Computer Science 520/620  
Spring 2012  
Prof. L. Osterweil

Software Models and Representations  
Part 4

Specification without Graphs and Diagrams

- The graphs are pictures of relations
- The semantics are provided by the relations
- What do we need the graphs for?
- Clarity
- Communication with non-CS people
- Do we need the graphs for communication with CS types?
- Might be more precise, concise
- Not everyone is “visual”

Use of Formalism Alone

- Main advantages are precision and rigor
- Semantic breadth of scope is possible.
  - Some things are hard to draw pictures of
- Main drawback is lack of clarity
  - At least for non-technical people

Formal Languages Approach

- Represent (model) specific aspects of software system using established formalisms (eg. set theory, logic, algebra) to provide the semantics
- Focus on some aspect(s), ignore the others
  - To provide clear understanding of that aspect
  - Avoid clutter
  - Provide rigor and detail about modeled aspect(s)
- We have already seen example(s) of this:
  - Use of finite mathematics, logic, graph theory to provide semantics for diagrams

Advantages of Formal Language

- Diagrams support clarity, good for customers, ??
  - Pictures support intuitive reasoning
  - Help identify gaps, shortcomings, weaknesses
  - Suggest truths, theorems, facts
  - But are generally based upon very weak semantics
    - Lack breadth of semantics
    - Often lack precision and detail
- Formal Languages, good for developers, ???
  - Strength is precision and rigor
  - Broad semantics are possible
  - Often feature considerable detail (that may interfere with clarity)

Defining Data Semantics with Formal Languages

- Focus on defining data types
  - Data type is a set of data instances all having some common characteristics and properties
  - Define the set, and characteristics
- User’s (client’s)-eye view of the data types to be used
- Describe the “accessing primitives” / “operators”, “methods”, functions providing the only mechanisms for manipulating instances of a given type
  - Dual notion to describing functions in terms of their data inputs and outputs
- Goal: Specify the types without specifying their implementation
- Being rigorous helps separate (even slightly) different notions of a data type from each other
Abstract Data Type Definition Approaches

- Natural language
- Diagrams
- Finite State Machines
- Axiomatic Set Theory
- Algebras


Algebraic Specification

- Draws upon the semantics of 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:

Function List:

CREATE: \rightarrow STACK
PUSH: STACK \times INTEGER \rightarrow STACK
TOP: STACK \rightarrow \{ INTEGER \cup INTERR \}
POP: STACK \rightarrow \{ STACK \cup STACKERR \}
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:
- TOP (PUSH(s,i)) = i
- TOP (CREATE) = INTERR
- POP (PUSH(s,i)) = s
- POP (CREATE) = STACKERR

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

Yet Another Stack

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

RELATION LIST:
- TOP (PUSH(s,i)) = i
- TOP (CREATE) = INTERR
- POP (PUSH(s,i)) = s
- POP (CREATE) = STACKERR
- LENGTH (STACK) = ……….

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);

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);

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

Is stack empty when Print Statement is encountered?

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

Is stack empty when Print Statement is encountered?

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

Is stack empty when Print Statement is encountered?

Example: 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(create, I)

Example
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?

Length of stack now?

Previous definition doesn’t address some characteristics
Other formalisms make it easier to specify other characteristics

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”
Axiomatic Specification of a Stack

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

Push makes a non-empty stack

Axiomatic Specification of Stack

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

Pushing doesn’t change what was already there (not covered by algebraic specification)

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

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

Pop and Push are inverses of each other

Axiomatic Specification of Stack

\[ \text{TOP} (\text{CREATE}) = \text{INTEGERERROR} \]
\[ \text{POP} (\text{CREATE}) = \text{STACKERROR} \]

Expected behavior of empty stack

Axiomatic Specification of a Stack

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

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

\[ \begin{align*}
& \text{STACK}(s) \land \text{INTEGER}(i) \land A(s) \\
& \Rightarrow A(PUSH(s,i)) \land [s \neq CREATE \Rightarrow A(POP(s))] \\
& \Rightarrow \forall s [\text{STACK}(s) \Rightarrow A(s)] 
\end{align*} \]

Full Axiomatic Specification of a Stack

\[ \begin{align*}
& \text{STACK}(CREATE) \\
& \land \text{STACK}(s) \land \text{INTEGER}(i) \Rightarrow \text{PUSH}(s,i) = \text{CREATE} \\
& \land \text{STACK}(s) \land \text{STACK}(s') \land \text{INTEGER}(i) \\
& \Rightarrow [\text{PUSH}(s,i) = \text{PUSH}(s',i) \Rightarrow (s = s')] \\
& \land \text{STACK}(s) \land \text{INTEGER}(i) \Rightarrow \text{TOP}(\text{PUSH}(s,i)) = i \\
& \land \text{TOP}(\text{CREATE}) = \text{INTEGERERROR} \\
& \land \text{POP}(\text{CREATE}) = \text{STACKERROR} \\
& \land \text{STACK}(s) \land \text{INTEGER}(i) \Rightarrow \\
& \text{STACK}(\text{PUSH}(s,i)) \land \\
& [\text{POP}(s) = \text{STACKERROR} \Rightarrow \text{STACK}(\text{POP}(s))] \land \\
& [\text{TOP}(s) = \text{INTEGERERROR} \Rightarrow \text{INTEGER}(\text{TOP}(s))] \\
& \land \forall A \[ A(\text{CREATE}) \land (\forall s)(\forall i) \]
\[ \begin{align*}
& \text{STACK}(s) \land \text{INTEGER}(i) \land A(s) \\
& \Rightarrow A(PUSH(s,i)) \land [s \neq CREATE \Rightarrow A(POP(s))] \\
& \Rightarrow \forall s [\text{STACK}(s) \Rightarrow A(s)] 
\end{align*} \]

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

<table>
<thead>
<tr>
<th>Name of the schema</th>
<th>known</th>
<th>P</th>
<th>NAME</th>
<th>birthday</th>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BirthdayBook</td>
<td>known</td>
<td>$\in$</td>
<td>NAME</td>
<td>$\rightarrow$</td>
<td>DATE</td>
</tr>
<tr>
<td>known = dom birthday</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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? }

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

AddBirthday

BirthdayBook

name? : NAME

date? : DATE

name? ∈ known

birthday' = birthday ∪ {name? , date? }

The ? after these symbols indicates that the element will be an input.

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

FindBirthday

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.

This shows there will be no change in the schema state.

Remind

BirthdayBook

today? : DATE

cards! : NAME

cards! = { n ∈ known | birthday(n) = today? }
Two more Schemas

\[ \text{BirthdayBook} \]
\[ \text{name? : NAME} \]
\[ \text{date! : DATE} \]
\[ \text{FindBirthday (name?) = \text{birthday} (name?)} \]
\[ \text{Remind} \]
\[ \text{BirthdayBook} \]
\[ \text{today? : DATE} \]
\[ \text{cards! : \{ NAME \}} \]
\[ \text{cards! = \{ n : \text{known} \mid \text{birthday}(n) = \text{today}? \}} \]

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

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