Abstract Specifications: A Review

Computer Science 520/620
Prof. Leon J. Osterweil
Spring 2012

Algebraic Specification

• Draws upon the semantics of modern algebra to form the basis of the semantics of data types

• Define a type as being the elements of an algebra

• Define the type in terms of how its functions interact with each other

• Consists of two parts:
  --Function list
  --Function interrelations:

• Function list: function templates

• Function interrelations: how the functions interact with each other
Algebraic Specification of a Stack

FUNCTION LIST:

CREATE: \rightarrow STACK
PUSH: STACK \times INTEGER \rightarrow STACK
TOP: STACK \rightarrow \{ INTEGER \cup INTERR \}
POP: STACK \rightarrow \{ STACK \cup STACKERR \}

RELATION LIST:
Algebraic Specification of a Stack

FUNCTION LIST:

CREATE: \rightarrow STACK

PUSH: STACK \times INTEGER \rightarrow STACK
Algebraic Specification of a Stack

FUNCTION LIST:

CREATE: \( \rightarrow \text{STACK} \)

PUSH: STACK \( \times \) INTEGER \( \rightarrow \text{STACK} \)

TOP: STACK \( \rightarrow \) \( \{ \text{INTEGER} \cup \text{INTERR} \} \)

POP: STACK \( \rightarrow \) \( \{ \text{STACK} \cup \text{STACKERR} \} \)

RELATION LIST:

TOP (PUSH(s,i)) = i

TOP (CREATE) = INTERR
Algebraic Specification of a Stack

**FUNCTION LIST:**
- **CREATE:** \( \rightarrow \text{STACK} \)
- **PUSH:** \( \text{STACK} \times \text{INTEGER} \rightarrow \text{STACK} \)
- **TOP:** \( \text{STACK} \rightarrow \{ \text{INTEGER} \cup \text{INTERR} \} \)
- **POP:** \( \text{STACK} \rightarrow \{ \text{STACK} \cup \text{STACKERR} \} \)

**RELATION LIST:**
- \( \text{TOP}(\text{PUSH}(s,i)) = i \)
- \( \text{TOP}(\text{CREATE}) = \text{INTERR} \)
- \( \text{POP}(\text{PUSH}(s,i)) = s \)
- \( \text{POP}(\text{CREATE}) = \text{STACKERR} \)

---

A Different Stack

**FUNCTION LIST:**
- **CREATE:** \( \rightarrow \text{STACK} \)
- **PUSH:** \( \text{STACK} \times \text{INTEGER} \rightarrow \text{STACK} \)
- **TOP:** \( \text{STACK} \rightarrow \{ \text{INTEGER} \cup \text{INTERR} \} \)
- **POP:** \( \text{STACK} \rightarrow \{ \text{STACK} \cup \text{STACKERR} \} \)
- **ISEMPTY:** \( \text{STACK} \rightarrow \{ \text{TRUE}, \text{FALSE} \} \)

**RELATION LIST:**
- \( \text{TOP}(\text{PUSH}(s,i)) = i \)
- \( \text{TOP}(\text{CREATE}) = \text{INTERR} \)
- \( \text{POP}(\text{PUSH}(s,i)) = s \)
- \( \text{POP}(\text{CREATE}) = \text{STACKERR} \)
- \( \text{ISEMPTY}(\text{CREATE}) = \text{TRUE}, \text{ELSE} = \text{FALSE} \)
Still Another Stack

FUNCTION LIST:

CREATE: \rightarrow \text{STACK}

PUSH: \text{STACK} \times \text{INTEGER} \rightarrow \text{STACK}

TOP: \text{STACK} \rightarrow \{ \text{INTEGER} \cup \text{INTERR} \}

POP: \text{STACK} \rightarrow \{ \text{STACK} \cup \text{STACKERR} \}

LENGTH: \text{STACK} \rightarrow \text{INTEGER}

RELATION LIST:

\text{TOP (PUSH}(s,i) \text{) } = i

\text{TOP (CREATE) } = \text{INTERR}

\text{POP (PUSH } (s,i) \text{) } = s

\text{POP (CREATE) } = \text{STACKERR}

\text{LENGTH (STACK) } = \ldots \ldots \ldots

What is this good for?

- Focus on the computational aspects of this type
- Provides simplification rules
  - E.g. defines inverses
  - Helps support reasoning about execution traces involving stacks
Example

Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
    if ( top (stack) = sentinel) then exit;
    sum := sum + top (stack);
    pop (stack);
    I := read ( );
push (stack, I);
end Do;
Print (sum);

Example

Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
    if ( top (stack) = sentinel) then exit;
    sum := sum + top (stack);
    pop (stack);
    I := read ( );
push (stack, I);
end Do;
Print (sum);

Is stack empty when Print Statement is encountered?
Example

Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
    if ( top (stack) = sentinel) then exit;
    sum := sum + top (stack);
    pop (stack);
    I := read ( );
push (stack, I);
end Do;
Print (sum);

Example: push(pop(push(push(create, I)),I),I)

Is stack empty when Print Statement is encountered?

Example

Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
    if ( top (stack) = sentinel) then exit;
    sum := sum + top (stack);
    pop (stack);
    I := read ( );
push (stack, I);
end Do;
Print (sum);

Example: push(pop(push(push(create, I)),I),I)

Is stack empty when Print Statement is encountered?
Example

Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
  if ( top (stack) = sentinel) then exit;
  sum := sum + top (stack);
  pop (stack);
  I := read ( );
push (stack, I);
end Do;
Print (sum);

Example: push(pop(push(create, I)),I)

Example

Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
  if ( top (stack) = sentinel) then exit;
  sum := sum + top (stack);
  pop (stack);
  I := read ( );
push (stack, I);
end Do;
Print (sum);

Example: push(pop(push(create, I)),I)
Create stack;
sum := 0;
I := read ( );
push (stack, I);
Do forever
    if ( top (stack) = sentinel) then exit;
    sum := sum + top (stack);
    pop (stack);
    I := read ( );
    push (stack, I);
end Do;
Print (sum);

Is stack empty when Print Statement is encountered?

Example: push(create, I)

Create stack;
Sum := 0;
I := read ( );
push (stack, I);
Do 10 times
    if ( top (stack) = sentinel) then exit;
    sum := sum + top (stack);
    I := read ( );
    push (stack, I);
end Do;
Print (sum);

Length of stack now?
Axiomatic Set Theory

ADT Specification

- Semantics of the data type derived from semantics of axiomatic set theory
- Supports rigorous reasoning about design and code
- Describes data type in terms of its abstract behaviors
- Describes accessing functions in terms of relations to each other
- Hard to write correct specifications (and know it)
- Reading them is hard too

Axiomatic Specification of a Stack

STACK(CREATE)
Axiomatic Specification of a Stack

STACK(CREATE)

It is true that you can create instances of objects of type "STACK".

Axiomatic Specification of a Stack

STACK(s) \land INTEGER(i) \implies \text{PUSH}(s,i) \neq \text{CREATE}
Axiomatic Specification of a Stack

STACK(s) \land INTEGER(i) \implies \text{PUSH}(s,i) \neq \text{CREATE}

Push makes a non-empty stack

Axiomatic Specification of Stack

STACK(s) \land STACK(s') \land INTEGER(i)

\implies [\text{PUSH}(s,i) = \text{PUSH}(s',i) \implies (s = s')]
Axiomatic Specification of Stack

\[ \text{STACK}(s) \land \text{STACK}(s') \land \text{INTEGER}(i) \Rightarrow [ (\text{PUSH}(s,i) = \text{PUSH}(s',i) ) \Rightarrow (s = s') ] \]

**Pushing doesn’t change what was already there**

Axiomatic Specification of Stack

\[ \text{STACK}(s) \land \text{INTEGER}(i) \Rightarrow \text{TOP} \left( \text{PUSH}(s,i) \right) = i \]
Axiomatic Specification of Stack

\[ \text{STACK}(s) \land \text{INTEGER}(i) \Rightarrow \text{TOP}(\text{PUSH}(s,i)) = i \]

Last in, first out

Axiomatic Specification of Stack

\[ \text{STACK}(s) \land \text{INTEGER}(i) \Rightarrow \text{POP}(\text{PUSH}(s,i)) = s \]
Axiomatic Specification of Stack

\[ \text{STACK}(s) \land \text{INTEGER}(i) \Rightarrow \text{POP}(\text{PUSH}(s,i)) = s \]

Pop and Push are inverses of each other

Axiomatic Specification of Stack

\[ \text{TOP} (\text{CREATE}) = \text{INTEGERERROR} \]
\[ \text{POP} (\text{CREATE}) = \text{STACKERROR} \]
**Axiomatic Specification of Stack**

\[
\begin{align*}
\text{TOP(CREATE)} &= \text{INTEGERERROR} \\
\text{POP(CREATE)} &= \text{STACKERROR}
\end{align*}
\]

Expected behavior of empty stack

**Axiomatic Specification of a Stack**

\[
\begin{align*}
\text{STACK}(s) \land \text{INTEGER}(i) & \implies \\
\text{STACK}(\text{PUSH}(s,i)) & \land \\
[\text{POP}(s) \neq \text{STACKERROR} & \implies \text{STACK(POP}(s)) ] & \land \\
[\text{TOP}(s) \neq \text{INTEGERERROR} & \implies \text{INTEGER(TOP}(s)) ]
\end{align*}
\]
Axiomatic Specification of a Stack

\[
\text{STACK}(s) \land \text{INTEGER}(i) \Rightarrow \\
\text{STACK}(\text{PUSH}(s,i)) \land \\
[\text{POP}(s) \neq \text{STACKERROR} \Rightarrow \text{STACK}(\text{POP}(s)) ] \land \\
[\text{TOP}(s) \neq \text{INTEGERERROR} \Rightarrow \text{INTEGER}(\text{TOP}(s)) ]
\]

Defining the range space of Push, Pop, and Top

Axiomatic Specification of a Stack

\[
\forall A [ \ A(\text{CREATE}) \land (\forall s)(\forall i) \\
\quad [\text{STACK}(s) \land \text{INTEGER}(i) \land A(s) \\
\quad \Rightarrow A(\text{PUSH}(s,i)) \land [s \neq \text{CREATE} \Rightarrow A(\text{POP}(s))] ] \\
\quad \Rightarrow \forall s [\text{STACK}(s) \Rightarrow A(s) ]
\]
Axiomatic Specification of a Stack

∀ A [ A(CREATE) ∧ (∀ s)(∀ i) 

[STACK(s) ∧ INTEGER(i) ∧ A(s) 

⇒ A(PUSH(s,i)) ∧ [s ≠ CREATE ⇒ A(POP(s)) ] ] 

⇒ ∀ s [STACK(s) ⇒ A(s) ]

Full Axiomatic Specification of a Stack

STACK(CREATE) 

∧ STACK(s) ∧ INTEGER(i) ⇒ PUSH(s,i) ≠ CREATE 

∧ STACK(s) ∧ STACK(s') ∧ INTEGER(i) 

⇒ [ PUSH(s,i) = PUSH(s', i) ⇒ (s = s') ] 

∧ STACK(s) ∧ INTEGER(i) ⇒ TOP(PUSH(s,i)) = i 

∧ STACK(s) ∧ INTEGER(i) ⇒ POP(PUSH(s,i)) = s 

∧ TOP(CREATE) = INTEGERERROR 

∧ POP(CREATE) = STACKERROR 

∧ STACK(s) ∧ INTEGER(i) ⇒ 

STACK(PUSH(s,i)) ∧ 

[POP(s) ≠ STACKERROR ⇒ STACK(POP(s)) ] ∧ 

[TOP(s) ≠ INTEGERERROR ⇒ INTEGER(TOP(s)) ] 

∧ ∀ A [ A(CREATE) ∧ (∀ s)(∀ i) 

[STACK(s) ∧ INTEGER(i) ∧ A(s) 

⇒ A(PUSH(s,i)) ∧ [s ≠ CREATE ⇒ A(POP(s)) ] ] 

⇒ ∀ s [STACK(s) ⇒ A(s) ]

All stacks act like stacks
What is this good for?

- Providing rigorous specifications about types to be built
- Providing rigorous specifications for modules to be built
- Supports logical reasoning about software using this type
- Proving rigorous theorems about software
- We will see more about this soon

Other Formal Approaches

- System structure: Its modules, their relations,
  - Z (pronounced “zed”)
  - Larch
  - VDL
- Concurrency structure
  - CSP (Cooperating Sequential Processes)
  - TSL (Task Sequence Language)
Z (Pronounced “Zed”)

- Developed at Oxford by Hoare, Spivey, etc.
- Represents WHAT software systems do without specifying HOW
- Uses set theory and function notation
- Describe systems as collections of SCHEMAS
  - inputs and outputs to functions
  - Invariants: statements whose truth is preserved by the functions
- Lots of idiosyncratic notation
- Intent is to make specifications brief, yet clear and precise

The “Birthday Book” Example

- Maintain a repository of information about birthdays
- Consists of (name, birthday) pairs
- Want to add pairs for people whose birthdays are to be remembered
- Want to know whose birthday falls on a given date
- Don’t care about how this is implemented
Example Schema

This schema describes the STATE SPACE of the system: the space of all states that the system can be in.
Example Schema

Name of the schema

"Set of" symbol: This line means
known is a set of elements of type NAME

BirthdayBook

known : \( \mathcal{P} \) NAME

birthday : NAME \( \rightarrow \) DATE

known = dom birthday

This schema describes the STATE SPACE of the system: the space of all states that the system can be in
Example Schema

Name of the schema

“Set of” symbol: This line means known is a set of elements of type NAME

BirthdayBook

known : \( \mathbb{P} \) NAME

birthday : NAME \( \rightarrow \) DATE

known = dom birthday

The invariant part of the schema:

This line means The set known is the domain of definition of the function birthday

“function” symbol used to denote a function from a set to a set A function from an element to an element is denoted by

This line means birthday is a function defined on a set of NAME and mapping into a set DATE

This schema describes the STATE SPACE of the system: the space of all states that the system can be in

Another Schema

This Delta symbol indicates that this schema will describe a state change

AddBirthday

\( \Delta \) BirthdayBook

name?  :  NAME
date?  :  DATE

\( \text{name?} \notin \text{known} \quad \text{birthday}' = \text{birthday} \cup \{\text{name?} \rightarrow \text{date?}\} \)
Another Schema

This Delta symbol indicates that this schema will describe a state change

AddBirthday

BirthdayBook

name? : NAME
date? : DATE

When the schema describes a change of state then it is necessary to distinguish between the value in an element before the state change and the value after. The ‘ denotes the value after the state change.
Two more Schemas

**FindBirthday**

Ξ BirthdayBook

name? : NAME
date! : DATE

name? e known
date! = birthday (name?)

**Remind**

Ξ BirthdayBook

today? : DATE

cards! : (NAME

cards! = { n : known | birthday(n) = today? }

Indicates there will be no change in schema state
Two more Schemas

FindBirthday

BirthdayBook

\[ \text{name? : NAME} \]
\[ \text{date! : DATE} \]

\[ \text{name? \in \text{known}} \]
\[ \text{date! = birthday(name?)} \]

Remind

BirthdayBook

\[ \text{today? : DATE} \]
\[ \text{cards! : \{ NAME} \]

\[ \text{cards! = \{ n : \text{known} \mid \text{birthday(n) = today?} \}} \]

Indicates there will be no change in schema state

Indicates an output value

This denotes the set of all elements n, drawn from the set known, such that \text{birthday(n) = today?}
Z Summary

• Schemas can be grouped and composed
• More notation: aimed at facilitating terse, precise communication
• Emphasis on what a system is supposed to do
• Indication of how it looks externally
• (Like Abstract Data Type specifications) basis for going on to think about HOW to implement