as it is in the algebraic approach. The difference between these two methods involves specification of the semantics of operations. The semantics of a type's operations are specified by specifying how each of the operations affects the abstract object (some operations such as membership and retrieval operations do not change the state of the abstract object.) This essentially means that to specify the semantics of a type's operations we specify how the operations affect an instance of the type. To prove implementation correctness, the implementation of each individual operation is proven correct (with respect to the semantics specified for it). It is noteworthy to point out here that an abstract object's operations are used only in specifying the semantics of a type's operations and are not used in implementing the type.

It is important that abstract data types be specified using a formal specification language for at least three reasons:
1) communicating the properties of a type in a precise and unambiguous manner,
2) proving implementation correctness, and
3) designing abstractions.

Communication of properties in a clear and concise manner is important because this permits a type to be understood by both the user of the type and the implementer of the type. Proving implementation correctness, though often nontrivial, should be done when it is an important aspect of software development for an organization using the resulting software. For instance, several governmental agencies believe that trusted software, i.e. verified software, is important for security reasons. Formal specification as an aid to the design of data abstractions may be the most important reason for doing the specification. From our experience in realizing data abstractions ranging from simple abstractions such as stacks to more complex abstractions such as database views, formally specifying an abstraction usually leads to a "better" and often simpler abstraction. With the exception of a few cases such as the one just mentioned, ease of demonstrating consistency of specifications and implementations is often of lesser importance than ease of examining and manipulating specifications.

3.0 The Constructive Specification Method

As stated above, a complete definition of an abstract data type consists of a specification of the type and an implementation of the type. Specification details are visible to a programmer while implementation details are hidden from programmer use. We will use the module encapsulation mechanism described in [CLAY880] to define abstract data types.

The constructive specification of an abstract data type consists of two parts: a logical structure specification and an operation specification. Each of these specifications is discussed in detail below.
3.1 The logical structure specification

When a programmer specifies an abstract data type, he usually has some preconceived notions about the data abstraction such as what an instance "looks like" (independent of any concrete representation), relationship(s), if any, between constituent objects, etc. (an instance of a type consists of a collection of constituent objects). In the constructive approach, these notions are communicated to the user and implementer of a type via a logical structure specification. These notions tend to make understanding a type easier and hence aid in communicating properties of a type. A logical structure specification essentially defines an abstract model of the type being specified. It does this by defining relationship(s), if any, between constituent objects, by defining restrictions to the relationship(s), and by expressing an instance of the type as a tuple of elements. The elements of such a tuple may be sets of constituent objects and relationship(s).

A logical structure specification does not specify an abstract data type; however, it is an important part of the specification of a type because the semantics of operations are specified in terms of how they affect an instance of the type.

The unbounded stack data type in Figure 1 illustrates the various parts of a logical structure specification. The objects section contains the names of the types of constituent objects. Constituent object types can be defined within a logical structure specification (see Figures 2 and 3) or they can be passed as parameter(s) to a defining module (see Figure 1). The relationships section defines any relationship(s) that may exist between constituent objects.
module mapping[
domaintype: Type, rangetype: Type];
Logical structure

objects
type NODE = record NAME: domaintype;
          ATTRIB: rangetype
        end;
occurrence \langle M: \text{collection NODE} \rangle

operations
syntax
NEWMAP: \rightarrow \text{mapping}
defmap: mapping \times \text{domaintype} \times \text{rangetype} \rightarrow \text{mapping}
evmap: mapping \times \text{domaintype} \rightarrow \text{rangetype} \cup \{\text{UNDEFINED}\}
isdefined: mapping \times \text{domaintype} \rightarrow \text{boolean}

semantics
ID: domaintype; \text{ATTRLIST: rangetype}; m: mapping;
NEWMAP = \emptyset;

evmap(m, ID) = \begin{cases}
\text{true} & \exists x \in m \: x.\text{NAME} = ID \\
\text{false} & \text{otherwise}
\end{cases}
isdefined(m, ID) = \begin{cases}
\text{true} & \exists x \in m \: x.\text{NAME} = ID \\
\text{false} & \text{otherwise}
\end{cases}
defmap(m, ID, \text{ATTRLIST}) = m \uplus \{ x \text{ with } [\text{NAME} = ID, \text{ATTRIB} = \text{ATTRLIST}] \};

end mapping;

Figure 2. Constructive specification of mapping data type
module stack[elementtype: Type];

logical structure

object ELEM: elementtype
relationships ONTOPOF: ELEM to ELEM
occurrence \langle S: \text{collection} ELEM, O: \text{ONTOPOF} \rangle

invariant assertions

1. ONTOPOF is linear

operations

\text{syntax}

\text{NEWSTACK:} \rightarrow \text{stack}
\text{push:} \text{stack} \times \text{elementtype} \rightarrow \text{stack}
\text{pop:} \text{stack} \rightarrow \text{stack}
\text{top:} \text{stack} \rightarrow \text{elementtype} \cup \{\text{UNDEFINED}\}
\text{replace:} \text{stack} \times \text{elementtype} \rightarrow \text{stack} \cup \{\text{ERROR}\}

\text{semantics}

x, y, z: \text{elementtype}; s: \text{stack};
\text{NEWSTACK} = \langle \emptyset, \emptyset \rangle

\text{push}(s, x) = s \text{ with } [S = S \cup \{x\}, 0 = \text{if } s = \text{NEWSTACK} \text{ then } \emptyset \text{ else } 0 \cup \{x, \text{top}(s)\}]

\text{pop}(s) = \text{if } S = \emptyset \text{ then } \text{NEWSTACK}
\text{else } s \text{ with } [S = S - \{x\}, 0 = 0 - \{x, y\}
\langle x, y \rangle \cup 0 ] \text{ where } x = \text{top}(s)

\text{top}(s) = \text{if } s = \text{NEWSTACK} \text{ then } \text{UNDEFINED} \text{ else } x
\text{where } x \in S \text{ and } (\emptyset \neq \{y \in S\} \langle y, x \rangle \cup 0)

\text{replace}(s, x) = \text{if } s = \text{NEWSTACK} \text{ then } \text{ERROR}
\text{else } s \text{ with } [S = (S - \{y\}) \cup \{x\},
0 = (0 - \langle y, z \rangle) \cup \{x, z\}] \text{ where } y = \text{top}(s)

end stack;

Figure 1. Constructive specification of unbounded stack type
module symboltable[domain: Type, range: Type];

logical structure

objects

  type NAME = record NAME: domain; ATTRIB: range end;
  type BLOCK = record CONTENTS: collection NODE; end;

relationships

  occurrence: BLOCK to BLOCK
  B: collection BLOCK, ND: NESTED

invariant assertions

  1. NESTED is linear
  2. \( \exists b \in B : b \Rightarrow b' \in B \)

operations

syntax

  INIT: \( \rightarrow \) symboltable
  *currentblock: symboltable \( \rightarrow \) BLOCK
  enterblock: symboltable \( \rightarrow \) symboltable
  leaveblock: symboltable \( \rightarrow \) symboltable
  isinblock: symboltable \( \times \) domain \( \rightarrow \) boolean
  retrieve: symboltable \( \times \) domain \( \rightarrow \) range
  addid: symboltable \( \times \) domain \( \times \) range \( \rightarrow \) symboltable

semantics

  b, B, cb, x, ID: BLOCK; x: NODE; ID: domain; s: symboltable;

  INIT = s with \( N = \emptyset, B = \{ b \mid b \text{ with } b.C contours = \emptyset \} \)

  currentblock(s) = b where \( \exists b \in B : b \Rightarrow b' \in B \)

  enterblock(s) = s with \( b.C contours = \emptyset \) where \( b \notin B \)

  leaveblock(s) = if ND = \emptyset then INIT
                  else s with \( N = N \setminus \{ cb.C contours \} \)
                         B = B \setminus \{ cb \}
                         ND = ND \setminus \{ \langle cb, b \rangle \}

  isinblock(s, ID) = if \( \exists x \in cb.C contours \Rightarrow x.NAME = ID \)
                      then TRUE
                      else FALSE;

  retrieve(s, ID) = if s = INIT then UNDEFINED
                     else if \( \exists x \in cb.C contours \Rightarrow x.NAME = ID \)
                          then x.ATTRIB
                          else retrieve(leaveblock(s), ID);

  addid(s, ID, ATTRLIST) = s with \( N = N \cup \{ x \mid x.NAME = ID, x.ATTRIB = ATTRLIST \} \)
                           B = B with \( cb.C contours = cb.C contours \cup \{ x \} \);

end symboltable;

Figure 3. Constructive specification of symboltable data type
Possibly the most important ingredient of a logical structure specification is the invariant assertion. The primary functions of the invariant assertion are to specify restrictions to relationships and to specify assertions about the model on which the semantics of operations are specified. When the semantics of operations are specified, care must be taken to ensure that the operations do not "violate" or "ignore" any of the assertions.

The abstract data types specified in this paper do not reflect the true importance of the invariant assertions. Invariant assertions are particularly valuable in specifying abstract data types where there are several relationships between constituent objects and there are important restrictions to these relationships. For example, in specifying database views there are usually several relationships between entity types. As an example, a specification of the presidential database described in [TAYLOR76] required 26 invariant assertions. In this context, a logical structure specification is analogous to a database subschema definition. However, there is an important difference between a logical structure specification and say a CODASYL subschema definition. The invariant assertions of a logical structure specification define the semantics of the relationships between entity types. A CODASYL subschema definition can only define the syntax of the relationships and not the semantics.

Wyckoff ([WYCKM80]) has suggested the use of diagrams to pictorially describe data types specified using the constructive approach. These diagrams can be used to aid in the specification of a data type or by a user and an implementer to help them understand the type. A diagram can be drawn directly from a logical structure specification. Diagrams for the stack, mapping and symboltable data types are shown in Figure 4. As we will see, the diagrams in Figure 4 aid us in developing and understanding the correspondence function.
given in Figure 6. In these diagrams $a \mathbin{R} b$, i.e. object $a$ is related to object $b$ via relation $R$, is represented as

![Diagram](image)

To take a closer look at logical structure specifications, we examine the unbounded stack data type and the symboltable data type in Figures 1 and 3, respectively. The stack type is a parameterized type with the type of elements in a stack instance passed as a parameter. An instance of the stack type consists of a collection of objects of type `elementtype`. The relationship between the elements of the stack is ONTOPOF; ONTOPOF is linear ([CLAYB79] provides a list of terms, such as linear, is ordered on, etc., that expedite writing invariant assertions.) The terms used in this paper should be self explanatory.
Constituent objects of a symbol table consist of instances of BLOCK and NODE. These object types are defined within module symboltable; however, the types of the component elements of NODE are passed as parameters. The relationships and occurrence sections need no explanation; however, the invariant assertions section does require some explanation. Assertion 2 states that there is a block which is assumed to be global to all blocks in a program. This same assumption is made in Guttag's specification of the symboltable data type ([GUTTJ78]) but it is not explicitly stated.
3.2 Specification of operations

The operations section consists of two parts: a syntax specification and a semantics specification. The syntax section defines the names, domains and ranges of a type's primitive operations. Hidden (or auxiliary) operations, i.e. operations that are used in specifying other operations but are not available for programmer use, are indicated by placing an asterisk ('*') to the left of their name. Operation **currentblock** in Figure 3 is an example of a hidden operation.

In many cases, specifying the semantics of a set of operations using the constructive approach is straightforward. With respect to the symboltable data type in Figure 3, the **INIT** operation creates a symbol table and establishes the outermost scope. Operation **enterblock** establishes a new block nested in the current block and **leaveblock** removes the current innermost block. Operation **isinblock** tests whether or not an identifier has been declared in the current block and **retrieve** retrieves the attribute list of an identifier from the block closest to the innermost block. Operation **addid** is explained by the following example. The symbol table for the program shown in Figure 5 at the point of compilation indicated by the arrow is given by

```
addid(addid(enterblock(addid(init, x, real)), x, complex), y, complex).
```

```
begin
  x: real;
  
end

begin
  x, y: complex;
  
end

Figure 5. Program segment
```
4.0 Implementation of Abstract Data Types

An implementation of an abstract data type using the constructive
specification method consists of a representation specification followed by
an implementation of the type's operations with respect to the
representation. The representation specification of an abstract data type
indicates the concrete object(s) used to represent the type and the
correspondence between an abstract object and its concrete object(s).
Correspondence is defined as a function from concrete object(s) to an abstract
object. The correspondence function is a homomorphism and it is identical in
functionality to the abstraction function of Moore [11, 12]. A
correspondence function is named and it is used in proofs of implementation
correctness.

A correspondence function may be relatively simple (i.e. it is for the
symbolic data type and its representation), or it may be quite complex. In
general, the more dissimilar the concrete objects and the abstract values they
represent, the more complex the correspondence function.

The correspondence function for the implementation of the symbolic
data type in Figure 6 is SYMT. The concrete objects in this case are
instances of the stack and mapping data types specified in Figures 1 and 3,
respectively. SYMT maps these concrete objects into a symbolic table. The
occurrence tuple shown in Figure 6 represents an occurrence of symbolic
where \( N = \text{UM}, B = S \) and \( ND = 0 \). \( M \) is defined in Figure 2 and \( S \) and \( O \) are
defined in Figure 1. This definition of SYMT can be easily understood by
looking at Figure 4. Implicit in the definition of SYMT is the fact that a
block is equivalent to an element of a stack and each element of a stack is a
collection of mappings.

The implementation section provides an implementation of an abstract
data type's operations in terms of concrete object(s) and concrete object
operations. An implementation of an operation is specified using composition of operations, tests for equality (or inequality) and the if-then-else construct. An implementation of the symboltable data type is given in Figure 6.

**representation**

```plaintext
symtab: symboltable; stk: stack; m: mapping;
symtab = SYMT(stk,m) where SYMT(stk,m) = <UM, S, 0>
```

**implementation**

```plaintext
stk: stack; id: domaintype; attr: rangetype;
INIT = SYMT(push(NEWSTACK, NEWMAP))
enterblock(SYMT(stk)) = SYMT(push(stk, NEWMAP))
addid(SYMT(stk), id, attr) = 
leavblock(SYMT(stk)) = if 0 = ∅ then SYMT(push(NEWSTACK, NEWMAP)) else SYMT(pop(stk))
retrievel SYMT(stk), id = if stk = push(NEWSTACK, NEWMAP) then undefined
else if isdefined(top(stk), id) then evmap(top(stk), id) else retrieve(leaveblock(SYMT(stk)), id)
isinblock(SYMT(stk), id) = isdefined(top(stk), id)
currentblock, SYMT(stk) = top(stk)
```

Figure 6. A representation and an implementation of symboltable data type

5.0 Proving Implementation Correctness

For the constructive specification method, a proof of implementation correctness involves showing that the implementation of each individual operation is correct with respect to a correspondence function. That is, for the symboltable data type we must show the following:
for each symboltable operation \( \sigma \) show that
\[
\sigma(SYMT) = SYMT(\sigma'),
\]
where \( \sigma' \) is an implementation of \( \sigma \).

A pictorial view of SYMT is given in Figure 7 using the implementation of
symboltable operation addid. This pictorial description suggests that proving
implementation correctness involves showing that the instance resulting from
the left side of each operation implementation (given in Figure 6) is the
same as the instance resulting from the corresponding right side.

![Figure 7. Pictorial meaning of addid(SYMT) = SYMT(replace)](image)

Before a proof of implementation correctness can be done for the
symboltable data type, it is necessary to prove an implementation invariant.
In general, an implementation invariant is a property that is true for all
values of a type produced by an implementation of the type. An implementation
invariant comes up as one proceeds through a proof of correctness. They make
themselves obvious when the rewrite process during a proof can no longer
continue. In general, one may not know all the implementation invariants when
a proof is initiated.

For the symboltable, the following implementation invariant, posed as a
theorem, must be proven:

for each symboltable = SYMT(stk), stk / NEWSTACK.

To prove the theorem, we must show that the invariant is true for all
symboltable operations that produce symboltable values. Since INIT,
enterblock, addid, and leaveblock are the symboltable operations that produce
Proof

right side

\[ \text{SYMT}(\text{push}(\text{stk}, \text{NEWMAP})) \]

\[ = \text{SYMT}(\text{stk with } [S = S \cup \{ \text{NEWMAP} \}, \ 0 = \text{if } \text{stk} = \text{NEWSTACK} \text{ then } \emptyset \text{ else } \emptyset \{ \text{NEWMAP}, \ \text{top(} \text{stk} \}) \}] \]

(by semantics of push)

\[ = \text{SYMT}(\text{stk with } [S = S \cup \{ \text{NEWMAP} \}, \ 0 = \emptyset \{ \text{NEWMAP}, \ \text{top(} \text{stk} \}) \}] \]

(by implementation invariant)

\[ = \text{symtab with } [B = B \cup \{ \text{with CONTENTS = } \{ \text{NEWMAP} \} \}] \]

(by correspondence function SYMT in Figure 6)

left side

\[ \text{enterblock}(\text{SYMT}(\text{stk})) = \text{enterblock}(\text{symtab}) \]

(by definition of SYMT)

\[ = \text{symtab with } [B = B \cup \{ \text{with CONTENTS = } \{ \text{NEWMAP} \} \} \]

\[ \text{ND} = \text{NDU}\{ \text{with CONTENTS = } \{ \text{NEWMAP} \} \} \]

(by semantics of enterblock)

The left and right sides are equal since \( \text{NEWMAP} \neq \emptyset \).

Figure 9. Proof of implementation correctness for operation enterblock.

5.1 Comments on proofs of implementation correctness

Proofs of implementation correctness for types specified using the constructive approach are relatively straightforward. For the constructive specification of the symboltable data type the degree of difficulty for a proof of implementation correctness is about the same as for Guttag's algebraic specification (see [GUTT78]). The proof of implementation correctness for the symboltable implementation given in Figure 6 is relatively simple since the representation and instances of the type are very similar in nature. When the representation of a type and instances of the type are dissimilar, in general, the proofs become more difficult. For example, we specified an ordered linear list (using both the algebraic specification
symbols that values, it suffices to show that given a stack, stk, for which the
invariant is true, each of these operations produces a new stack for which the
invariant is still true. Thus, we will have shown inductively that the
invariant is true for all values of the symbolable type. Wyckoff
[114, 125] gives a complete proof of the invariant; we repeat only the proof
invariant, operation addid (see Figure 8).

\[
\text{\texttt{defmap(top(stk), id, attr})}
\]

(by implementation of operation addid)

\[
\text{\texttt{defmap(top(stk), id, attr})}
\]

(by semantics of \texttt{replace})

\[
\text{\texttt{defmap(top(stk), id, attr})}
\]

Therefore the invariant is true for operation addid.

Figure 8. Proof of the implementation invariant using operation addid.

A complete proof of implementation correctness for the implementation
given in Figure 6 is given by Wyckoff. To illustrate our proof procedure, we
repeat only a proof of implementation correctness for operation \texttt{enterblock}
(see Figure 9). The other proofs are similar. In Figure 9 we show

\[
\text{\texttt{enterblock(SYMT(stk))}} = \text{\texttt{SYMT(push(stk, NEWMAP})}}
\]

by showing that the instances resulting from the left and right sides are
identical.

The development of the right side in Figure 9 makes use of the semantics
of stack operation \texttt{push}, the implementation invariant defined above, the
function \texttt{SYMT}, and the fact that \texttt{cb = top(stk)}. The development of the left
side is simpler and makes use of the semantics of operation \texttt{enterblock} and
\texttt{SYMT}. 

```
method and the constructive method) and then implemented it using a binary search tree. In our attempts to complete a proof of implementation correctness for the list, we had to consider several special cases of the binary search tree for both specifications. For the algebraic specification, considerable ingenuity was required as well as proving some auxiliary theorems to permit rewriting to continue during proofs (see [CLAY8861]). Like the proofs for the algebraic specification of the linear list, the proofs for the constructive specification were relatively long; however, they were straightforward in comparison.

6.0 A Database View Example

In this section we realize a database view as an abstract data type and specify it using the constructive specification method. This example is included because database views are substantially different than the other data abstractions specified in this paper. The difference exists because databases are normally shared data objects with many different users capable of updating the database and the updates may affect the values of a view.

A database view is an abstraction of an underlying database. A view has the following properties:

1) an underlying database is assumed to exist; otherwise, it is impossible to derive a view,
2) views are not materialized, i.e. they are not stored in a database, and
3) in a shared database environment, some values of a view may be created by other users who are updating the underlying database.

An underlying database is considered to be the representation of a database view and an implementation of a database view is an implementation of the view's operations with respect to the underlying database.
In the example described below, the database view specified is a relational view ([LDATE(77)]). For simplicity, we assume that the view consists of a single relation (referred to as a view relation) and the representation is a single relation (referred to as a base relation) stored in the database. The base relation consists of a set of tuples with the following attributes

(car_no, model, body_no, yr, current_value, mi, disp, dest, rc, col, loc)

while an instance of the view relation consists of a set of tuples with the following attributes

(car_no, mi, dest, disp, rc)

Each car owned by the car rental company (described below) has a tuple corresponding to it stored in the database with car_no being a primary key.

The environment for this example is as follows. A car rental company in a large city has a central headquarters where car purchases are made, allocations of cars to local rental offices are made, etc. When a new car is purchased, all information on the car such as license plate number (car_no), body identification number, color, model, etc. are stored in the database by headquarters database system personnel. When a car is assigned to a local rental office, the value of the location attribute of the particular car's tuple is set to the location of the local office to which it is assigned and its disposition (disp) is set to 'avail', meaning that it is available to be rented.

The function of a local rental office differs from that of the headquarters. A rental office does not need access to all of the information in a car tuple. For this reason, a local office's database consists of a set of the view relation tuples described above. A local office can apply the operations rent_car, return_car, maint_car, return_maint, and avail_car to its database represented by its view. Operations rent_car and return_car are used when a car is rented and returned, respectively. Operations maint_car and return_maint are used when a car is sent to maintenance and returned from
maintenance, respectively. Operation avail_car is used to obtain a set of available cars having a specified rate class. The rental car database view described here is a simplified version of a view described in [CLAYB81a]. In the description of the view in [CLAYB81a], the view also has rental car history and maintenance history view relations.

The database view is realized as the abstract data type dbview and specified using the constructive specification method (see Figure 10). In this specification we assume that the data types car_number, mileage, etc. are available for use in module dbview. Type re_set is defined as a set of car numbers. Type dbview is a parameterized type, with parameter xloc (the value of xloc is a local rental office location). A particular instance of dbview is a set of tuples consisting of only those cars assigned to a particular local rental office. For this reason the local office location does not have to be specified as a parameter in each of dbview's operations.
module dbview(xloc: location);

logical structure
objects
  type TUPLE = record
    ca: car_number;
    mi: mileage;
    dest: destination;
    disp: disposition;
    rc: rate_class
  end;

occurrence <collection TUPLE>

operations
  syntax
  rent_car: dbview x car_number x destination → dbview U {UNDEFINED}
  return_car: dbview x car_number x mileage → dbview U {ERROR}
  maint_car: dbview x car_number → dbview U {ERROR}
  return_maint: dbview x Car_number → dbview U {ERROR}
  avail_car: dbview x rate_class → rc_set

semantics
  c: car_number; d: destination; m: mileage;
  r: rate_class; t: TUPLE; db: dbview;
  rent car(db, c, d) =
    if ∃t ∈ db (t.ca = c and t.disp = 'avail')
      then db with [t.disp = 'rented', t.dest = d]
      else UNDEFINED;

  return car(db, c, m) =
    if ∃t ∈ db (t.ca = c and t.disp = 'rented')
      then db with [t.disp = 'avail', t.mi = m]
      else ERROR;

  maint car(db, c) =
    if ∃t ∈ db (t.ca = c and t.disp = 'avail')
      then db with [t.disp = 'maint']
      else ERROR;

  return maint(db, c) =
    if ∃t ∈ db (t.ca = c and t.disp = 'maint')
      then db with [t.disp = 'avail']
      else ERROR;

  avail car(db, r) = \{t.ca | t ∈ db and t.rc = r and t.disp = 'avail'\};

end dbview;

Figure 10. Constructive specification of a relational database view

A representation of dbview is given in Figure 11. In this representation specification, the correspondence function VIEWMAP is defined as a derivation of view relation tuples from base relation tuples. This is a natural way to define correspondence functions for relational database views and has been adopted in RIGEL ([ROWEL79]) and in EXT_Pascal ([CLAYB81a]).
representation
do: dbview;
db = VIEWMAP(R) where
VIEWMAP(R) = \{<r.ca, r.mi, r.dest, r.disp, r.rc> | r \in R \text{ and } r.loc = xloc}\}

Figure 11. A representation for view dbview

The function VIEWMAP defines an instance of dbview as consisting of a set of tuples having the five attributes: ca, mi, dest, disp, and rc. R is the representation of instances of dbview and it is the base relation from which instances of dbview are derived.

An implementation of dbview's operations is omitted here but they would be implemented in terms of R's operations, namely append, delete and replace. The interested reader should see [CLAYB81b] for an implementation of dbview and a proof of implementation correctness for some of dbview's operations.

6.1 Comments on the database view example

The database view example was included in this paper for two reasons; (1) to show the utility of the constructive specification method, and (2) to serve as an example for comparison with the algebraic specification method in an environment where sharing of representation data occurs.

In the database view example, values of an instance of dbview may be created by the headquarters database personnel. For example, a car tuple may be removed from a view if the location of the car is modified, i.e. the car is assigned elsewhere or sold, by headquarters personnel, or the headquarters may decide to alter the rate_class of some or all of its cars.

Specifying data abstractions in which representation data is shared causes no particular problems for the constructive specification; however, it does cause some problems for the algebraic method. The reason for this is that with the algebraic method, the behavior of an object and the generation of that object are intertwined. The problem incurred by the algebraic method is that some of the constructor operations that create values of dbview are
not in doview’s operation set but instead are included in the underlying
database’s operation set. With the algebraic specification, all values of an
abstract data type must be produced by some sequence of constructors. To
specify doview algebraically, we have to include the auxiliary (or hidden)
operations emptyview and add car. Claybrook ([CLAY64b]) provides a
complete specification of doview using Guttag’s algebraic specification
method. The specification requires 12 axioms.

7.0 Comparison of Specification Methods

Below we summarize properties and characteristics of constructive and
algebraic specification methods. Each method appears to have some inherent
problems that are difficult to handle. It may seem that we are overly
criticizing the algebraic specification method. However, one of the problems
in comparing these two methods is that the algebraic method has been examined
and researched by more people than the constructive method and thus more is
known about its strengths and weaknesses.

In the algebraic approach, a data type is specified by giving a set of
axioms relating the type’s primitive operations. For the constructive
specification, the logical structure of a type is specified, thus defining an
abstract model of the type, and the operations are specified with respect to
this model. With the abstract model approach, an abstract object such as a
sequence or set is selected and the type’s primitive operations are specified
with respect to the abstract object. The constructive specification method
described in this paper is very similar to the abstract model approach;
therefore, any comments made about one applies to the other.

Algebraic specifications are usually considered to be more elegant than
specifications developed using a constructive method. If we do not consider
hidden operations as unnecessary detail then the algebraic method does not
introduce unnecessary detail. Since it has a mathematical basis in algebra,
it is well suited to formal analysis and has been used as the basis for
"semi-automatic" verification systems such as AFFIRM ([MUSSD79]). Another feature of the algebraic specification method is that it is compact; however, as we shall see below, algebraic specifications are not always compact.

Several authors ([FLONL79], [BERZV79], [MAJSM79]) have pointed out some serious problems with algebraic specifications. These problems are for the most part inherent in the basic methodology used to develop the method and, thus, are difficult to remedy. Fortunately, most of these problems are not problems for the constructive methods. Majster ([MAJSM79]) states that one of the problems with algebraic specifications is the necessity to introduce hidden or auxiliary operations for some specifications. These operations are not harmful from a theoretical point of view but they do cause two problems. The number of auxiliary operations can become quite large and the number of axioms may increase sharply (especially if some of the auxiliary operations must be constructors). Majster performed a case study for the description of a file with nine (9) operations and 10 auxiliary operations and ended up with over 50 axioms. One of the claimed benefits of algebraic specifications, compactness, was lost in this specification. Another problem is that the auxiliary operations have to be implemented.

Another major problem with algebraic specifications involves producing a well-formed specification, i.e. producing a complete and consistent set of axioms. There does not seem to be any straightforward and intuitive mathematical procedure for checking completeness and consistency. Inconsistencies occur when axioms result in two objects being equivalent when in fact they should have been different. A complete axiom set is one to which an independent axiom cannot be added. A more thorough discussion of completeness and consistency can be found in [GUTTJ80].

Operations in algebraic specifications are all functions. They do not permit side effects and they can return only a single type of value. In addition, there are problems with partial operations and error equations ([MAJSM79]). The problem with errors occurs because an error is not of the
value type produced by an operation yet it may be the value produced by an
operation. This can lead to contradictions. Goguen ([GOGUJ78]) and Guttag
([GUTTJ77]) have suggested solutions to these problems.

Verifying implementation correctness using a constructive specification
method is conceptually easier than verifying implementation correctness using
algebraic specifications. Flon ([FLONL79]) describes why this is the case.
Algebraic methods deal with values rather than objects. Values are immutable;
that is, an operation can change one value into another but it cannot change
the state of a particular value. This means that operations cannot have side
effects although in a typical procedural implementation operations have side
effects. For instance, in an implementation of pop and push, these stack
operations will not result in new stacks but rather will alter existing
stacks. Constructive specification methods do not suffer from this problem
since operations specified constructively deal with changing the state of
objects.

A proof of implementation correctness must be done for each operation in
a constructive specification whereas for algebraic specifications we must show
that an implementation satisfies each axiom. For some data types, the number
of axioms may be much larger that the number of operations. From our
experience, there is nothing to suggest showing that an axiom is satisfied by
an implementation is any easier than showing that an implementation of an
operation is correct with respect to a homorphism. This suggests that the
effort required to prove implementation correctness may be substantially
larger for the algebraic approach than for the constructive approach. If one
of the operations is changed then all of the axioms referring to the
corresponding operation will have to be reverified.

In general, we have found that designing abstractions using constructive
methods is much easier than the algebraic specification method. Minor changes
in the behavior of an operation are easier to describe for constructive
specifications. A modification in the definition of one constructively
defined operation does not normally affect the other operations. In an
algebraic specification the meanings of the operations are defined in terms of
the relations between them, so that a change in an operation or change in an
axiom can affect other operations and axioms. It is difficult to produce a
well-formed algebraic specification for a new data abstraction especially if
the exact behavior required is not yet completely designed.

A major complaint against constructive specification methods is that
they are not minimal, i.e. they introduce unnecessary detail. Proponents of
the constructive methods argue that only those properties that are necessary
and relevant to specifying semantics are specified. Auxiliary operations
required by the algebraic method to specify some data abstractions such as the
data base view in Figure 10 can be considered to be unnecessary detail.

Another complaint against constructive specifications is that they
suggest implementations and/or constrain the concrete objects. Berzins
([BERZV79]) states that the issues of time and space efficiency often
requires that the representation used in an implementation differ
significantly from the model used in the specification of semantics. One
problem inherent in a constructive specification method is that specifying the
semantics of complex operations may be quite lengthy. Constructive
specifications sometimes lack the succinctness found in algebraic
specifications.

In general, we have found constructive specifications easier for the
user of a type to interpret. They permit an implementation to be more easily
developed.

From our discussion above, it appears that constructive and algebraic
approaches are ideally suited for some applications and poorly suited for
others. Unfortunately, little attention has been devoted to the problems that
occur when one tries to integrate the kind of examples occurring in the
literature into real software written in real programming languages. The
examples used in published reports have been well known mathematical objects.
These objects are familiar to most, thus it is difficult to demonstrate the value of specifying the semantics of operations, especially for communication purposes or in designing data abstractions. The database view example in this paper and in [CLAYB81a], [CLAYB81b] is an attempt to illustrate the importance of specifications in a practical situation. What has been missing in the literature are examples that provide user oriented operations on non-mathematical objects.

References


DATEG77 Date, C. J. An Introduction to Database Systems, Addison-Wesley, 1977.


Publications


"Abstractly Identical Terms of An Abstract Data Type", Sperry Research Center Report RP-81-9 (submitted to TOPLAS).


Personnel

Billy G. Claybrook, Principal Investigator (10 months)

Anne-Marie Discepolo, co-investigator (3 months)

James C. Cleaveland, consultant (1 month)
DATE
ILMED
-8