Algebraic Specification

Specifying abstract types in terms of relationships between type operations

Topics for Today

• The algebraic approach to formal specification

Interface Specification

• Formal specifications are particularly appropriate for defining sub-system interfaces
• 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
Algebraic Specification Overview

**Algebraic 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**

```
algebra specName
  introduces
    sort whatYouDefine;
    imports whatYouImport;
  Informal description of the sort and its operations
  operations
    Operation signatures defining the names and types of the parameters to the operations on the sort
  constrains allOperationsInSort so that
    Axioms defining the operations over the sort
```

**Specification of an Array**

- Arrays are collections of generic type Elem
  - They have a lower and upper bound (accessed through the operations First and Last)
  - Individual elements are accessed through their individual index
- Create creates an array with all cells initialized to Undefined
  - Assign creates a new array which is the same as its input with the specified element assigned the new value
  - Eval reveals the value of an element
  - If Eval is attempted outside the bounds, Undefined is returned
The Interface to the Array

```latex
\textbf{algebra} \texttt{arraySpec} \\
\textbf{introduces} \\
\texttt{sort Array;} \\
\textbf{imports} \texttt{INTEGER;} \\
\textbf{operations} \\
\texttt{Create (Int, Int) \rightarrow Array} \\
\texttt{Assign (Array, Int, Elem) \rightarrow Array} \\
\texttt{First (Array) \rightarrow Int} \\
\texttt{Last (Array) \rightarrow Int} \\
\texttt{Eval (Array, Int) \rightarrow Elem}
```

How the Array Works

```latex
\textbf{constrains} \texttt{Create, Assign, First, Last, Eval so that} \\
\textbf{for all} \{x, y, n, m : Int, v : Elem\}
```

Specification Operations

- Constructor operations
  - Operations which create entities of the type being specified
- Inspection operations
  - Operations which evaluate entities of the type being specified
- To specify behavior, define the inspector operations for each constructor operation
Operations on a String ADT

- Constructor operations which evaluate to sort String
  - new, append, add
- Inspection operations which take sort String as a parameter and return some other sort
  - isEmpty, length, equal
- append can be defined using the simpler constructors new and add

The Interface to the String

```plaintext
algebra stringSpec
  introduces
  sorts String;
  imports Char, Nat, Bool;
  operations
      new () → String
      append (String, String) → String
      add (String, Char) → String
      length (String) → Nat
      isEmpty (String) → Bool
      equal (String, String) → Bool
```

How the String Works

```plaintext
constrains new, append, add, length, isEmpty so that for all [x, y, z : String, c : Elem]
```
Recursion in Specifications

- Operations are often specified recursively
- \(\text{append}(y, \text{add}(z, c)) = \text{add}(\text{append}(y, z), c)\)
  - \(\text{append}(\text{“ab”}, \text{“cd”})\)
  - \(\text{append}(\text{“ab”}, \text{add}(\text{“c”}, d))\)
  - \(\text{add}(\text{append}(\text{“ab”}, \text{“c”}), d)\)
  - \(\text{add}(\text{append}(\text{“ab”}, \text{add}(\text{new()}, c), d)\)
  - \(\text{add}(\text{add}(\text{append}(\text{“ab”}, \text{new()}), c), d)\)
  - \(\text{add}(\text{add}(\text{“ab”}, c), d)\)
  - \(\text{add}(\text{“abc”}, d)\)
  - \(\text{“abed”}\)

More Recursion

- \(\text{equal}(\text{add}(x, c), \text{add}(y, c)) = \text{equal}(x, y)\)
  - \(\text{equal}(\text{“abc”}, \text{“abc”})\)

Recursion Again

- \(\text{equal}(\text{add}(x, c), \text{add}(y, c)) = \text{equal}(x, y)\)
  - \(\text{equal}(\text{“abc”}, \text{“axc”})\)
How the String Works (right!)

- constrains new, append, add, length, isEmpty so that
- for all \([x, y, z : \text{String}, c, d : \text{Elem}]\)
- isEmpty(new()) = true
- isEmpty(append(x, c)) = false
- length(new()) = 0
- length(append(x, c)) = length(x) + 1
- append(c, new()) = c
- append(x, append(y, z)) = append(append(x, y), z, c)
- equalC('a', 'a') = true
- equalC('a', 'b') = false
- ...
- equal(new(), new()) = true
- equal(new(), append(x, c)) = false
- equal(append(x, c), new()) = false
- equal(new(), append(x, c)) = true
- equal(append(x, c), append(y, d)) = equal(x, y) and equalC(c, d)

Recursion Yet Again

- equal(append(x, c), append(y, d)) = equal(x, y) and equalC(c, d)
- equalC(“abc”, “axc”)

Primitive Constructors

- It is sometimes necessary to introduce additional constructors to simplify the specification
- For example, equalC
- The other constructors can then be defined using these more primitive constructors
Stack as an Algebra

```plaintext
algebra stackSpec
introduces
  sorts Stack;
  imports Elem, Int, Bool;
  create() → Stack
  empty(Stack) → Bool
  push(Stack, Elem) → Stack
  pop(Stack) → Stack
  top(Stack) → Elem
  height(Stack) → Int
```

In Class Work

- Define the axioms for the Stack ADT
- Back here again at
- Spare time?
  - Define bottom(Stack) → Elem

We Have Learned

- 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
- Algebraic specifications may be defined by defining the semantics of each inspection operation for each constructor operation