Voldemort Type

Graphical user interface, text, application

Description automatically generated

Text, letter

Description automatically generated

**ABSTRACT DATA TYPES**

Data Types

Predefined

Type constructors: build new data types

How to provide “queue”?

* What should be the data values?
* What should be the operations?
* How to implement (data representation, operations)?

Queue

A common abstract data type is a queue.

Diagram

Description automatically generated

A queue is a first in, first out (FIFO) structure or in the other sense, a last in, last out (LILO) structure. A queue is sometimes generalized as a structure where insertion (enqueue) occur at one end and removal (dequeue) occurs at the other end.

What are inadequate here?

* The operations are not associated with the data type
  + You can use the operation on an invalid value
* Users see all the details: direct access to data elements, implementations
  + Implementation dependent
  + Users can even mess up with things

What do we want?

* For basic types:
  + 4 bytes or 2 bytes, users don’t need to know.
  + Can only use predefined operations.
* Similarly, for the “Queue” data type:
  + ?

Abstract Data Types

Built-in types have important properties that "abstract" away the implementation: use of int and its operations (+, \*, etc.) normally do not require knowledge of bit patterns (2's complement? 4 bytes?)

User-defined types do not in general have this property: internal structure is visible to all code

Use of internal structure makes it difficult to change later

Operations on data (except the most basic) not specified and often hard to find

A mechanism of a programming language designed to imitate the abstract properties of a built-in type as much as possible

Must include a specification of the operations that can be applied to the data

Must hide the implementation details from client code

These properties are sometimes called encapsulation & information hiding (with different emphases)

Encapsulation:

* All definitions of allowed operations for a data type in one place.

Information Hiding:

* Separation of implementation details from definitions. Hide the details .

Algebraic Specification of ADT

Syntactic specification (signature, interface) – (user’s perspective):

* the name of the type, the prototype of the operations
* concrete syntax

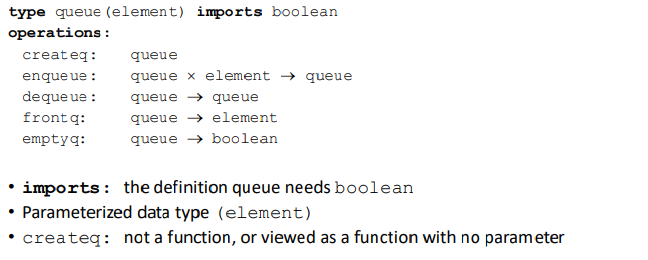
Semantic specification (axioms, implementation):

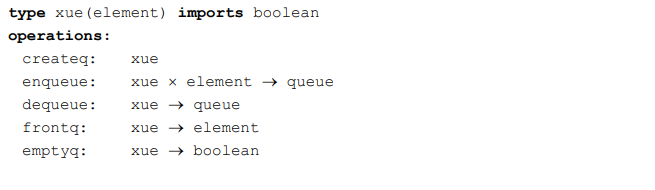
* what is this doing?
* guide for required properties in implementation
* mathematical properties of the operations
* abstract syntax

They don’t specify:

* data representation
* implementation details

Syntactic Specification





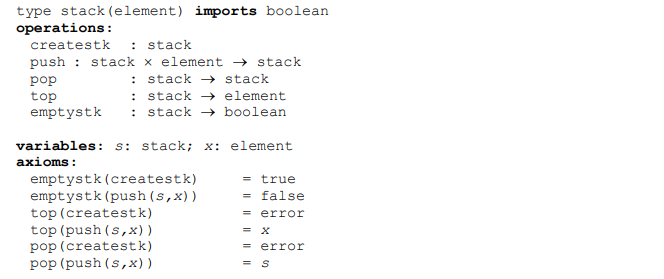
Algebraic Specification

Graphical user interface, text, application

Description automatically generated

axiom: fact, definition

Stack



Are these axioms enough?

* emptystk
  + we control both for empty and non-empty stack so we are done
* createstk
  + we control created stack will be empty via top and emptystk
* top-pop
  + we control these for both empty and non-empty stack so we are done
* push
  + it is simple

So axioms are sufficient but we can write some other things:

* top(pop(push(push(s,x), y)))
  + y is another element variable
  + this should give me x

Let’s say I want to change something, I want to know how many entries are there in the stack.

* New adt 🡪 StackCount
  + This will have all the operations above and also count operation that gives me number

Syntax:

* count : stack 🡪 int
  + I have to add int next to boolean as an import 🡪 … imports boolean, int

Semantic:

* count(createstk) = 0
* count(push(s, x)) = count(s) + 1

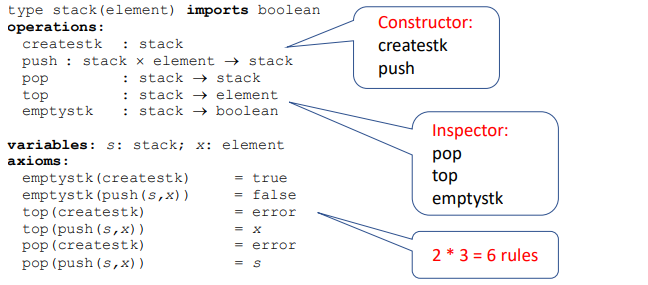
We have controlled for both if stack is empty and if stack is empty. We know “push(s, x)” is not empty because for the second emptystk axiom.

You can use axioms as test codes. They are very simple to convert to test code. This is test-driven programming.

Axioms:

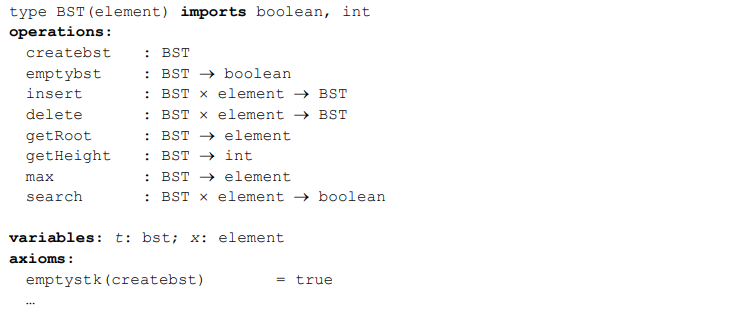
* How many axioms are sufficient for proving all necessary properties?

Some Heuristics



You can have more than sufficient number of axioms but you shouldn’t have less than that.

Binary Search Tree





getRoot(insert(createbst, x)) = x

…

Other Examples of ADT

* Stack
* Queue
* Tree
* Set
* Map
* Vector
* List
* Priority Queue
* Graph
* …

Algebraic Specification Notes

Specifications are usually written in functional form with no side effects or assignment. So no "void" functions

Specifications are often simplified to make axiom writing easier

* E.g., in the stack example, pop does not return the top, only the (previously created) stack below the current top.
* We could have written pop as “pop: stack 🡪 element x stack”, but the axioms are more complex

ADT Language Mechanisms

Most languages do not have a specific ADT mechanism – instead they have a more general module mechanism

Specific ADT mechanisms

* ML abstype – but newer module mechanism is more useful…

General module mechanism: not just about a single data type and its operations

* Separate compilation and name control: C, C++, Java
* Ada, ML

Class in OO languages (which has many of the properties needed by an ADT mechanism)

Modules

Module: A program unit with a public interface and a private implementation; all services that are available from a module are described in its public interface and are exported to other modules, and all services that are needed by a module must be imported from other modules.

A module offers general services, which may include types and operations on those types, but are not restricted to these.

Modules have nice properties:

* A module can be (re)used in any way that its public interface allows.
* A module implementation can change without affecting the behavior of other modules.

In addition to ADT, module supports structuring of large programs: Separate compilation and name control

Modules are the principle mechanism used to decompose large programs

Example – a compiler:

Diagram

Description automatically generated with medium confidence

At the end there will be some sort of linking and loading process

Modules usually offer an additional benefit: names within one module do not clash with names in other modules

Modules usually have a close relationship to separate compilation

Languages that have comprehensive module mechanisms:

* Ada, where they are called packages (not to be confused with Java packages)
* ML, where they are called structures

Languages that have weak mechanisms with some module-like properties:

* C++, where they are called namespaces
* Java, where they are called packages

Languages with no module mechanism:

* C (but modules can be imitated using separate compilation)
* Pascal

C++ Namespaces & Java Packages

C++ and Java do not have modules in the sense of Ada and ML: classes are used instead

C++ and Java do have mechanisms for controlling name clashes and organizing code into groups: namespaces in C++, packages in Java.

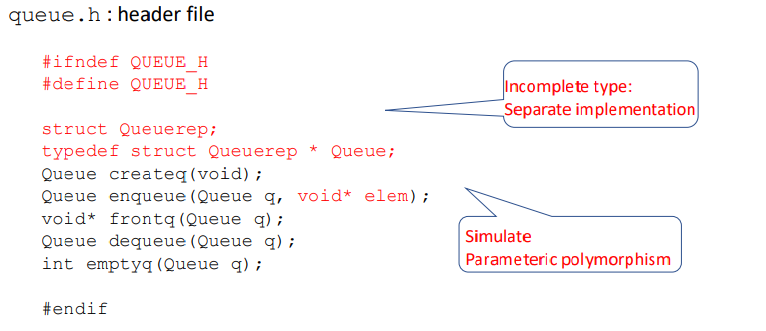
Clients must use similar dot notation as in Ada and ML to reference names in namespaces/packages.

Each of these languages has a mechanism for automatically dereferencing names:

* Ada: use
* ML: open
* C++: using [namespace]
* Java: import

Only Ada has explicit dependency syntax (keyword with). Java class loader automatically searches for code. C++ requires textual inclusions for declarations, linker must search for code. ML "compilation manager" does this too (not in ML specification).

C: Separate Compilation



Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

C++ Namespaces

Text

Description automatically generated with medium confidence

Graphical user interface, text, application

Description automatically generated

Java Packages

Diagram, text

Description automatically generated

Example:

Package java.util

* <http://java.sun.com/j2se/1.5.0/docs/api/java/util/package-summary.html>

Interface Collection

* <http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collection.html>

Class PriorityQueue

* <http://java.sun.com/j2se/1.5.0/docs/api/java/util/PriorityQueue.html>

boost: providing free peer-reviewed portable C++ source libraries

* <http://www.boost.org/>

Problems with Modules

Modules are not types

* Modules sometimes used to imitate OO classes
* Module interface usually contains types, whose representations may be exposed

Modules are static

* Modules are primarily compile-time artifacts
* Use of a module to imitate a class (without exporting a type) results in only oneavailable object

Modules do not control values of exported types

* Assignment can cause undesirable aliasing
* Equality tests may not be appropriate
* ML and Ada have some ability to control these (with effort)

ADT:

* Syntax
  + Can be achieved by module mechanism
* Semantic

These problems can be (mostly) overcome by using OO classes, with modules relegated to code organization status (C++, Java)

Significant problems still exist, even with OO :

* Modules do not expose dependencies (dependency of operations captured by the axioms is not there)
  + Only Ada documents compilation dependencies in code (keyword with)
  + Hidden implementation dependencies can be worse: order relation is a common one
  + C++ does a particularly good job of hiding these
  + Ada uses constrained polymorphism
  + Java uses interfaces such as Comparable, Comparator
* Modules do not express semantics
  + Universally ignored in today's languages
  + Useful for proving code correctness