



6.170 Lecture 15 Understanding ADTs



MIT EECS

©MIT 6.170 Slide 1



Ways to get your design right


The hard way

- Start hacking
- When something doesn't work, hack some more
(How do you know it doesn't work?)
- Apply caffeine liberally

The easier way

- Plan first (specs, system decomposition, tests, ...)
- Less apparent progress upfront
- Faster completion times
- Better delivered product
- Less frustration

©MIT 6.170 Slide 2



Ways to verify your code

The hard way

- Make up some inputs
- If it doesn't crash, ship it
- When it fails in the field, attempt to debug


The easier way

- Reason about possible behaviors and desired outcomes
- Construct simple tests that exercise those behaviors

Another way that can be easy

- Prove that the system does what you want
 - Rep invariants are preserved
 - Implementation satisfies specification
- Proof can be formal or informal (we will be informal)
- Complementary to testing

©MIT 6.170 Slide 3




Uses of reasoning

Goal: correct code

- Verify that rep invariant is satisfied
- Verify that the implementation satisfies the spec
- Verify that client code behaves correctly
 - Assuming that the implementation is correct

©MIT 6.170 Slide 4



Goal: demonstrate that rep invariant is satisfied

Exhaustive testing

- Create every possible object of the type
- Check rep invariant for each object
- Problem: impractical


Limited testing

- Choose representative objects of the type
- Check rep invariant for each object
- Problem: did you choose well?

Reasoning

- Prove that all objects of the type satisfy the rep invariant
- Sometimes easier than testing, sometimes harder
- Every good programmer uses it as appropriate

©MIT 6.170 Slide 5



All possible objects (and values) of a type

Make a new object

- constructors
- producers

Modify an existing object

- mutators
- observers, producers (why?)

Limited number of operations, but infinitely many objects

- Maybe infinitely many values as well

©MIT 6.170 Slide 6

6.170 Examples of making objects

```

    a = constructor()
      / | \
    b = producer(a)  c = a.mutator()  d = a.observer()
      / | \
    e = producer(b)  f = b.mutator()  g = b.observer()
      / | \
    ...
  
```

Infinitely many possibilities
We cannot perform a proof that considers each possibility case-by-case

©MIT 6.170 Slide 7

6.170 Solution: induction

Induction: technique for proving infinitely many facts using finitely many proof steps

For constructors (“basis step”)
Prove the property holds on exit

For all other methods (“inductive step”)
Prove that if the property holds on entry, then it holds on exit

If the basis and inductive steps are true:
There is no way to make an object for which the property does not hold
The property holds for all objects

©MIT 6.170 Slide 8

6.170 Inductive proof that $x+1 > x$

ADT: the natural numbers (non-negative integers)
 constructor: 0 (zero)
 producer: succ (successor: $\text{succ}(x) = x+1$)
 mutators: none
 observers: value

Axioms:

- $\text{succ}(0) > 0$
- $(\text{succ}(i) > \text{succ}(j)) \Leftrightarrow i > j$

Goal: prove that for all natural numbers x , $\text{succ}(x) > x$

Possibilities for x :

- x is 0
 $\text{succ}(0) > 0$ axiom #1
- x is $\text{succ}(y)$ for some y
 $\text{succ}(y) > y$ assumption
 $\text{succ}(\text{succ}(y)) > \text{succ}(y)$ axiom #2
 $\text{succ}(x) > x$ def of x

©MIT 6.170 Slide 9

6.170 CharSet Abstraction

Overview: CharSets are finite sets of chars

```

public CharSet ( )
  effects: creates a fresh, empty CharSet
public void insert (char c);
  modifies: this
  effects: thispost = thispre ∪ {c}
public void delete (char c);
  modifies: this
  effects: thispost = thispre - {c}
public boolean member (char c);
  returns: (c ∈ this)
public int size ( );
  returns: cardinality of this
  
```

©MIT 6.170 Slide 10

6.170 Outline for remainder of lecture

- 1. Prove that rep invariant is satisfied**
- 2. Prove that client code behaves correctly**
(Assuming that the implementation is correct)

©MIT 6.170 Slide 11

6.170 Implementation of CharSet

Rep invariant: elts has no nulls and no duplicates

```

public CharSet ( ) {
  elts = new ArrayList <Character>( );
}
public void delete (char c) {
  elts.remove (new Character (c));
}
public void insert (char c) {
  if (!member(c)) elts.add (new Character (c));
}
public boolean member (char c) {
  return elts.contains (new Character (c));
}
...
  
```

©MIT 6.170 Slide 12



Proof of CharSet representation invariant

Rep invariant: elts has no nulls and no duplicates

Base case:

Constructor sets elts to the empty ArrayList<Character>
This satisfies the rep invariant

Inductive step:

For each other operation:

Assume rep invariant holds before the operation
Prove rep invariant holds after the operation

©MIT 6.170

Slide 13



Inductive step, member

Rep invariant: elts has no nulls and no duplicates

```
public boolean member (char c) {
    return elts.contains (new Character (c));
}
```

member doesn't change elts, so rep invariant is preserved

Why do we even need to check member?

After all, the specification says that it does not mutate set.

Reasoning must account for all possible arguments

Best not to involve the specific values in the proof

©MIT 6.170

Slide 14



Inductive step, delete

Rep invariant: elts has no nulls and no duplicates

```
public void delete (char c) {
    elts.remove (new Character (c));
}
```

Either leaves elts unchanged or removes element.
Rep invariant can only be made false by adding elements.
Thus, rep invariant is preserved.

©MIT 6.170

Slide 15



Inductive step, insert

Rep invariant: elts has no nulls and no duplicates

```
public void insert (char c) {
    if (!this.member(c)) elts.add (new Character (c));
}
```

If c is in $elts_{pre}$:
elts is unchanged, so rep invariant is preserved.

If c is not in $elts_{pre}$:
new elt is not null or a duplicate, so rep invariant is preserved

©MIT 6.170

Slide 16



Mutations to the rep

The inductive step must consider all possible changes to the rep

If there is representation exposure, and the proof does not account for this, then the proof is invalid

This is an important reason to protect the rep: then the compiler can help you verify that there are no external changes

©MIT 6.170

Slide 17



Induction for reasoning about uses of ADT's

Induction on specification, not on code

Abstract values (e.g., specification fields) may differ from concrete representation

Can ignore observers, since they do not affect abstract state

How do we know that?

Axioms

specs of operations

axioms of types used in overview parts of specifications

©MIT 6.170

Slide 18



Letter sets (case-insensitive character sets)

Overview: A LetterSet (case-insensitive char set) is a finite set of characters. No LetterSet contains two chars with the same lower-case representation.

```

public LetterSet ();
  effects: creates an empty LetterSet
public void insert (char c);
  // Insert c if no other char with same lower-case representation.
  modifies: this
  effects: thispost =
    if (∃c1 ∈ this s.t. toLowerCase(c1) = toLowerCase(c))
      then thispre U {c}
    else thispre
public void delete (char c);
  modifies: this
  effects: thispost = thispre - {c}
public boolean member (char c);
  returns: (c ∈ this)
public int size ();
  returns: |this|

```



Goal: prove that LetterSet contains two letters

Prove: $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S [\text{toLowerCase}(c_1) \neq \text{toLowerCase}(c_2)])$

Two possibilities for how S was made: by the constructor, or by insert

Base case: $S = \{ \}$, (made by the constructor):

property holds (vacuously true)

Inductive case (S made by a call of the form "T.insert(c)"):

Assume: $|T| > 1 \Rightarrow (\exists c_3, c_4 \in T [\text{toLowerCase}(c_3) \neq \text{toLowerCase}(c_4)])$

Show: $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S [\text{toLowerCase}(c_1) \neq \text{toLowerCase}(c_2)])$

where $S = \text{"if } (\exists c_5 \in T \text{ s.t. } \text{toLowerCase}(c_5) = \text{toLowerCase}(c))$

then T else T U {c}"

The value for S came from the specification of insert, applied to T.insert(c):

public void insert (char c);

modifies: this

effects: this_{post} = if (∃c₁ ∈ S s.t. toLowerCase(c₁) = toLowerCase(c))

then this_{pre}

else this_{pre} U {c}



Inductive case: S = T.insert(c)

Goal (from previous slide):

Assume: $|T| > 1 \Rightarrow (\exists c_3, c_4 \in T [\text{toLowerCase}(c_3) \neq \text{toLowerCase}(c_4)])$

Show: $|S| > 1 \Rightarrow (\exists c_1, c_2 \in S [\text{toLowerCase}(c_1) \neq \text{toLowerCase}(c_2)])$

where $S = \text{"if } (\exists c_5 \in T \text{ s.t. } \text{toLowerCase}(c_5) = \text{toLowerCase}(c))$

then T else T U {c}"

Consider the two possibilities for S (from "if ... then T else T U {c}"):

If $S = T$, the theorem holds by induction hypothesis (the assumption above)

If $S = T U \{c\}$, there are three cases to consider:

©MIT 6.170 $|T| = 0$: Vacuous case, since hypothesis of theorem

("|S| > 1") is false