10 Algebraic Specification

Objectives

- To explain the role of formal specifications in the definition of interfaces between sub-systems.
- To introduce the algebraic approach to formal specification where abstract data types or object classes are specified by defining the properties of their associated operations.
- To describe a systematic way to write an algebraic specification.
- To illustrate a number of incremental ways to construct algebraic specifications from simpler specifications.

Contents

Large systems are usually decomposed into sub-systems which are developed independently. Sub-systems obviously make use of other sub-systems so an essential part of the specification process is to define sub-system interfaces. Once the interfaces are agreed and defined, the sub-systems can be developed independently.

Sub-system interfaces are often defined as a set of abstract data types or objects (Figure 10.1). Each sub-system implements these interfaces and all sub-system access is through the interfaces. It is therefore essential that the sub-system interface is clearly and unambiguously specified. This reduces the chances of misunderstandings between the sub-system providing a facility and the sub-system using that facility.

The starting point for the specification is an informal interface specification, expressed as a set of abstract data types or object classes, that has been negotiated by the sub-system designers. The algebraic approach is particularly suitable for the definition of sub-system interfaces. This method of formal specification defines an object class or abstract data type in terms of the relationships between the type operations.

Guttag [Guttag, 1977 #162] first discussed this approach in the specification of abstract data types. Cohen *et al.* [Cohen, 1986 #17] show how the technique can be extended to complete system specification using an example of a document retrieval system. Liskov and Guttag [Liskov, 1986 #497] also cover the algebraic specification of abstract data types. Several languages for algebraic specification have been developed including OBJ [Futatsugi, 1985 #155] and Larch [Guttag, 1985 #163].

It is sometimes difficult to prove that algebraic specifications are mathematically complete and consistent. Van Vliet [van Vliet, 1993 #498] discusses some of the reasons for this in a way which is understandable by non-mathematicians. These problems do not, however, detract from their usefulness in supporting the critical process of interface specification. However, an incomplete formal specification may be more precise than informal interface definitions. Theoretical limitations of the algebraic approach do not necessarily detract from its practical utility.

As I discussed in Chapter 9, there are good reasons for developing a formal specification even if no mathematical manipulation of the specification is carried out. Developing a formal specification forces an analysis of an informal interface description. It therefore may reveal potential inconsistencies and ambiguities in that description. The formal specification supplements the informal description. It reduces communication problems between the developers and the users of the sub-system interface.

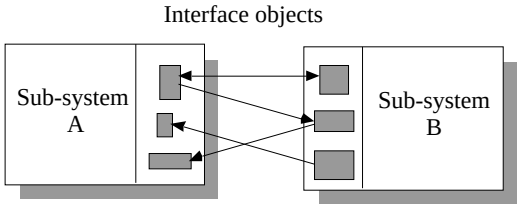

Figure 10.1 Sub-system interface objects

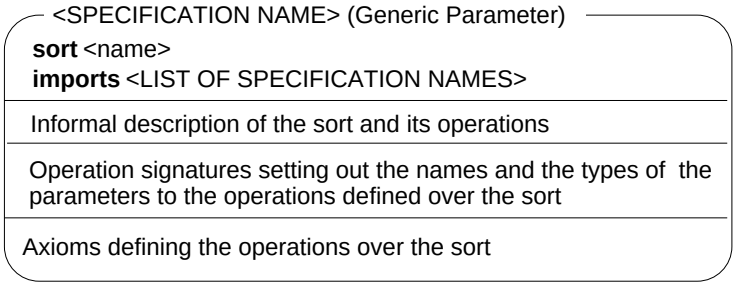

Figure 10.2 The format of an algebraic specification

In the notation which I use here, each specification has a name and an optional generic parameter list. By allowing generic parameters, abstract types which are collections of other types (arrays, lists, etc.) may be specified without concern for the types in the collection. The generic type may be instantiated to create a more specific specification. For example, an array with a generic parameter could be instantiated to an array of integers, an array of strings, an array of arrays and so on. As part of this generic type specification, operations which must be defined over the generic type may be specified.

Figure 10.2 illustrates how algebraic specifications are presented in this chapter. The body of the specification has four components.

1. An introduction that declares the sort (the type name) of the entity being specified. A sort is the name of a set of objects. It is usually implemented as a type. The introduction may also include an imports declaration where the names of specifications (not the sort names) defining other sorts are declared. Importing a specification makes these sorts available for use.

2. A description part where the operations are described informally. This makes the formal specification easier to understand. The formal specification complements this description by providing an unambiguous syntax and semantics for the type operations.

3. The signature part defines the syntax of the interface to the object class or abstract data type. The names of the operations that are defined, the number and sorts of their parameters and the sort of operation results is described in the signature.

4. The axioms part defines the semantics of the operations by defining a set of axioms which characterise the behaviour of the abstract data type. These axioms relate the operations used to construct entities of the defined sort with operations used to inspect its values.

To illustrate the parts of an abstract data type specification, consider the specification of an array (Figure 10.3). This is a generic abstract data type provided in almost all programming languages. It is an *abstract* data type (although it is not defined as such in Ada, C or C++) because it has a restricted set of allowed operations. It is a *generic* type because the elements of an array can usually be of

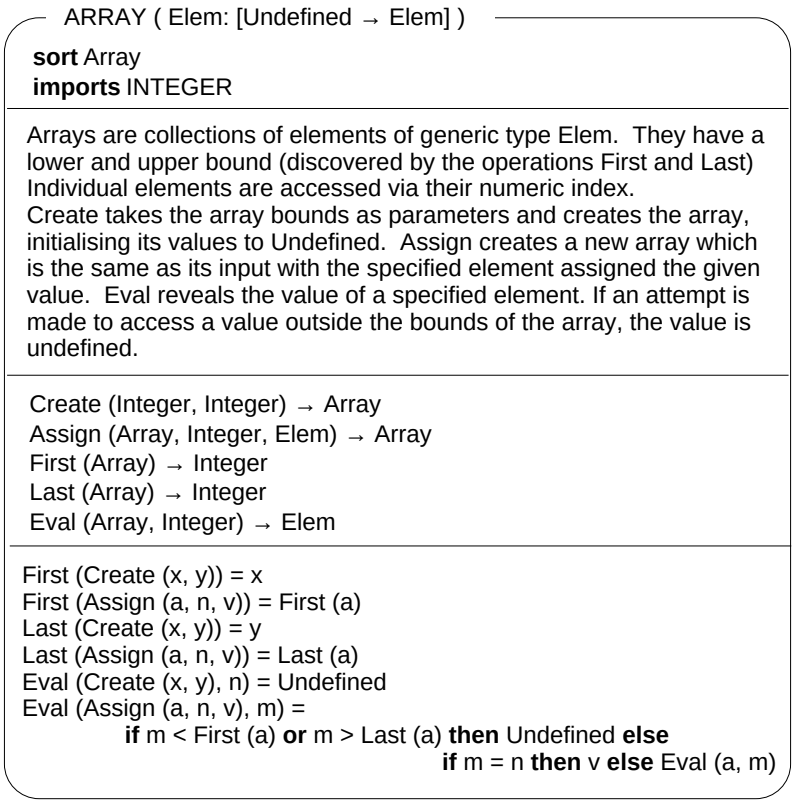

Figure 10.3 The specification of sort Array

any other type. The defined operations create the array, discover the lower and upper bounds, find the value of an array element and assign a value to an array element.

I use the notation suggested by Cohen *et al.* [Cohen, 1986 #17] to introduce generic parameters. The name of the generic parameter is Elem meaning any element type. Any operations which must be defined for the instantiated parameter must also be specified with the generic parameter. In the array specification, this means that the sort which instantiates Elem must have an operation called Undefined associated with it. Undefined is a special value whose type is Elem. It indicates that the evaluation of some operation has resulted in an error. For example, the result of the operation is Undefined when an attempt is made to access an element which has not been defined a value.

10.1 Systematic algebraic specification

In this section, I describe a systematic approach that may be used to define an algebraic specification of an abstract data type or object class. This is illustrated by specifying an abstract data type representing a very simple linked list. There are six

stages in this approach. These are not necessarily carried out in sequence. As the specification is developed, the specifier refines the results of earlier stages in the process.

The stages in developing an algebraic specification are:

1. *Specification structuring* The informal interface specification must be structured into a set of abstract data types or object classes. Operations should be proposed for each interface entity.

2. *Specification naming* Establish a name for the specification, decide whether or not it requires generic parameters and decide on a name for the sorts identified.

3. *Operation selection* Choose a set of operations on these entities based on the identified interface functionality. This should include operations to create instances of the sort, to modify the value of instances and to inspect the instance values. You may have to add functions to those initially identified in the informal interface definition.

4. *Informal operation specification* Write an informal specification of each operation. This should describe how the operations affect the defined sort.

5. *Syntax definition* Define the syntax of the operations and the parameters to each operation. This represents the signature part of the formal specification. Update the informal specification at this stage if necessary.

6. *Axiom definition* Define the semantics of the operations. This is described in Section 10.1.1.

To develop the example of the list specification, assume that the first stage, namely specification structuring has been carried out and that the need for a list has been identified. The name of the specification and the name of the sort can be the same although it is useful to distinguish between these by using some convention. I use upper case for the specification name (LIST) and lower-case with an initial capital for the sort name (List). As lists are collections of other types, the specification has a generic parameter (Elem).

In general, for each abstract type, the required operations must include an operation to bring instances of the type into existence (Create) and to construct the type from its elements (Cons). In the case of lists, we need an operation to evaluate the first list element (Head), an operation which returns the list created by removing the first element (Tail) and an operation to count the number of list elements (Length). We shall see in Section 10.2 how to add more operations to this set.

To define the syntax of each of these operations, you must decide which parameters are required for the operation and the results of the operation. In general, input parameters are either the sort being defined (List) or the generic sort. Results of operations may be either of those sorts or some other sort such as integer or Boolean. In the list example, the Length operation returns an integer. An imports declaration declaring that the specification of integer is used should therefore be included in the specification. Finally, the semantics are defined as a set of equations as explained in Section 10.1.1. The final specification is shown in Figure 10.4.

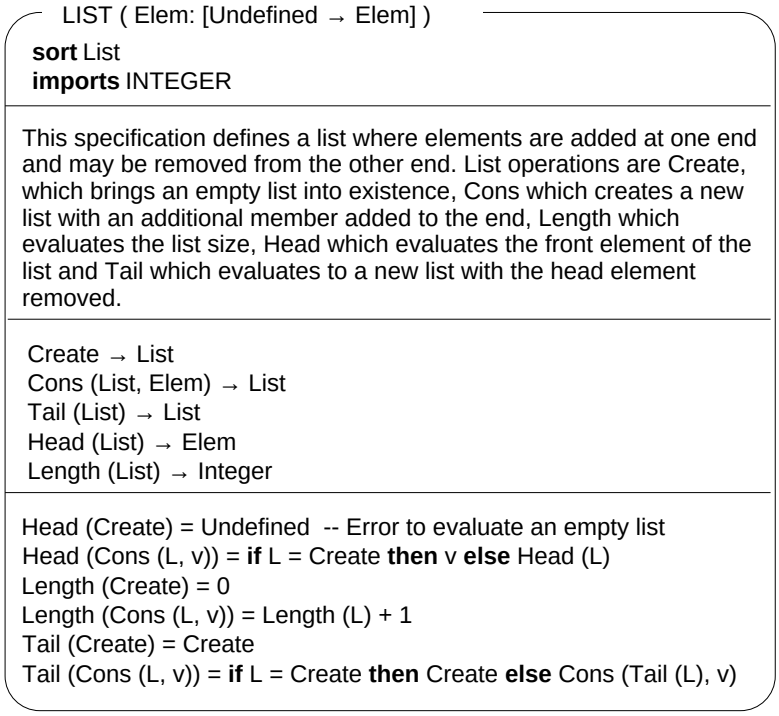

Figure 10.4 The specification of sort List

10.1.1 Defining axioms

The axioms which define the semantics of an abstract data type are written using the operations defined in the signature part. They specify the semantics by setting out what is always true about the behaviour of entities with that abstract type.

Operations on an abstract data type usually fall into two classes.

1. *Constructor operations* which create or modify entities of the sort defined in the specification. Typically, these are given names such as Create, Update, Add, etc.
2. *Inspection operations* which evaluate attributes of the sort defined in the specification. Typically, these are given names which correspond to attribute names or names such as Eval, Get, etc.

A good rule of thumb for writing an algebraic specification is to establish the constructor operations and write down an axiom for each inspection operation over each constructor. This suggests that if there are m constructor operations and n inspection operations there should be m*n axioms defined.

However, the constructor operations associated with an abstract type may not be primitive constructors. Primitive constructors are operations which can't be expressed using other constructors. If a constructor operation can be defined using

other constructors, it is only necessary to define the inspection operations using the primitive constructors.

An example of this is given in the specification a list shown in Figure 10.4. The constructor operations are Create, Cons and Tail which build lists. The access operations are Head and Length which are used to discover list attributes. The Tail operation is not a primitive constructor as it can be defined using Cons and Create. There is therefore no need to define axioms over the Tail operation for Head and Length operations. These would include redundant information that could be derived from other axioms.

Evaluating the head of an empty list results in an undefined value. The specifications of Head and Tail show that Head evaluates the front of the list and Tail evaluates to the input list with its head removed. The specification of Head states that the head of a list created using Cons is either the value added to the list (if the initial list is empty) or is the same as the head of the initial list parameter to Cons. Adding an element to a list does not affect its head unless the list is empty.

The value of the Tail operation is the list which is formed by taking the input list and removing its head. The definition of Tail shows how recursion is used in constructing algebraic specifications. The operation is defined on empty lists then recursively on non-empty lists with the recursion terminating when the empty list results. This is a very common technique to use when writing algebraic specifications.

It is sometimes easier to understand recursive specifications by developing a short example. Say we have a list [5, 7] where 5 is the front of the list and 7 the end of the list. The operation Cons ([5, 7], 9) should return a list [5, 7, 9] and a Tail operation applied to this should return the list [7, 9]. The sequence of equations which results from substituting the parameters in the above specification with these values is:

```
Tail ([5, 7, 9]) =
 Tail (Cons ( [5, 7], 9)) =
  Cons (Tail ([5, 7]), 9) =
   Cons (Tail (Cons ([5], 7)), 9) =
    Cons (Cons (Tail ([5]), 7), 9) =
     Cons (Cons (Tail (Cons ([], 5)), 7), 9) =
      Cons (Cons ([Create], 7), 9) =
       Cons ([7], 9) =
          [7, 9]
```

The systematic rewriting of the axiom for Tail illustrates that it does indeed produce the anticipated result. The axiom for Head can be verified using a similar approach.

10.1.2 Primitive constructor operations

When developing an algebraic specification, it is sometimes necessary to introduce additional constructor operations in addition to those identified as part of the interface specification. The interface constructors are then defined in terms of these more primitive operations.

These additional primitive constructors may be required because it is difficult or impossible to define the inspection functions in terms of the interface functions. We can see an example of this in a binary tree specification with the identified interface functions as shown in Figure 10.5.

It is impossible to specify the inspection operations (Left, Data, Right, Is_empty, Contains) in terms of the Add function. An extra function (Build) is therefore added to the specification to simplify their definition. There is no easy or automatic way to identify these functions. If you find it very difficult to specify inspection functions in terms of the identified constructors, this may mean that you have to think about the problem and try to identify a more primitive constructor operation.

The specification for Binary_tree with the Add constructor defined in terms of the Build constructor and other functions is shown in Figure 10.6.

The notation .=. (Elem, Elem) means that the equality operator '=' is an infix operator with operands of type Elem. The precise notion of equality depends on the sort of the entities to which the operator is applied. It must therefore be defined for each abstract type which may be used to instantiate Elem.

Operation	Description
Create	Creates an empty tree.
Add (Binary_tree, Elem)	Adds a node to the binary tree using the usual ordering principles i.e. if it is less than the current node it is entered in the left subtree; if it is greater than or equal to the current node, it is entered in the right sub-tree.
Left (Binary_tree)	Returns the left sub-tree of the top of the tree.
Data (Binary_tree)	Returns the value of the data element at the top of the tree.
Right (Binary_tree)	Returns the right sub-tree of the top of the tree.
Is_empty (Binary_tree)	Returns true if the tree does not contain any elements.
Contains (Binary_tree, Elem)	Returns true if the tree contains the given element.

Figure 10.5 Operations on a binary tree

BINTREE (Elem: [Undefined → Elem, .=. → Bool, .<. → Bool])

sort Binary_tree
imports BOOLEAN

Defines a binary tree where the data is of generic type Elem.
See Figure 10.5 for interface operation description.
Build is an additional primitive constructor operation which is introduced to simplify the specification. It builds a tree given the value of a node and the left and right sub-trees.

Create → Binary_tree
Add (Binary_tree, Elem) → Binary_tree
Left (Binary_tree) → Binary_tree
Data (Binary_tree) → Elem
Right (Binary_tree) → Binary_tree
Is_empty → Boolean
Contains (Binary_tree, Elem) → Boolean
Build (Binary_tree, Elem, Binary_tree) → Binary_tree

Add (Create, E) = Build (Create, E, Create)
Add (B, E) = **if** E < Data (B) **then** Add (Left (B), E)
else Add (Right (B), E)
Left (Create) = Create
Right (Create) = Create
Data (Create) = Undefined
Left (Build (L, D, R)) = L
Right (Build (L, D, R)) = R
Data (Build (L, D, R)) = D
Is_empty (Create) = true
Is_empty (Build (L, D, R)) = false
Contains (Create, E) = false
Contains (Build (L, D, R), E) = **if** E= D **then** true **else** if E < D **then**
Contains (L, D) **else** Contains (R, D)

Figure 10.6 The specification of sort Binary_tree

10.2 Incremental specification

Writing formal specifications is time-consuming and is an expensive software process activity. A good strategy to minimise the amount of effort needed to develop a specification is to reuse specifications which have already been developed. To do this, you need to derive specifications in an incremental way. Simple specifications then serve as building blocks for more complex specifications.

There are a number of different ways in which specifications can be reused. I will discuss three of these here namely:

1. The instantiation of generic specifications.

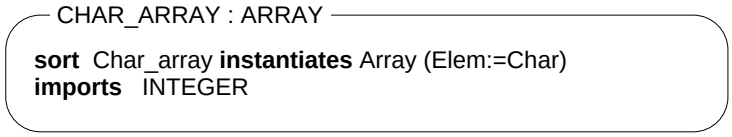

Figure 10.7 The specification of a character array

2. The incremental development of specifications.
3. The enrichment of specifications.

10.2.1 Specification instantiation

The simplest form of reuse is to take an existing specification which has been specified with a generic parameter and instantiate this with some other sort. Figure 10.7 shows an example of how the array specification given in Figure 10.3 can be instantiated to create the specification of an array of characters. I assume that the sort Char has been defined in a separate specification. It must have a constant operation called Undefined associated with it. This could be implemented using some reserved bit pattern.

To instantiate a specification, the name of the generic specification is given along with the name of the specification being defined. The new sort name is defined by instantiating it with the name of the generic sort and the element type. When specifications are instantiated, the set of operations available is the same as the set of operations in the generic specification.

10.2.2 Incremental development

The incremental development of specifications involves developing simple specifications then using these to specify more complex entities. The simple specifications are imported into the more complex specifications. This means the operations which are defined on the imported specifications are available for use in the importing specification.

Figure 10.8 is an example of a general-purpose specification building block. A basic building block for a graphical system is an object class representing a Cartesian coordinate. Figure 10.8 shows an example of a simple algebraic specification of a sort called Coord. The operations are create a coordinate, test coordinates for equality and access the X and Y components.

The specification of Coord can be used in the specification of a cursor in a graphical user interface (Figure 10.9). Cursors can be moved around the screen to point at a particular screen element. They have an associated representation (such as an arrow) which may change depending on the area of the screen where the cursor is positioned. This is supported in the specification by importing the specification of a bitmap (not defined here). Assume that the change of cursor is invoked by some event handler which detects the cursor position with respect to other displayed objects. Note that the operations defined in COORD may be accessed directly if their

name is distinct from names defined in CURSOR. If the names clash, the operation name in the imported specification must be preceded by the specification name.

COORD

sort Coord
imports INTEGER, BOOLEAN

Defines a sort representing a Cartesian coordinate. The operations defined on Coord are X and Y which evaluate the x and y attributes of an entity of this sort and Eq which compares two entities of sort Coord for equality.

Create (Integer, Integer) → Coord ;
X (Coord) → Integer ;
Y (Coord) → Integer ;
Eq (Coord, Coord) → Boolean ;

X (Create (x, y)) = x
Y (Create (x, y)) = y
Eq (Create (x1, y1), Create (x2, y2)) = ((x1= x2) and (y1=y2))

Figure 10.8 The specification of sort Coord

CURSOR

sort Cursor
imports INTEGER, COORD, BITMAP

A cursor is a representation of a screen position. Defined operations are Create which associates an icon with the cursor at a screen position, Position which returns the current coordinate of the cursor, Translate which moves the cursor a given amount in the x and y directions and Change_Icon which causes the cursor icon to be switched.

The Display operation is not defined formally. Informally, it causes the icon associated with the cursor to be displayed so that the top-left corner of the icon represents the cursor's position. When displayed, the 'clear' parts of the cursor bitmap should not obscure the underlying objects.

Create (Coord, Bitmap) → Cursor
Translate (Cursor, Integer, Integer) → Cursor
Position (Cursor) → Coord
Change_Icon (Cursor, Bitmap) → Cursor
Display (Cursor) → Cursor

Translate (Create (C, Icon), xd, yd) =
    Create (COORD.Create (X(C)+xd, Y(C)+yd), Icon)
Position (Create (C, Icon)) = C
Position (Translate (C, xd, yd)) = COORD.Create (X(C)+xd, Y(C)+yd)
Change_Icon (Create (C, Icon), Icon2) = Create (C, Icon2)

Figure 10.9 The specification of sort Cursor

The specification of a cursor class illustrates a problem with algebraic specification. It is difficult to use this approach to specify input and output operations. Most formal specification techniques are deficient in this respect. However, the algebraic approach is particularly inconvenient because it is difficult to specify global state changes which are the side-effects of operations.

It is good design practice in an object-oriented system for each object class to include an operation which displays objects. In the case of cursors, the icon associated with the cursor is displayed in such a way that the 'clear' parts of the cursor icon does not obscure other display objects. In general, cursor icons occupy a group of coordinates only one of which is the cursor 'hot spot'. The specification must establish the position of this hot-spot with respect to the cursor icon.

This is an example of an informal specification which is shorter and more readable than a formal specification. When formal specifications are developed as part of the software specification process, it is always important to keep in mind that formality is usually intended to clarify the specification. It is not an objective in its own right. You should not insist on formality when it does not have any real benefits. In the specification of Cursor given in Figure 10.9, I have therefore excluded the Display operation from the formal part of the specification.

10.2.3 Specification enrichment

The enrichment of a specification is like inheritance in object-oriented development. The operations and axioms on the base sort are inherited and become part of the specification. New operations in the specification may overwrite operations with the same name in the base sort, operations may be added to the base specification or removed from it.

Enrichment is not the same as importing a specification. When a specification is imported, the sort and its operations defined in the imported specification are made accessible (brought into the scope of in programming language terms) to the specification being defined. They do not become part of that specification.

As an illustration of enrichment, consider the list specification defined in Figure 10.4. A list with additional functionality is needed where elements can be added to either end and an operation to test for list membership is included. Figure 10.10 summarises the operations on the List sort. The Add operation adds an element to the front of the list and the Member operation tests if a given value is contained in the list.

Figure 10.11 shows the basic definition of List may be used as a component in the definition of a sort New_List which is an enriched version of the sort List. New_List inherits the operations and axioms defined on List so that these also apply to that sort. In effect, they could be written into the specification NEW_LIST with the name List replaced by New_List.

To complete the specification the access operations Head, Tail and Member must be defined over the new constructor (Add) and Member must be specified over previously defined constructor operations.

Operation	Description
Create	Brings a list into existence.
Cons (New_list, Elem)	Adds an element to the end of the list.
Add (New_list, Elem)	Adds an element to the front of the list.
Head (New_list)	Returns the first element in the list.
Tail (New_list)	Returns the list with the first element removed.
Member (New_list, Elem)	Returns true if an element of the list matches Elem
Length (New_list)	Returns the number of elements in the list

Figure 10.10 The operations on sort List

When a sort is created by enrichment, the names of the generic parameters of the base sort are inherited. The generic parameters in an enriched specification must include the operations from the base sort. In NEW_LIST, the parameters of LIST are extended with an additional equality operation ("=").

In the above examples of array and list specifications, the operations on a sort have been shown as functions which evaluate to a single atomic value. In many cases, this is a reasonable model of the system which is being specified. However, there are some classes of operation which, when implemented, involve modifying more than one entity. For example, the familiar stack pop operation returns a value from a stack and also removes the top element from the stack.

It is possible to model such operations using multiple simpler operations which take the top value from the stack and which remove the top stack element. However, a more natural approach is to define operations which return a tuple rather than a single value. Rather than returning a single value, the function has multiple output values. Thus, the stack pop operation might have the signature:

NEW_LIST (Elem: [Undefined → Elem; .=. → Boolean])

sort New_List **enrich** List
imports INTEGER, BOOLEAN

Defines an extended form of list which inherits the operations and properties of the simpler specification of List and which adds new operations (Add and Member) to these.
See Figure 10.10 for a description of the list operations.

Add (New_List, Elem) → New_List
Member (New_List, Elem) → Boolean

Add (Create, v) = Cons (Create, v)
Member (Create, v) = false
Member (Add (L, v), v1) = ((v = v1) **or** Member (L, v1))
Member (Cons (L, v), v1) = ((v = v1) **or** Member (L, v1))
Head (Add (L, v)) = v
Tail (Add (L, v)) = L
Length (Add (L, v)) = Length (L) + 1

Figure 10.11 The specification of sort List

Operation	Description
Create	Brings a queue into existence.
Cons (Queue, Elem)	Adds an element to the end of the queue.
Head (Queue)	Returns the element at the front of the queue.
Tail (Queue)	Returns the queue minus its front element.
Length (Queue)	Returns the number of elements in the queue.
Get (Queue)	Returns a tuple composed of the element at the head of the queue and the queue with the front element removed

Figure 10.12 The operations on sort Queue

QUEUE (Elem: [Undefined → Elem])

sort Queue **enrich** List
imports INTEGER

This specification defines a queue which is a first-in, first-out data structure. It can therefore be specified as a List where the insert operation adds a member to the end of the queue.
See Figure 10.12 for a description of queue operations.

Get (Queue) → (Elem, Queue)

Get (Create) = (Undefined, Create)
Get (Cons (Q, v)) = (Head (Q), Tail (Cons (Q, v)))

Figure 10.13 The specification of a queue

Pop (Stack) → (Elem, Stack)

Operations which evaluate to a tuple are used in a specification of a queue which can be specified as an enrichment of lists. An operation is added which evaluates to a pair consisting of the first item on the queue and the queue minus its head. The operations on sort Queue are shown in Figure 10.12 and the queue specification in Figure 10.13.

10.3 Error specification

A problem which faces the developer of a specification is how to indicate errors and exceptional conditions. The basic problem is that under normal conditions, the result of an operation may be of some sort X, but under exceptional conditions, an error should be indicated. The appropriate error indicator may not be of the same sort as the normal result so a type clash occurs.

There are several ways of tackling this problem. Three possibilities are:

1. A special distinguished, constant operation such as Undefined may be defined. In exceptional cases, the operation evaluates to Undefined. We have already seen examples of this technique in the array specification in Figure 10.3. The Eval operation evaluates to Undefined if the index is out of bounds. The value Undefined is untyped so can be the result of any specification operation.
2. The operation may evaluate to a tuple where one component of the tuple indicates whether or not the operation has evaluated successfully. The specification of a queue shown in Figure 10.13 illustrated how a tuple could be the result of an operation. The examples in Chapter 20 which is concerned with software reuse illustrate how this approach can be implemented.
3. The specification may include an exceptions section which defines conditions under which the axioms do not hold.

Figure 10.14 illustrates how an exceptions section can be added to the specification of a list which was introduced in Figure 10.4. In this case, the exceptions part specifies that if the length of the list L is 0 then the Head operation, applied to L, fails. Notice that this means that no operations need be associated with the generic specification parameter. Guttag [Guttag, 1980 #499] discusses this approach to error specification in more detail.

LIST (Elem)

sort List
imports INTEGER

See Figure 10.4

Create → List
Cons (List, Elem) → List
Tail (List) → List
Head (List) → Elem
Length (List) → Integer

Head (Cons (L, v)) = **if** L = Create **then** v **else** Head (L)
Length (Create) = 0
Length (Cons (L, v)) = Length (L) + 1
Tail (Create) = Create
Tail (Cons (L, v)) = **if** L = Create **then** Create **else** Cons (Tail (L), v)

exceptions
Length (L) = 0 **failure** (Head (L))

Figure 10.14 The specification of List with an exception part

KEY POINTS

- Algebraic specification is a particularly appropriate technique when interfaces between software systems must be specified.
- Algebraic specification involves designing the operations on an abstract data type or object and specifying them in terms of their inter-relationships.
- An algebraic specification consists of two formal parts. A signature part where the operations and their parameters are set out and an axioms part where the relationships between these operations are defined.
- Formal specifications should always have an associated informal description to make the formal semantics more understandable.
- Algebraic specifications should be constructed by identifying constructor operations, which create instances of the type or class, and inspection operations which inspect the values of these instances. The semantics of each inspection operation should be defined for each constructor.
- Complex formal specifications may be constructed from simple building blocks. Specifications can be developed from simpler specifications by instantiating a generic specification, incremental specification development and specification enrichment.
- Errors in operations can be specified by identifying distinguished 'error values', by associating an error indicator with the value of an operation or by incorporating a special section in a specification which defines values for exceptional situations.

FURTHER READING

Recent books on algebraic specification have concentrated on describing the method on its own without placing it in the context of a wider software development process. I have therefore suggested these older references which should be available in libraries. They are better for understanding how this technique can be used.

The Specification of Complex Systems This excellent introductory text contains a good chapter discussing algebraic specification. A simple electronic mail system is used as an example. (B. Cohen, W.T. Harwood and M.I. Jackson, 1986, Addison-Wesley)

'Formal Specification as a Design Tool'. This paper is included in a collection of papers on specification which includes other papers on algebraic specification. I think this paper is particularly useful as it illustrates the practical use of formal specification. (J.V. Guttag and J.J. Horning, in *Software Specification Techniques*, Gehani, N. and McGettrick, A.D. (eds.), 1986, Addison-Wesley)

Abstraction and Specification in Program Development This is a general text on systems development with good chapters on algebraic specification. (B. Liskov and J. Guttag, 1986, MIT Press)

EXERCISES

10.1 Explain why formal specification is a valuable technique for defining the interfaces between sub-systems.

10.2 An abstract data type representing a stack has the following operations associated with it:

New: Bring a stack into existence
Push: Add an element to the top of the stack
Top: Evaluate the element on top of the stack
Retract: Remove the top element from the stack and return the modified stack
Empty: True if there are no elements on the stack

Write an algebraic specification of this stack. Make reasonable assumptions about the syntax and semantics of the stack operations.

10.3 Modify the example presented in Figure 10.3 (array specification) by adding a new operation called ArrayAssign which assigns all the values of one array to another array given that the arrays have the same number of elements.

10.4 An abstract data type called Set has a signature defined as follows:

New → Set
Add (Set, Elem) → Set
Size (Set) → Integer
Remove (Set) → Elem
Contains (Set, Elem) → Boolean
Delete (Set, Elem) → Set

Explain informally what these operations are likely to do. Write axioms which formally define your informal English specification.

10.5 Using the equation rewriting approach as used in Example 10.4, verify that the operation Add ([10, 7, 4], 8) on the list defined in Figure 10.11 causes the list [8, 10, 7, 4] to be built. (Hint: Show the head of the list is 8 and the tail is [10, 7, 4]).

10.6 Write a formal algebraic specification of a sort Symbol_table whose operations are informally defined as follows:

Create: Bring a symbol table into existence
Enter: Enter a symbol and its type into the table.
Lookup: Return the type associated with a name in the table.
Delete: Remove a name, type pair from the table, given a name as a parameter.
Replace: Replace the type associated with a given name with the type specified as a parameter.

The Enter operation fails if the name is in the table. The Lookup, Delete and Replace operations fail if the name is not in the table.

10.7 Discuss how your specification would have to be modified if a block-structured symbol table was required. A block structured symbol table is one used in compiling a language with block structure like Pascal where declarations in an inner block override the outer block declarations if the same name is used.

10.8 Enrich the specification of List (Figure 10.11) with further operations to implement an ordered list. Add a new operation called Insert which inserts an element in the correct place to maintain the ordering and an operation Remove which, given an element value, removes the element with that value from the list.

10.9 For all of the abstract data types you have specified, write Ada or C++ package specifications defining a package to implement the abstract type. Pay particular attention to error handling in the implementation.