ECE 321 Software Requirements Engineering

Lecture 9: Algebraic specifications

Mid-term course and instruction feedback

- You should have received an email requesting this feedback
- Please fill it in!
  - It gives me feedback on how to improve the course
  - Both positive and negative feedback are welcome
    - Please be constructive :)

Algebraic specifications

- Descriptive specifications
  - Describe system in terms of its operations and how they cooperate
  - Describe sets of values and operations on them
- Many formalisms exist
  - Larch, CCS, Lotos
  - All use abstract algebra

‘Classical’ vs. ‘abstract’ algebra

- Classical
  - $x + y + z = 10$
    - x, y, z are numbers
- Abstract
  - Structured expressions, strings and words (and their relationships)

Abstract algebra

- An algebraic structure is an arbitrary set
  - With one or more operations
  - With rules
- Abstract algebra is the study of these structures

A simple example of an algebraic structure

- The set of all possible colours C

```
For all [c1, c2 : C]
```

$\texttt{Merge}(c_1,\ c_2)\ \texttt{==}\ \texttt{Merge}(c_2,\ c_1)$ (*commutative*)

- Abstract algebra is not necessarily about mathematics / numbers

Terminology of abstract specifications

- **Sort**
  - A set within an algebra
  - e.g., Integer, Stack, Boolean
- **Operation**
  - Maps tuples of values to values
  - e.g., +, IsEmpty
- **Signature**
  - Tuple of sorts for domain (‘input’) and range (‘output’)
  - e.g., Int, Int -> Int, or Stack -> Boolean

More terminology of abstract specifications

- **Generators**
  - Operations that are used to create instances of an algebra
  - The minimal set of operators that can generate all other operators within an algebra

Algebraic specification of a stack

```
algebra StackOfItems
   imports Boolean;
   introduces
      sorts Stack, Item;
      operations
         Create: -> Stack;
         IsEmpty: Stack -> Boolean;
         Push: Stack × Item -> Stack;
         Pop: Stack -> Stack;
         Top: Stack -> Item;
   constrains Create, IsEmpty, Push, Pop, Top so that
      Stack generated by [Create, Push]
```

Algebraic specification of a stack

```
algebra StackOfItems
   imports Boolean;
   introduces
      sorts Stack, Item;
```

Algebra of **StackOfItems** is <u>defined</u>

Boolean type is <u>imported</u> (it is defined in some other place)

Two new **sorts** (sets) in the algebra <u>introduced</u>:

Stack

Item

Algebraic specification of a stack

```
operations
    Create: -> Stack;
    IsEmpty: Stack -> Boolean;
    Push: Stack × Item -> Stack;
    Pop: Stack -> Stack;
    Top: Stack -> Item;
```

Some **operations** on the introduced sorts are defined, e.g.:

- **Create** to create an empty stack; returns Stack
- **Push** to add an Item to stack; return Stack

Note that either domain or range should contain at least one Stack

Algebraic specification of a stack

```
constrains Create, IsEmpty, Push, Pop, Top so that
    Stack generated by [Create, Push]
```

Create and **Push** are the generators and thus left undefined

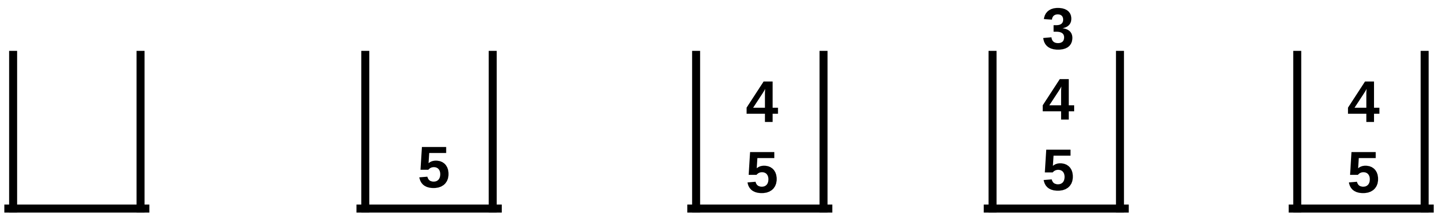
The other operations are defined using the generators

We always choose a minimal set of generators

Generating a stack with its generators

Pop(Push(Push(Push(Create,5),4),3))

Generating a stack with its generators



Pop(Push(Push(Push(Create,5),4),3))

== Push(Push(Push(Create,5),4),3)

- Any state can be generated as a sequence of pushes and a create
- **Push** and **Create** can define any other operation
- Also we see that **Pop** has to be preceded by **Push**

Algebraic specification of a stack

```
algebra StackOfItems
   imports Boolean;
   introduces
      sorts Stack, Item;
      operations
         Create: -> Stack;
         IsEmpty: Stack -> Boolean;
         Push: Stack × Item -> Stack;
         Pop: Stack -> Stack;
         Top: Stack -> Item;
   constrains Create, IsEmpty, Push, Pop, Top so that
      Stack generated by [Create, Push]
```

We are still missing the relations between the operations!

- For every operation, we need to specify how it behaves for each generator (**Create** and **Push**) using **axioms**
- The idea is that:
  - **Create** represents an empty stack
  - **Push(someStack, someItem)** represents a non-empty stack
- So:
  - IsEmpty(Create) = ?
  - IsEmpty(Push(someStack, someItem)) = ?
  - Pop(Create) = ?
  - ... etc.

Let us complete the axioms together

```
algebra StackOfItems
   imports Boolean;
   introduces
      sorts Stack, Item;
      operations
         Create: -> Stack;
         IsEmpty: Stack -> Boolean;
         Push: Stack × Item -> Stack;
         Pop: Stack -> Stack;
         Top: Stack -> Item;
   constrains Create, IsEmpty, Push, Pop, Top so that
      Stack generated by [Create, Push]
```

The axioms for StackOfItems

```
    for all [s: Stack; i: Item]
      IsEmpty(Create) = true;
      IsEmpty(Push(s,i)) = false;
      Pop(Create) = error;
      Pop(Push(s,i)) = s;
      Top(Create) = error;
      Top(Push(s,i)) = i;
end StackOfItem;
```

Algebraic specification of Boolean

```
algebra BooleanSpec;
    introduces
      sorts Boolean;
      operations
         true :  -> Boolean;
         false : -> Boolean;
         ¬ : Boolean -> Boolean;
         ∧ : Boolean x Boolean -> Boolean;
         ∨ : Boolean x Boolean -> Boolean;
         ≡ : Boolean x Boolean -> Boolean;
    constrains true, false, ¬∧∨≡  so that
       Boolean generated by [true, false]
```

The axioms for Boolean

```
    for all [a: Boolean; b: Boolean]
        ¬ (true) = false;
        ¬ (false) = true;
      true ∧ true =  true;
      true ∧ false = false;
      …
      false ≡ false = true;
      false ≡ true = false;
      true ≡ false = false;
      true ≡ true = true;
      …
end BooleanSpec;
```

Some notes about algebraic specifications

- They can use
  - if ... then ... else
  - Recursion
- Order of operations does not matter
  - For readability it is best to first define basic operations

Algebraic specification of String

```
algebra StringSpec;
    introduces
      imports Integer, Boolean;
      sorts String, Char;
      operations
         Create: -> String;
         Append: String x String -> String;
         Add: String x Char -> String;
         Length: String -> Integer;
         IsEmpty: String -> Boolean;
         Equal: String x String -> Boolean;
    constrains Create, Append, Add, Length, IsEmpty, Equal so that
      String generated by [Create, Add]
```

The axioms for String 1/2

```
for all [s₁:String; s₂:String; c₁:Char; c₂:Char ]
  IsEmpty (Create) = true;
  IsEmpty (Add(s₁, c₁)) = false;
  Length (Create) = 0;
  Length (Add(s₁, c₁)) = Length(s₁)+1;
  Append (s₁,Create) = s₁;
  Append (s₁,Add(s₂,c₂)) = Add(Append(s₁,s₂), c₂);
```

The axioms for String 2/2

```
        Equal (Create, Create) = true;
        Equal (Create, Add(s1, c1)) = false;
        Equal (Add(s1, c1), Create) = false;
        Equal (Add(s1,c1), Add(s2,c2)) =
            if (c1 != c2) then false;
            else Equal(s1, s2);
end StringSpec;
```

Algebraic specification of Set

```
algebra SetOfItems
    imports Boolean, Integer;
    introduces
      sorts Set, Item;
    operations
      Create: -> Set;
      Add: Set x Item -> Set;
      Remove: Set x Item -> Set;
      Has: Set x Item -> Boolean;
      IsEmpty: Set -> Boolean;
      Cardinality: Set -> Integer;
    constrains Create, Add, Remove, Has, IsEmpty, Cardinality
      so that Set generated by [Create, Add]
```

The axioms for Set 1/2

```
for all [s: Set; i₁: Item; i₂: Item]
   IsEmpty(Create) = true;
   IsEmpty(Add(s,i₁)) = false;

   Has(Create, i₁) = false;
   Has(Add(s,i₁), i₂) =
      if (i₁ == i₂) then true;
       else  Has(s,i₂);
```

The axioms for Set 2/2

```
    Remove(Create, i₁) = Create;
    Remove(Add(s,i₁), i₂) =
       if (i₁ == i₂) then s;
       else Add(Remove(s,i₂), i₁);

    Cardinality(Create) = 0;
    Cardinality(Add(s,i₁)) = 1+Cardinality(s);

end SetOfItems;
```

More notes about algebraic specifications

- Multiple specifications may have the same syntax
  - But semantics help us to understand the specification properly
- Proper naming of operations and sorts is done to make understanding and documentation easier

About the seen specifications

- They use Larch
- Once a specification is written in Larch
  - You can test its properties
    - Independent of implementation language
  - You can generate source code (implementation)
    - Ports to several languages exist, including C, C++, Pascal, Ada, Smalltalk

Assignment 3

- Will be posted this afternoon on eClass
- Make sure to start early so you can ask questions!
- Final exam will contain a hands-on question about algebraic specifications