CSE 331
Software Design & Implementation

Winter 2026

Section 7 – ADT Correctness

Administrivia

- HW6 released tonight - **Due @ 11:59pm next Wed**

Review - Defensive Programming

Idea: Try to catch bugs early so that they do not propagate

- reduces debugging

Tools:

- Type checking
- Reasoning (i.e., floyd logic)
- Testing

Examples:

- Add checks for invalid inputs
- Check RI holds at the start and the end of mutators

Review - ADT Correctness

- To prove a method of a *mutable* ADT correct, we need to:
  - show the spec's postcondition holds
  - show the RI holds before and after each mutator
- Essentially, these compose the postcondition Q to prove

- For *immutable* ADTs
  - only need to prove RI is established by the constructor

    because fields don't change, we know RI will remain true

- AF doesn't need to be proven (it always holds when RI holds)

    it is just a given relationship between fields and obj to use in other proofs

    think of it like a fact :)

Review - Mutable ADT Correctness

Correctness checks required for mutable ADTS:

Constructors:

- creates concrete state satisfying the RI
- creates abstract state satisfying the AF
- ensure that there are no aliases

Observers:

- check that the return value is the one required by the spec

Mutators

- abstract state produced is the one required by the spec
- check the RI still holds at end and beginning of the method
- ensure that there are no aliases

Aliases

- Objects in “Heap State” means that its still being used after the call stack finishes.

- Extra references to these objects are called “aliases”

- When having aliases to mutable heap state:
  - We can gain efficiency in some cases.
  - We must keep track of all aliases that can mutate that state.

- For 331, mutable aliasing across files is a BUG! (Saw this in HW5)
  - Allows other portions of your code to break you code
    - **Representation Exposure**
  - “Copy in, copy out” to avoid aliases

Avoiding Aliases

(a) "Copy out": give out copies rather than references

```
// Class that maintains an array in a specific order
public class MyClass {
    // RI: vals is sorted
    private List<String> vals;
    …
    public List<String> values() {
        return this.vals;              // unsafe!
        return new ArrayList<>(vals); // make a copy
    }
}
…
```

- **Do not hand out access to your own array**

Aliases

(b) "Copy in": make a copy of anything you want to keep

```
public class MyClass {
    // RI: vals is sorted
    private List<String> vals;
    …
    // @requires A is sorted
    public MyClass(List<String> A) {
        this.vals = A;              // unsafe!
        this.vals = new ArrayList<>(A); // make a copy
}
…
```

- **Do not assign your own fields to something that someone else has access to.**