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: → STACK
PUSH: STACK X INTEGER → STACK
TOP: STACK → (INTEGER ∪ INTERR)
POP: STACK → (STACK ∪ STACKERR)

RELATION LIST:

TOP (PUSH(s,i)) = i
TOP (CREATE) = INTERR
POP (PUSH(s,i)) = s
POP (CREATE) = STACKERR

A Different Stack

FUNCTION LIST:

CREATE: → STACK
PUSH: STACK X INTEGER → STACK
TOP: STACK → (INTEGER ∪ INTERR)
POP: STACK → (STACK ∪ STACKERR)
ISEMPTY: STACK → {TRUE, FALSE}

RELATION LIST:

TOP (PUSH(s,i)) = i
TOP (CREATE) = INTERR
POP (PUSH(s,i)) = s
POP (CREATE) = STACKERR
ISEMPTY (CREATE) = TRUE, ELSE = FALSE

Still Another Stack

FUNCTION LIST:

CREATE: → STACK
PUSH: STACK X INTEGER → STACK
TOP: STACK → (INTEGER ∪ INTERR)
POP: STACK → (STACK ∪ STACKERR)
LENGTH: STACK → INTEGER

RELATION LIST:

TOP (PUSH(s,i)) = i
TOP (CREATE) = INTERR
POP (PUSH(s,i)) = s
POP (CREATE) = STACKERR
LENGTH (STACK) = ..........
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: push(pop(push(pop(push(create, I)),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);

Is stack empty when Print Statement is encountered?

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);

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);

Is stack empty when Print Statement is encountered?

Example: push(create, I)

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)

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

STACK(s) ∧ INTEGER(i) ⇒ PUSH(s,i) ≠ CREATE
Axiomatic Specification of a Stack

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

Push makes a non-empty stack

Axiomatic Specification of Stack

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

Pushing doesn’t change what was already there

Axiomatic Specification of Stack

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

Last in, first out

Axiomatic Specification of Stack

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

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

\text{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 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

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

∀ A \[ A(CREATE) \land (\forall s)(\forall i) \]

\[ [STACK(s) \land INTEGER(i) \land A(s)] \land
A(PUSH(s,i)) \land [s \neq CREATE \implies A(POP(s))] \]

⇒ \forall s [STACK(s) \implies A(s)]

Full Axiomatic Specification of a Stack

STACK(CREATE) \land

⇒ STACK(s) \land INTEGER(i) \implies PUSH(s,i) = CREATE

⇒ STACK(s) \land STACK(s') \land INTEGER(i) \implies PUSH(s,i) = PUSH(s',i)

⇒ STACK(s) \land INTEGER(i) \implies TOP(PUSH(s,i)) = i

⇒ STACK(s) \land INTEGER(i) \implies POP(PUSH(s,i)) = s

⇒ TOP(CREATE) = INTEGERERROR

⇒ POP(CREATE) = STACKERROR

⇒ STACK(POP(s)) = STACKERROR

⇒ POP(PUSH(s,i)) = s

⇒ TOP(POP(s)) = INTEGERERROR

⇒ \forall A \[ A(CREATE) \land (\forall s)(\forall i) \]

⇒ STACK(s) \land INTEGER(i) \implies A(s) \land A(PUSH(s,i)) \land

⇒ \forall s [STACK(s) \implies A(s)]

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

BirthdayBook

known : \{ \text{NAME} \}

birthday : \text{NAME} \rightarrow \text{DATE}

known = \text{dom birthday}

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

Another Schema

AddBirthday

\[ \Delta \text{BirthdayBook} \]

name? : \text{NAME}

date? : \text{DATE}

name? \notin \text{known}

birthday = \text{birthday} \cup \{ \text{name?}, \text{date?} \}

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

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

AddBirthday

BirthdayBook

name? : NAME
date? : DATE

Another Schema

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

AddBirthday

BirthdayBook

name? : NAME
date? : DATE

name? ∈ known

birthday = birthday' ∪ (name? ∪ 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 state change. The ' denotes the value after the state change.

Two more Schemas

FindBirthday

BirthdayBook

name? : NAME
date! : DATE

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

Remind

BirthdayBook

today? : DATE
cards! : NAME

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

Indicates there will be no change in schema state
Indicates an output value

Two more Schemas

FindBirthday

BirthdayBook

name? : NAME
date! : DATE

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

Remind

BirthdayBook

today? : DATE
cards! : NAME

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

Two more Schemas

FindBirthday

BirthdayBook

name? : NAME
date! : DATE

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

Remind

BirthdayBook

today? : DATE
cards! : NAME

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

This denotes the set of all elements n, drawn from the set known, such that 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