ABSTRACT DATA TYPES

TODD D. VANCE

1. Abstract Data Type

One method for designing classes in an object-oriented fashion is via an *abstract data type*. The various abstract data types can be defined formally or semi-formally, or in a casual language. There are benefits to each of the various levels of formality. The author proposes an axiomatic approach (similar to Nell/Walker (http://dl.acm.org/citation.cfm?id=230973), though perhaps less formally; see also Liskov/Zilles 1974, Programming with Abstract Data Types) as a way of accurately communicating the specification of a class as well as unit tests. This approach to abstract data types can be thought of as being to the design of a system as "test driven development" is to the writing of the programs for a system.

2. Set

The ADT **set** can be specified by the following operations and axioms. We assume another type **E** representing the type of the elements of the set. This specifies an immutable set, where elements are added or removed by returning a new set. Here are the supported operations:

(1) empty : $\emptyset \to \mathbf{set}$.
(2) size : $\mathbf{set} \to \mathbf{N}$ (where $\mathbf{N}$ represents nonnegative integers).
(3) contains : $\mathbf{set} \times \mathbf{E} \to \mathbf{bool}$ (where $\mathbf{bool} = \{\text{true}, \text{false}\}$ represents the Boolean type).
(4) add : $\mathbf{set} \times \mathbf{E} \to \mathbf{set}$
(5) remove : $\mathbf{set} \times \mathbf{E} \to \mathbf{set}$

Informally, empy is a nullary operation, that is, a constant. It represents the empty set. The size operation applied to any set returns the number of elements. The contains operation applied to a set and an element returns true if the element is in the set. Otherwise, it returns false. The add operation applied to a set and an element returns the set whose elements are the elements of the specified set along with the specified element. The remove operation applied to a set and an element returns the set whose elements are the elements of the specified set except for the specified element.

The axioms specifying the behavior of the operations are as follows:

(1) $\text{size}(\text{empty}) = 0$
(2) if $\text{contains}(s, e) = \text{false}$, then $\text{size}(\text{add}(s, e)) = \text{size}(s) + 1$
(3) if $\text{contains}(s, e) = \text{true}$, then $\text{add}(s, e) = s$
(4) if $\text{contains}(s, e) = \text{true}$, then $\text{size}(\text{remove}(s, e)) = \text{size}(s) - 1$
(5) if $\text{contains}(s, e) = \text{false}$, then $\text{remove}(s, e) = s$
(6) $\text{contains}(\mathit{empty}, e) = \text{false}$

Date: August 15, 2016.

(7) contains(add(s, e), e) = true
(8) If $e \neq f$, then contains(add(s, e), f) = contains(s, f)
(9) contains(remove(s, e), e) = false
(10) If $e \neq f$, then contains(remove(s, e), f) = contains(s, f)

One could specify a mutable version of **set** as a structure having just one modifiable attribute: the immutable set. Thus, we have something like s, such that s.set is a **set**. We can modify s: s.set $\leftarrow t$ where t is some **set**.

Instead of empty is new **set** which is a structure whose attribute is equal to empty. So, $s \leftarrow$ new **set** causes s.set to equal empty.

The add operation then modifies s as follows: add(s, e) performs the operation s.set $\leftarrow$ add(set, e).

Similarly, the remove operation modifies s as follows: remove(s, e) performs the operation s.set $\leftarrow$ remove(set, e).

The remaining operations just operate directly on s.set. So, contains(s, e) is equal to contains(s.set, e), and likewise for the other operations.

The analogous scheme can be used to turn any immutable ADT having operations that return different instances of the ADT into a corresponding mutable ADT.