Algebraic Specification

Learning Objective

...Specifying abstract types in terms of relationships between type operations.

Frederick T Sheldon
Assistant Professor of Computer Science
Washington State University
Objectives

⊗ To explain the role of formal specifications in sub-system interface definition
⊗ To introduce the algebraic approach to formal specification
⊗ To describe the systematic construction of algebraic specifications
⊗ To illustrate a number of incremental ways to write algebraic specifications
Topics covered

- Systematic algebraic specification
- Structured specification
- Error specification
Sub-system interfaces

Sub-system A

Interface objects

Sub-system B

CS 580.1/483.1 Software Specification and Analysis
Instructor: F.T. Sheldon
Interface specification

- Formal specification is particularly appropriate for *defining sub-system interfaces*. It provides an unambiguous interface description and allows for parallel sub-system development.
- Interfaces may be *defined as a set of abstract data types or object classes*.
- Algebraic specification is particularly appropriate for *ADT specification as it focuses on operations and their relationships*.
Specification structure

- **Introduction**
  - Introduces the *sort* (type) name and imported specifications

- **Informal description**
  - Describes the *type or object class operations*

- **Signature**
  - Defines the *syntax of the type or class operations*

- **Axioms**
  - Defines axioms which *characterize the behavior of the type*
Specification format

\[ \text{sort} \ < \text{name} > \]
\[ \text{imports} \ < \text{LIST OF SPECIFICATION NAMES} > \]

- Informal description of the sort and its operations
- Operation signatures setting out the names and the types of the parameters to the operations defined over the sort
- Axioms defining the operations over the sort
### Specification of an array

**Signature**

<table>
<thead>
<tr>
<th>Create (Integer, Integer) → Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign (Array, Integer, Elem) → Array</td>
</tr>
<tr>
<td>First (Array) → Integer</td>
</tr>
<tr>
<td>Last (Array) → Integer</td>
</tr>
<tr>
<td>Eval (Array, Integer) → Elem</td>
</tr>
</tbody>
</table>

**Axioms**

<table>
<thead>
<tr>
<th>First (Create (x, y)) = x</th>
</tr>
</thead>
<tbody>
<tr>
<td>First (Assign (a, n, v)) = First (a)</td>
</tr>
<tr>
<td>Last (Create (x, y)) = y</td>
</tr>
<tr>
<td>Last (Assign (a, n, v)) = Last (a)</td>
</tr>
<tr>
<td>Eval (Create (x, y), n) = Undefined</td>
</tr>
<tr>
<td>Eval (Assign (a, n, v), m) =</td>
</tr>
</tbody>
</table>

\[
\text{if } m < \text{First (a)} \text{ or } m > \text{Last (a)} \text{ then Undefined else } \text{if } m = n \text{ then } v \text{ else } \text{Eval (a, m)}
\]

Arrays are collections of elements of generic type Elem. They have a lower and upper bound (discovered by the operations First and Last). Individual elements are accessed via their numeric index. Create takes the array bounds as parameters and creates the array, initialising its values to Undefined. Assign creates a new array which is the same as its input with the specified element assigned the given value. Eval reveals the value of a specified element. If an attempt is made to access a value outside the bounds of the array, the value is undefined.

**ARRAY** (Elem: [Undefined → Elem])
Systematic algebraic specification

- Algebraic specifications of a system may be developed in a systematic way
  - Specification structuring
  - Specification naming
  - Operation selection
  - Informal operation specification
  - Syntax definition
  - Axiom definition
Specification operations

- **Constructor operations** – Operations which create entities of the type being specified

- **Inspection operations** – Operations which evaluate entities of the type being specified

- **Specify behavior** – Define the **inspector operations for each constructor operation**
Operations on a list ADT

- **Constructor operations** – evaluate to sort List
  - Create, Cons and Tail

- **Inspection operations** – take sort List as a parameter and return some other sort
  - Head and Length

- Tail can be defined using the simpler constructors Create and Cons. No need to define Head and Length with Tail.
List specification

**sort** List
**imports** INTEGER

Defines a list where elements are added at the end and removed from the front. The operations are Create, which brings an empty list into existence, Cons, which creates a new list with an added member, Length, which evaluates the list size, Head, which evaluates the front element of the list, and Tail, which creates a list by removing the head from its input list.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create → List</td>
<td></td>
</tr>
<tr>
<td>Cons (List, Elem) → List</td>
<td></td>
</tr>
<tr>
<td>Tail (List) → List</td>
<td></td>
</tr>
<tr>
<td>Head (List) → Elem</td>
<td></td>
</tr>
<tr>
<td>Length (List) → Integer</td>
<td></td>
</tr>
</tbody>
</table>

Head (Create) = Undefined -- Error to evaluate an empty list
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)
Recursion in specifications

- Operations are often specified recursively
- $\text{Tail} \ (\text{Cons} \ (L, \ v)) = \text{if} \ L = \text{Create} \ \text{then} \ \text{Create} \ \text{else} \ \text{Cons} \ (\text{Tail} \ (L), \ v)$

  $\text{Cons} \ ([5, 7], 9) = [5, 7, 9]
  \text{Tail} \ ([5, 7, 9]) = \text{Tail} \ (\text{Cons} \ ([5, 7], 9)) =
  \text{Cons} \ (\text{Tail} \ ([5, 7]), 9) = \text{Cons} \ (\text{Tail} \ (\text{Cons} \ ([5], 7)), 9) =
  \text{Cons} \ (\text{Cons} \ (\text{Tail} \ ([5]), 7), 9) =
  \text{Cons} \ (\text{Cons} \ (\text{Tail} \ (\text{Cons} \ ([], 5)), 7), 9) =
  \text{Cons} \ (\text{Cons} \ ([\text{Create}], 7), 9) = \text{Cons} \ ([7], 9) = [7, 9]$
Primitive constructors

- It is sometimes necessary to introduce additional constructors to simplify the specification.
- The other constructors can then be defined using these more primitive constructors.
- In the binary tree specification, a primitive constructor “Build” is added.
# Operations on a binary tree

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>Creates an empty tree.</td>
</tr>
<tr>
<td>Add (Binary_tree, Elem)</td>
<td>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 subtree.</td>
</tr>
<tr>
<td>Left (Binary_tree)</td>
<td>Returns the left sub-tree of the top of the tree.</td>
</tr>
<tr>
<td>Data (Binary_tree)</td>
<td>Returns the value of the data element at the top of the tree.</td>
</tr>
<tr>
<td>Right (Binary_tree)</td>
<td>Returns the right sub-tree of the top of the tree.</td>
</tr>
<tr>
<td>Is_empty (Binary_tree)</td>
<td>Returns true if the tree does not contain any elements.</td>
</tr>
<tr>
<td>Contains (Binary_tree, Elem)</td>
<td>Returns true if the tree contains the given element.</td>
</tr>
</tbody>
</table>
## Binary tree specification

```plaintext

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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create → Binary_tree</td>
<td></td>
</tr>
<tr>
<td>Add (Binary_tree, Elem) → Binary_tree</td>
<td></td>
</tr>
<tr>
<td>Left (Binary_tree) → Binary_tree</td>
<td></td>
</tr>
<tr>
<td>Data (Binary_tree) → Elem</td>
<td></td>
</tr>
<tr>
<td>Right (Binary_tree) → Binary_tree</td>
<td></td>
</tr>
<tr>
<td>Is_empty (Binary_tree) → Boolean</td>
<td></td>
</tr>
<tr>
<td>Contains (Binary_tree, Elem) → Boolean</td>
<td></td>
</tr>
<tr>
<td>Build (Binary_tree, Elem, Binary_tree) → Binary_tree</td>
<td></td>
</tr>
</tbody>
</table>

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)
```
Structured specification

- Specifications should be constructed in a structured way. Other specifications should be reused whenever possible.
- Specification instantiation – A generic specification is instantiated with a given sort.
- Incremental specification – Use simple specifications in more complex specifications.
- Specification enrichment – A specification is constructed by inheritance from other specifications.
Specification instantiation

\begin{itemize}
  \item \textbf{sort} Char_array \textbf{instantiates} Array (Elem:=Char)
  \item \textbf{imports} INTEGER
\end{itemize}
Incremental specification

- Develop a simple specification then use this in more complex specifications
- Try to establish a library of specification building blocks that may be reused
- In a graphical user interface, the specification of a Cartesian coordinate can be reused in the specification of a cursor
- Display operations are hard to specify algebraically. May be informally specified
## Coord specification

<table>
<thead>
<tr>
<th>COORD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sort</strong> Coord</td>
</tr>
<tr>
<td><strong>imports</strong> INTEGER, BOOLEAN</td>
</tr>
</tbody>
</table>

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))
Cursor specification

<table>
<thead>
<tr>
<th>CURSOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
</tr>
<tr>
<td>imports</td>
</tr>
</tbody>
</table>

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

\[
\text{Translate} (\text{Create} (C, \text{Icon}), \text{xd}, \text{yd}) = \text{Create} (\text{COORD.Create} (X(C)+\text{xd}, Y(C)+\text{yd}), \text{Icon})
\]

\[
\text{Position} (\text{Create} (C, \text{Icon})) = C
\]

\[
\text{Position} (\text{Translate} (C, \text{xd}, \text{yd})) = \text{COORD.Create} (X(C)+\text{xd}, Y(C)+\text{yd})
\]

\[
\text{Change_Icon} (\text{Create} (C, \text{Icon}), \text{Icon 2}) = \text{Create} (C, \text{Icon2})
\]
Specification enrichment

- Starting with a reusable specification building block, new operations are added to create a more complex type.
- **Enrichment can be continued to any number of levels. It is comparable to inheritance.**
- Not the same as importing a specification:
  - Importing makes a specification available for use
  - Enrichment creates a specification for a new sort
- The names of the generic parameters of the base sort are inherited when a sort is enriched
## Operations on New_list

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>Brings a list into existence.</td>
</tr>
<tr>
<td>Cons (New_list, Elem)</td>
<td>Adds an element to the end of the list.</td>
</tr>
<tr>
<td>Add (New_list, Elem)</td>
<td>Adds an element to the front of the list.</td>
</tr>
<tr>
<td>Head (New_list)</td>
<td>Returns the first element in the list.</td>
</tr>
<tr>
<td>Tail (New_list)</td>
<td>Returns the list with the first element removed.</td>
</tr>
<tr>
<td>Member (New_list, Elem)</td>
<td>Returns true if an element of the list matches Elem</td>
</tr>
<tr>
<td>Length (New_list)</td>
<td>Returns the number of elements in the list</td>
</tr>
</tbody>
</table>
New_list specification

NEW_LIST ( Elem: [Undefined \(\rightarrow\) Elem; \(\rightarrow\) 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) \(\rightarrow\) New_List

Member (New_List, Elem) \(\rightarrow\) 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
Multi-value operations

- **Some operations affect more than one entity**
  - Logically, a function returns more than one value

- **Stack pop** operation returns both the value popped from the stack AND the modified stack

- May be modeled algebraically using multiple operations (TOP and RETRACT for a stack) but a *more intuitive approach is to define operations which return a tuple* rather than a single value
# Queue operations

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

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) \rightarrow (Elem, Queue)

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

- Under normal conditions the result of an operation may be sort X but under exceptional conditions, an error should be indicated and the returned sort is different.

- Problem may be tackled in three ways
  - Use a special distinguished constant operation (Undefined) which conforms to the type of the returned value. 
    » See Array specification
  - Define operation evaluation to be a tuple, where an element indicates success of failure.
    » See Queue specification
  - Include a special failure section in the specification
List with exception part

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 fi failure (Head (L))
Key points

- Algebraic specification is particularly appropriate for **sub-system interface specification**
- Algebraic specification involves specifying operations on an abstract data types or **object in terms of their inter-relationships**
- An algebraic specification has a **signature** part defining syntax and an **axioms** part defining semantics
- Formal specifications should have an associated **informal description** to make them more readable
Key points

- Algebraic specifications may be defined by defining the semantics of each inspection operation for each constructor operation.
- Specification should be developed incrementally from simpler specification building blocks.
- Errors can be specified either by defining distinguished error values, by defining a tuple where one part indicates success or failure or by including an error section in a specification.