Lecture 12 - Abstraction Functions and Rep Invariants

Abstraction Functions

Maps the concrete representation to the abstract value it represents.

AF: Object $\to$ abstract value
AF(CharSet this) = $\{c \mid c \text{ is contained in this.elts}\}$

Rep Invariant: Condition that must hold for any valid representation
Domain of AF: Anything that satisfies the rep invariant
Codomain of AF: All possible outputs (larger than the range, which is what you actually get) **Bijective**: Different inputs produce different outputs, every element in codomain has at least one input that maps to it.

Example: stack

Context: implemented using an array

push: move top element pointer +1, add element to top index
pop: set top index to 0, move top element pointer −1
pop: move top element pointer −1

Both pop methods result in different states:
$< [17, 0, 0], \text{top} = 1 >$

$< [17, -9, 0], \text{top} = 1 >$

But these states map to the same abstract value.

AF is a function. Object $\to$ abstract value.
AF^{-1} is not a function. abstract value $\to$ object.

Example: member

```
boolean member(Character cl) {
    int i = elts.indexOf(cl);
    if (i == 1) { return false; }

    // Move-to-front optimization
    Character c2 = elts.elementAt(0);
    elts.set(0, cl);
    elts.set(1, c2);
    return true;
}
```

AF maps both reps to same abstract value.

Move-to-front speeds up repeated membership tests. Mutates rep, but does not change abstract value.

AF(auction) = $\{a, c, i, n, o, t, u\}$ = AF(caution)
AF(shrub) = $\{b, h, r, s, u\}$ = AF(brush)

Is member a mutator? No. Only the rep is mutated, not the abstract value.

Activity: Write AF for CharSet ADT:
Using List

AF(c) = $\{\mathrm{c.elts[i].charValue,}, 0 \leq i < \mathrm{c.elems.size}\}$

Using BitSet

AF(c) = $\{\mathrm{char(i)} \mid 0 \leq i < \mathrm{c.charPresence.size\ c.charPresence[i]} == \mathrm{True}\}$

Note: Precise english description is also sufficient

Tips on writing abstraction functions: **check notes**

ADTs and Java
- Make operations in ADT public
- Make other operations and fields of class private
- Clients can only access ADT operations

Java Interfaces
- Clients only see ADT, not implementation
- Multiple implementations have no code in common
- Cannot typically include creators (constructors) or fields

in general, we prefer interfaces to classes

Subtyping

Sometimes every B is an A.

In a library database:
- every book is a library holding
- every CD is a library holding

Subtyping expresses this:

B is a subtype of A means:

"every object that satisfies interface B also satisfies interface A". Goal: Code written using A's spec operates correctly even if given a ButPlus: clarify design, share tests, (sometimes) share code.

Subtypes are suitable for supertypes
- All subtypes satisfy supertypes spec
- All subtypes will not have moer expectations that supertypes

B is a true subtype of A if B has stronger or equal strength spec to A- This is not the same as a Java subtype (or subclass)

Substitution (subtype)
- B subtype of A iff an object of B can masquerade as an object of A in any Context

Inheritance (Java subclass)
- Extend a class
- Abstract out repeated code
- Not necessarily a true subtype