Algebraic Specifications
Supplement

Kenneth M. Anderson
Foundations of Software Engineering
CSCI 5828 — Spring 2008
Today’s Lecture

- Examine Algebraic Specifications
  - Compare Stack and Queue specifications
  - Use knowledge gained to look at example in textbook
Algebraic Specifications

- Algebras are akin to abstract data types
  - Sets of Values
  - With Three Types of Operators
    - **Generators**: Create new instance of data type
    - **Queries**: Answer questions about the data type
      - Return values are NOT instances of the data type but rather are boolean values or values stored inside the data type
    - **Manipulators**: return values of the data type but are not generators, they are altering an existing instance of the data type in some well defined way
Terminology

- Homogeneous Algebra
  Single set and its operations
- Heterogeneous Algebra
  Multiple sets and their operations
- Signature
  Collection of sets in a heterogeneous algebra
- Sort
  A set within an algebra
Terminology

- **Syntax**
  Signature plus operations with domains and ranges (i.e. functions)

- **Semantics**
  Equations involving operations; axioms
Algebraic Specification of Stack

algebra StackOfItem
Algebraic Specification of Stack

algebra StackOfItem
    imports Boolean;
Algebraic Specification of Stack

algebra StackOfItem
imports Boolean;
introduces
  sorts Stack, Item;
Algebraic Specification of Stack

algebra StackOfItem
  imports Boolean;
  introduces
  sorts Stack, Item;
  operations
    Create: → Stack;
    IsEmpty: Stack → Boolean;
    Push: Stack × Item → Stack;
    Pop: Stack → Stack;
    Top: Stack → Item;
Algebraic Specification of Stack

algebra StackOfItem
  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 Queue

algebra QueueOfItem
Algebraic Specification of Queue

algebra QueueOfItem
  imports Boolean;
algebra QueueOfItem
    imports Boolean;
    introduces
        sorts Queue, Item;
Algebraic Specification of Queue

algebra QueueOfItem
    imports Boolean;
    introduces
        sorts Queue, Item;
        operations
            Create: → Queue;
            IsEmpty: Queue → Boolean;
            Enqueue: Queue × Item → Queue;
            Dequeue: Queue → Queue;
            Front: Queue → Item;
Algebraic Specification of Queue

algebra QueueOfItem
  imports Boolean;
  introduces
    sorts Queue, Item;
    operations
      Create: → Queue;
      isEmpty: Queue → Boolean;
      Enqueue: Queue × Item → Queue;
      Dequeue: Queue → Queue;
      Front: Queue → Item;
  constrains Create, isEmpty, Enqueue, Dequeue, Front so that
  Queue generated by [Create, Enqueue]
Algebraic Specification of Pizza

algebra Nonsense
imports Boolean;
introduces
sorts Pizza, Car;
operations
Cat: \rightarrow Pizza;
Horse: Pizza \rightarrow Boolean;
Dog: Pizza \times Car \rightarrow Pizza;
Bird: Pizza \rightarrow Pizza;
Mouse: Pizza \rightarrow Car;
constrains Cat, Horse, Dog, Bird, Mouse so that
Pizza generated by [Cat, Horse]
Algebraic Specification of Stack

algebra StackOfItem
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]
How Generators Work

- The generators of Stack are Create and Push
  - We can think of generators as creating strings that can be “pattern matched” by other operators
- So, the following strings all represent stacks
  - Create
  - Push(Create, 1)
  - Push(Push(Create, 1), 2)
  - Push(Push(Push(Create, 1), 2), 3)
- In general, the Push operator has the form
  - Push(Stack, Item) and the result is a new Stack
Semantic Specification of Stack

for all [s: Stack; i: Item]

end StackOfItem;
for all [s: Stack; i: Item]
IsEmpty(Create) = true;
end StackOfItem;
Semantic Specification of Stack

for all [s: Stack; i: Item]  
IsEmpty(Create) = true;  
IsEmpty(Push(s,i)) = false;

These first two rules say:  
if you pass Create to IsEmpty  
we return true, otherwise  
we return false

end StackOfItem;
Semantic Specification of Stack

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

end StackOfItem;
Semantic Specification of Stack

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

end StackOfItem;

These next two rules say:
It is an error to pass Create to the Pop and Top operations
Semantic Specification of Stack

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

end StackOfItem;
Semantic Specification of Stack

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

These last two rules say:
If you Pop a stack, you get its internal stack. If you apply Top to a stack, you get its item.
How do Pop and Top work?

- Pop(Push(Push(Push(Create, 1), 2), 3))
- The rule says
  - Pop(Push(s,i)) = s;
- So, we apply the pattern match and the part in bold above matches “s” and so we return
  - Push(Push(Create, 1), 2)
- And have essentially popped the original stack

- Top(Push(Push(Push(Create, 1), 2), 3))
  - This expression evaluates to “3”
algebra QueueOfItem
  imports Boolean;
  introduces
  sorts Queue, Item;
  operations
  Create: → Queue;
  isEmpty: Queue → Boolean;
  Enqueue: Queue × Item → Queue;
  Dequeue: Queue → Queue;
  Front: Queue → Item;
  constrains Create, isEmpty, Enqueue, Dequeue, Front so that
  Queue generated by [Create, Enqueue]
Semantic Specification of Queue

for all \([q: \text{Queue}; i: \text{Item}]\)

end QueueOfItem;
Semantic Specification of Queue

for all [q: Queue; i: Item]
  isEmpty(Create) = true;

end QueueOfItem;
Semantic Specification of Queue

for all [q: Queue; i: Item]
  isEmpty(Create) = true;
  isEmpty(Enqueue(q,i)) = false;

end QueueOfItem;
for all [q: Queue; i: Item]
   IsEmpty(Create) = true;
   IsEmpty(Enqueue(q,i)) = false;
   Dequeue(Create) = error;
end QueueOfItem;
for all [q: Queue; i: Item]
IsEmpty(Create) = true;
IsEmpty(Enqueue(q,i)) = false;
Dequeue(Create) = error;
Front(Create) = error;

end QueueOfItem;
for all [q: Queue; i: Item]
IsEmpty(Create) = true;
IsEmpty(Enqueue(q,i)) = false;
Dequeue(Create) = error;
Front(Create) = error;
Dequeue(Enqueue(q,i))

end QueueOfItem;
for all [q: Queue; i: Item]
IsEmpty(Create) = true;
IsEmpty(Enqueue(q,i)) = false;
Dequeue(Create) = error;
Front(Create) = error;
Dequeue(Enqueue(q,i)) = if (IsEmpty(q))
Semantic Specification of Queue

for all [q: Queue; i: Item]

IsEmpty(Create) = true;
IsEmpty(Enqueue(q,i)) = false;
Dequeue(Create) = error;
Front(Create) = error;
Dequeue(Enqueue(q,i)) = if (IsEmpty(q))
then Create

end QueueOfItem;
Semantic Specification of Queue

for all \([q: \text{Queue}; \; i: \text{Item}]\)
\[\text{IsEmpty}(\text{Create}) = \text{true};\]
\[\text{IsEmpty}(\text{Enqueue}(q,i)) = \text{false};\]
\[\text{Dequeue}(\text{Create}) = \text{error};\]
\[\text{Front}(\text{Create}) = \text{error};\]
\[\text{Dequeue}(\text{Enqueue}(q,i)) = \text{if (IsEmpty(q))}\]
\[\quad \text{then Create}\]
\[\quad \text{else Enqueue(Dequeue(q),i)};\]

end QueueOfItem;
Semantic Specification of Queue

for all [q: Queue; i: Item]
 IsEmpty(Create) = true;
IsEmpty(Enqueue(q,i)) = false;
Dequeue(Create) = error;
Front(Create) = error;
Dequeue(Enqueue(q,i)) = if (IsEmpty(q))
  then Create
  else Enqueue(Dequeue(q),i);

Front(Enqueue(q,i))

end QueueOfItem;
for all [q: Queue; i: Item]
IsEmpty(Create) = true;
IsEmpty(Enqueue(q,i)) = false;
Dequeue(Create) = error;
Front(Create) = error;
Dequeue(Enqueue(q,i)) = if (IsEmpty(q))
    then Create
    else Enqueue(Dequeue(q),i);
Front(Enqueue(q,i)) = if (IsEmpty(q))
end QueueOfItem;
Semantic Specification of Queue

for all [q: Queue; i: Item]
IsEmpty(Create) = true;
IsEmpty(Enqueue(q,i)) = false;
Dequeue(Create) = error;
Front(Create) = error;
Dequeue(Enqueue(q,i)) = if (IsEmpty(q))
then Create
else Enqueue(Dequeue(q),i);
Front(Enqueue(q,i)) = if (IsEmpty(q))
then i
end QueueOfItem;
Semantic Specification of Queue

for all [q: Queue; i: Item]
IsEmpty(Create) = true;
IsEmpty(Enqueue(q,i)) = false;
Dequeue(Create) = error;
Front(Create) = error;
Dequeue(Enqueue(q,i)) = if (IsEmpty(q))
    then Create
    else Enqueue(Dequeue(q),i);
Front(Enqueue(q,i)) = if (IsEmpty(q))
    then i
    else Front(q);
end QueueOfItem;
First: Queue Generators

- Create and Enqueue(q, i) are generators
- The following are valid queues
  - Create
  - Enqueue(Create, 1)
  - Enqueue(Enqueue(Create, 1), 2)
  - Enqueue(Enqueue(Enqueue(Create, 1), 2), 3)

- IsEmpty operator is easy to understand
  - Create is Empty, anything else is not
Second: What about Front?

- **Rule:** \( \text{Front}(\text{Enqueue}(q,i)) = \begin{cases} i & \text{if (IsEmpty(q))} \\ \text{Front}(q) & \text{else} \end{cases} \)

- \( \text{Front}(\text{Enqueue}(\text{Enqueue}(\text{Create}, 1), 2)) \)
  - \( q \) is highlighted in bold; it's not empty, so
  - \( \text{Front}(\text{Enqueue}(\text{Create}, 1)) \)
    - \( q \) is again highlighted in bold; it IS empty, so
    - 1

- And that indeed is the front of the original queue
Third: What about Dequeue?

- **Rule:** Dequeue(Enqueue(q,i)) = if (IsEmpty(q))
  then Create
  else Enqueue(Dequeue(q),i)

- Dequeue(Enqueue(Enqueue(Create, 1), 2))
- Enqueue(Dequeue(Enqueue(Create, 1)), 2)
- Enqueue(Create, 2)

- We are left with a queue in which the first element was indeed removed
Textbook Example: Library

- Important to realize that example in book is incomplete
- Operators are:
  - New, buy, lose, borrow, return, reserve, unreserve, recall, isInCatalogue, isOnLoan, isOnReserve
- Generators: New, buy, borrow, reserve
- Queries: isInCatalogue, isOnLoan, isOnReserve
- Manipulators: lose, return, unreserve, recall
Example Libraries

- New
- buy(New, a)
- buy(buy(New, a), b)
- borrow(buy(buy(New, a), b), b)
- reserve(borrow(buy(buy(New, a), b), b), a)

- Last library has two books “a” and “b”
  - a is on reserve, b has been borrowed
Example: IsInCatalogue (I)

- Rules
  - $\text{isInCatalogue}(\text{New}, i) \equiv \text{ERROR}$
  - $\text{isInCatalogue}(\text{buy}(\text{lib}, i), i_2) \equiv$
    - if $i = i_2$ then true else $\text{isInCatalogue}(\text{lib}, i_2)$
  - $\text{isInCatalogue}(\text{borrow}(\text{lib}, i), i_2) \equiv$
    - $\text{isInCatalogue}(\text{lib}, i_2)$
  - $\text{isInCatalogue}(\text{reserve}(\text{lib}, i), i_2) \equiv$
    - $\text{isInCatalogue}(\text{lib}, i_2)$

- We must supply definitions for each non-generator being applied to instances of each generator
Example: IsInCatalogue (II)

- IsInCatalogue(borrow(buy(buy(New, a), b), b), a)
- IsInCatalogue(buy(buy(New, a), b), a)
- IsInCatalogue(buy(New, a), a)
- True
- IsInCatalogue(borrow(buy(buy(New, a), b), b), c)
- IsInCatalogue(buy(buy(New, a), b), c)
- IsInCatalogue(buy(New, a), c)
- IsInCatalogue(New, c)
- False
Example: Lose (I)

- **Rules**
  - \(\text{lose}(\text{New}, i) \equiv \text{ERROR}\)
  - \(\text{lose}(\text{buy}(\text{lib}, i), i2) \equiv\)
    - if \(i = i2\) then lib else buy(lose(lib, i2), i)
  - \(\text{lose}(\text{borrow}(\text{lib}, i), i2) \equiv\)
    - if \(i = i2\) then lose(lib, i2) else borrow(lose(lib, i2), i)
  - \(\text{lose}(\text{reserve}(\text{lib}, i), i2) \equiv\)
    - if \(i = i2\) then lose(lib, i2) else reserve(lose(lib, i2), i)
Example: Lose (II)

- lose(reserve(borrow(buy(buy(New, a), b), b), a), a)
- lose(borrow(buy(buy(New, a), b), b), a)
- borrow(lose(buy(buy(New, a), b), a), b)
- borrow(buy(lose(buy(New, a), a), b), b)
- borrow(buy(New, b), b)

- In moving to the last step, the entire phrase
  - lose(buy(New, a), a)
- was simply replaced with
  - New
Summary

- Algebraic specifications model the behavior of a system via operations on structured strings that capture the system’s state
  - Other notations can “tempt” developers into specifying the implementation of a system early
  - That is, other notations tend to suggest particular implementations
    - UML class model ⇒ Classes in OO language
    - Data Flow Diagrams ⇒ Data Processing Modules
    - Z specification ⇒ sets, sequences, and functions
- Algebraic specs can reduce this temptation since their suggested implementation is so inefficient!