Exam 1 Review

Exam 1

- Exam 1 today at 6:55pm-8:45pm.

- Honor Code:
  - Open book, open notes, open slides,
  - no use of compilers, no communication with others.
  - You must only submit *__your own__* answers.
- Type into Submitty (like Quizzes).

Topics

- <u>Reasoning about code</u>
  - Forward and backward reasoning, logical conditions, Hoare triples, weakest precondition, rules for assignment, sequence, if-then-else, loops, loop invariants, decrementing functions

Forward Reasoning

- Forward reasoning simulates the execution of code. Introduces facts as it goes along

  E.g., { x = 1 }

  ```
  y = 2*x
  ```

  { x = 1 && y = 2 }

  ```
  z = x + y
  ```

  { x = 1 && y = 2 && z = 3 }

  - Collects all facts, often those facts are irrelevant to the goal

Backward Reasoning

- Backward reasoning "goes backwards". Starting from a given postcondition, finds the weakest precondition that ensures the postcondition

E.g., { y < 1 } // Simplify 2y < y+1 into { y < 1 }

```
z = y + 1
```
// Substitute y+1 for z in 2y < z

{ 2*y < z }

```
x = 2*y
```
// Substitute rhs 2*y for x in x < z

{ x < z }

- More focused and more useful

Condition Strength

- “P is stronger than Q” means “P implies Q”
- “P is stronger than Q” means “P guarantees no less than Q”
  - E.g., x>0 is stronger than x>-1
- Values satisfying P also satisfy Q
  - E.g., values satisfying x>0 also satisfy x>-1

- Stronger means more specific
- Weaker means more general

Condition Strength

- Which one is stronger?

$x > -10$ or $\underline{x > 0}$

$\underline{x > 0 \wedge y = 0}$ or $x > 0 \vee y = 0$

$0 \leq x \leq 10$ or $5 \leq x \leq 11$ (Neither!)

$\underline{y \equiv 2 \pmod 4}$ or y is even

$\underline{x = 10}$ or x is even

Hoare Triples

- A Hoare Triple: { P } **`code`** { Q }
  - P and Q are logical statements about program values, and **`code`** is program code (in our case, Java code)
- "{ P } **`code`** { Q }" means "if P is true and we execute **`code`**, and it terminates, then Q is true afterwards"
  - "{ P } **`code`** { Q }" is a logical formula, just like "0≤index"

Examples of Hoare Triples

{ x>0 } **`x++`** { x>1 } is true

{ x>0 } **`x++`** { x>-1 } is true

{ x≥0 } **`x++`** { x>1 } is false. Why?

{x>0} **`x++`** {x>0} is ??

{x<0} **`x++`** {x<0} is ??

{x=a} **`if (x < 0) x=-x`** { x = | a | } is ??

{x=y} **`x=x+3`** {x=y} is ??

Rules for Backward Reasoning: Assignment

// precondition: ??

`x = expression`

// postcondition: Q

Rule: the weakest precondition =
Q, with all occurrences of **`x`** in Q replaced by **`expression`**

More formally:

wp("**`x=expression;`**",Q) = Q with all occurrences of **`x`** replaced by **`expression`**

Rules for Backward Reasoning: Sequence

// precondition: ??
`S1;` // statement
`S2;` // another statement
// postcondition: Q

Work backwards:
precondition is wp("`S1;S2;`", Q) = wp("`S1;`",wp("`S2;`",Q))

Example:

```
// precondition: ??
x = 0;
y = x+1;
// postcondition: y>0
```

```
// precondition: ??
x = 0;
// postcondition for x=0; same as
// precondition for y=x+1;
y = x+1;
// postcondition y>0
```

Rules for If-then-else

Forward reasoning

{ P }
if b
  { P ∧ **b** }
  S1
  { Q1 }
else
  { P ∧ ¬**b** }
  S2
  { Q2 }
{ Q1 ∨ Q2 }

Backward reasoning

{ (**b**∧wp("**S1**",Q))∨(¬**b**∧wp("**S2**",Q)) }
if b
  { wp("**S1**",Q) }
  S1
  { Q }
else
  { wp("**S2**",Q) }
  S2
  { Q }
{ Q }

Reasoning About Loops by Induction

1. Partial correctness
   - Figure out loop invariant
   - Prove loop invariant using computation induction
   - Loop exit condition (i.e., !b) and loop invariant imply the desired postcondition
2. Termination
   - Figure out "decrementing function" D:
     1) D is bounded, 2) at each iteration D decreases, and 3) D = 0 (or min) AND loop invariant, imply loop exit condition (i.e., D = 0 (or min) AND loop invariant imply !b)

Example: Partial correctness

i+z = x is our loop invariant

Precondition: x >= 0;

```
i = x;
z = 0;
while (i != 0) {
   z = z+1;
   i = i-1;
}
```

Postcondition: x = z;

1) i=x and z=0 give us that i+z = x holds at 0th iteration of loop // Base case
2) Assuming that i+z = x holds after k^{th} iteration, we show it holds after $(k+1)^{st}$ iteration // Induction
   $z_{new} = z + 1$ and $i_{new} = i - 1$ thus
   $z_{new} + i_{new} = z + 1 + i - 1 = z + i = x$
3) If loop terminated, we know i = 0. Since z+i = x holds, we have x = z

Example: Termination

Precondition: x >= 0;

```
i = x;
z = 0;
while (i != 0) {
   z = z+1;
   i = i-1;
}
```

Postcondition: x = z;

First, prove that i >= 0. Indeed i >= 0 is a loop invariant.

Decrementing function D is i.
1) D is bounded below by 0 (see above).
2) D decreases
3) D = 0 implies i = 0 (loop exit condition).

Reasoning About Loops

```
{P} while (b) S {Q}
```

Reasoning About Loops

- Loop invariant Inv must be such that

1) P => Inv // Inv holds before loop. <u>Base case</u>
2) { Inv && **b** } **S** { Inv } // Assuming Inv held after k^{th} iteration and execution took a $(k+1)^{st}$ iteration, then Inv holds after $(k+1)^{st}$ iteration. <u>Induction</u>
3) (Inv && !**b**) => Q // The exit condition **!b** and loop invariant Inv must imply postcondition

- Decrementing function D must be such that

1) D is bounded
2) D decreases every time we go through the loop
3) (D = 0 (or min) && Inv) => **!b**

Topics

- <u>Specifications</u>
  - Benefits of specifications, PoS specification convention, specification style, specification strength (stronger vs. weaker specifications), comparing specifications via logical formulas, converting PoS specifications into logical formulas

Specifications

- A specification consists of a precondition and a postcondition
  - Precondition: conditions that hold before method executes
  - Postcondition: conditions that hold after method finished execution (if precondition held!)

Specifications

- A specification is a contract between a method and its caller
  - Obligations of the method (implementation of specification): agrees to provide postcondition if precondition held!
  - Obligations of the caller (user of specification): agrees to meet the precondition and not expect more than postcondition promised

Example Specification

Precondition: `len ≥ 1 && arr.length = len`

Postcondition: returns `arr[0]+…+arr[arr.length-1]`

```
double sum(int[] arr, int len) {
    double sum = arr[0];
    int i = 1;
    while (i < len) {
        sum = sum + arr[i];
        i = i+1;
    }
    return sum;
}
```

For our purposes, we can write specifications that are a bit less formal than this example. Mathematical rigor is welcome, but not always necessary.

Benefits of Specifications

- Precisely documents method behavior
  - Imagine if you had to read the code of the Java libraries to figure what they do!
  - An abstraction --- abstracts away unnecessary detail
- Promotes modularity
- Enables reasoning about correctness
  - Through testing and/or verification

PoS Specifications

- Specification convention due to Michael Ernst
- The precondition
  - requires: clause spells out constraints on client
- The postcondition
  - modifies: lists objects (typically parameters) that may be modified by the method. Any object not listed under this clause is guaranteed untouched
  - effects: describes final state of modified objects
  - returns: describes return value
  - throws: lists possible exceptions

Example

```
static List<Integer> listAdd(List<Integer> lst1,
                             List<Integer> lst2)
```

requires: **`lst1`**, **`lst2`** are non-null.
`lst1` and **`lst2`** are same size.

modifies: none

effects: none

returns: a new list of the same size, whose i-th element is the sum of the i-th elements of **`lst1`** and **`lst2`**

```
static List<Integer> listAdd(List<Integer> lst1,
                             List<Integer> lst2) {
 List<Integer> res = new ArrayList<Integer>();
 for (int i = 0; i < lst1.size(); i++)
   res.add(lst1.get(i) + lst2.get(i));
 return res;
}
```

Another Example

```
static void listAdd2(List<Integer> lst1,
                     List<Integer> lst2)
```

requires: **`lst1`**, **`lst2`** are non-null.
`lst1` and **`lst2`** are same size.

modifies: **`lst1`**

effects: i-th element of **`lst1`** is replaced with the sum of i-th elements of **`lst1`** and **`lst2`**

returns: none

```
static void listAdd(List<Integer> lst1,
                    List<Integer> lst2) {
 for (int i = 0; i < lst1.size(); i++) {
   lst1.set(i,lst1.get(i) + lst2.get(i));
 }
}
```

Specification Style

- A method is called for its side effects (effects clause) or its return value (returns clause)
  - It is bad style to have both effects and return
  - There are exceptions
    - E.g., HashMap.put returns the previous value
- Main point of spec is to be helpful
  - Being overly formal may not help
  - Being too informal does not help either
- If spec turns too complex: redesign. Better to simplify code than document complexity!

What's Wrong?

```
static void uniquefy(List<Integer> lst)
```

requires: ?
modifies: ?
effects: ?
returns: ?

```
static void uniquefy(List<Integer> lst) {
  for (int i = 0; i < lst.size()-1; i++)
   if (lst.get(i) == lst.get(i+1))
     lst.remove(i);
}
```

Specification Strength

- "A is stronger than B" means
  - For every implementation *I*
    - "*I* satisfies A" implies "*I* satisfies B"
    - The opposite is not necessarily true
  - For every client *C*
    - "*C* works with B" implies "*C* works with A"
    - The opposite is not necessarily true
- Principle of substitutability:
  - <u>A stronger spec can always be substituted for a weaker one</u>

Why Care About Specification Strength?

- Because of substitutability!

- Principle of substitutability
  - A stronger specification can always be substituted for a weaker one
  - I.e., an implementation that conforms to a stronger specification can be used in a client that expects a weaker specification

Substitutability

- Substitutability guarantees correct software updates, correct class hierarchies
- Client code: `X x; … x.foo(index);`
  - Client is "polymorphic": written against `X`, but it is expected to work with <u>any</u> subclass of `X`
  - A subclass of `X`, say `Y`, may have its own implementation of `foo, Y.foo(int)`. Client must work correctly with `Y.foo(int`!
- If spec of `Y.foo(int)` is stronger than that of `X.foo(int)` then we can safely substitute `Y.foo(int)` for `X.foo(int)`!

Strengthening and Weakening Specification

- Strengthen a specification
  - Require less of client: fewer conditions in requires clause
  - Promise more to client: effects, modifies, returns
- Weaken a specification
  - Require more of client: add conditions to requires
  - Promise less to client: effects, modifies, returns clauses are weaker, thus easier to satisfy in code

Comparing by Logical Formulas

- Let Spec A: $\{P_A\}$ **code** $\{Q_A\}$,
  Spec B: $\{P_B\}$ **code** $\{Q_B\}$.

We say code satisfies a specification with precondition P and postcondition Q iff {P} **code** {Q} Hoare triple is true.

Do not confuse it with P => Q.

e.g., { x = 0 } `x = 1;` { x = 1 } is true, but

x = 0 => x = 1 is false.

also, { x = 0 } `x = -1;` { x >= 0 } is false, but

x = 0 => x >= 0 is true.

Comparing by Logical Formulas

- Let Spec A: $\{P_A\}$ **code** $\{Q_A\}$,
  Spec B: $\{P_B\}$ **code** $\{Q_B\}$.

The following are equivalent:

- $P_B \Rightarrow P_A$ and $Q_A \Rightarrow Q_B$
- A is stronger than B
- A => B

Comparing by Logical Formulas

Let $A = \{P_A\}$ code $\{Q_A\}$,

$B = \{P_B\}$ code $\{Q_B\}$ be Hoare triples.

A is stronger than B if and only if P_A is weaker than P_B and Q_A is stronger than Q_B, i.e.,

- $A \Rightarrow B \iff (P_B \Rightarrow P_A \wedge Q_A \Rightarrow Q_B)$.

$A \Rightarrow B$ means that any code satisfying A also satisfies B.

Exercise: Order by Strength

Spec A: requires: a non-negative int argument

returns: an int in [1..10]

Spec B: requires: int argument

returns: an int in [2..5]

Spec C: requires: true

returns: an int in [2..5]

Spec D: requires: an int in [1..10]

returns: an int in [1..20]

Converting PoS Specs into Logical Formulas

- PoS specification

  requires: R

  modifies: M

  effects: E // absorbs throws, returns and effects

Spec is equivalent to the following logical formula:

{R} code { E ˆ (nothing but M is modified)}

Step 1: absorb throws and returns into effects E

Step 2: {R} code {(E ˆ(nothing but M is modified)}

Convert Spec to Formula, step 1: absorb throws and returns into effects

- `set` from `java.util.ArrayList<T>`

```
T set(int index, T element)
```

requires: true
modifies: this[index]
effects: $this_{post}$[index] = element
throws: IndexOutOfBoundsException if index < 0 || index ≥ size
returns: $this_{pre}$[index]

Absorb effects, returns and throws into new effects:

if index < 0 || index ≥ size then
    throws IndexOutOfBoundsException
  else $this_{post}$[index] = element and returns $this_{pre}$[index]

Convert Spec to Formula, step 2: Convert into Formula

- **`set`** from **`java.util.ArrayList<T>`**

```
T set(int index, T element)
```

requires: true
modifies: this[index]
effects: if index < 0 || index ≥ size then
throws IndexOutOfBoundsException
else
$this_{post}$[index] = element and returns $this_{pre}$[index]

Denote effects expression by E. Resulting formula is:

{true} code {(E ∧ (forall i ≠ index, $this_{post}$[i] = $this_{pre}$[i]))

Stronger Specification

- S1 is stronger than S2 iff

$\{R_1\}$ code $\{E_1 \wedge (\text{only } M_1 \text{ is modified})\}$
=>
$\{R_2\}$ code $\{E_2 \wedge (\text{only } M_2 \text{ is modified})\}$

iff $R_2 \Rightarrow R_1 \wedge (E_1 \wedge (\text{only } M_1 \text{ is modified}) \Rightarrow (E_2 \wedge (\text{only } M_2 \text{ is modified}))$

if $R_2 \Rightarrow R_1 \wedge E_1 \Rightarrow E_2 \wedge (\text{only } M_1 \text{ is modified}) \Rightarrow (\text{only } M_2 \text{ is modified})$

iff $R_2 \Rightarrow R_1 \wedge E_1 \Rightarrow E_2 \wedge (M_1 \subseteq M_2)$

Stronger Specification

- S1 is stronger than S2 if $R_2 \Rightarrow R_1 \wedge E_1 \Rightarrow E_2 \wedge (M_1 \subseteq M_2)$

- A stronger specification:
  - Requires less
  - Guarantees more
  - Modifies less

Topics

- ADTs
  - Benefits of ADT methodology, Specifying ADTs, Rep invariants, Representation exposure, Checking rep invariants, Abstraction functions

ADTs

- Abstract Data Type (ADT): higher-level data abstraction
  - The ADT is operations + object
  - A specification mechanism
  - A way of thinking about programs and design

An ADT Is a Set of Operations

- Operations operate on data representation
- ADT abstracts from organization to meaning of data
- ADT abstracts from structure to use
- Data representation does not matter!

```
class Roint {
□ float x, y;
}
```

```
class Point {
□ float r, theta;
}
```

- Instead, think of a type as a set of operations: create, x(), y(), r(), theta().
- Force clients to call operations to access data

Specifying an ADT

immutable	**mutable**
`class TypeName`	**`class TypeName`**
1. overview	1. overview
2. abstract fields	2. abstract fields
3. creators	3. creators
4. observers	4. observers
5. producers	5. producers (rare!)
~~6. mutators~~	6. mutators

Connecting Implementation to Specification

- Representation invariant: Object → boolean
  - Indicates whether data representation is well-formed. Only well-formed representations are meaningful
  - Defines the set of valid values
- Abstraction function: Object → abstract value
  - What the data structure really means
    - E.g., array [2, 3, -1] represents $-x^2 + 3x + 2$
  - How the data structure is to be interpreted

Representation Exposure

- Client can get control over rep and break the rep invariant! Consider

```
IntSet s = new IntSet();
s.add(1);
List<Integer> li = s.getElements();
li.add(1); // Breaks IntSet's rep invariant!
```

- Representation exposure is external access to the rep. **AVOID!!!**
- If you allow representation exposure, document why and how and feel bad about it

Representation Exposure

- Make a copy on the way out:

```
public List<Integer> getElements() {
  return new ArrayList<Integer>(data);
}
```

- Mutating a copy does not affect **`IntSet`**'s rep

```
IntSet s = new IntSet();
s.add(1);
List<Integer> li = s.getElements();
li.add(1); //mutates new copy, not IntSet's rep
```

Representation Exposure

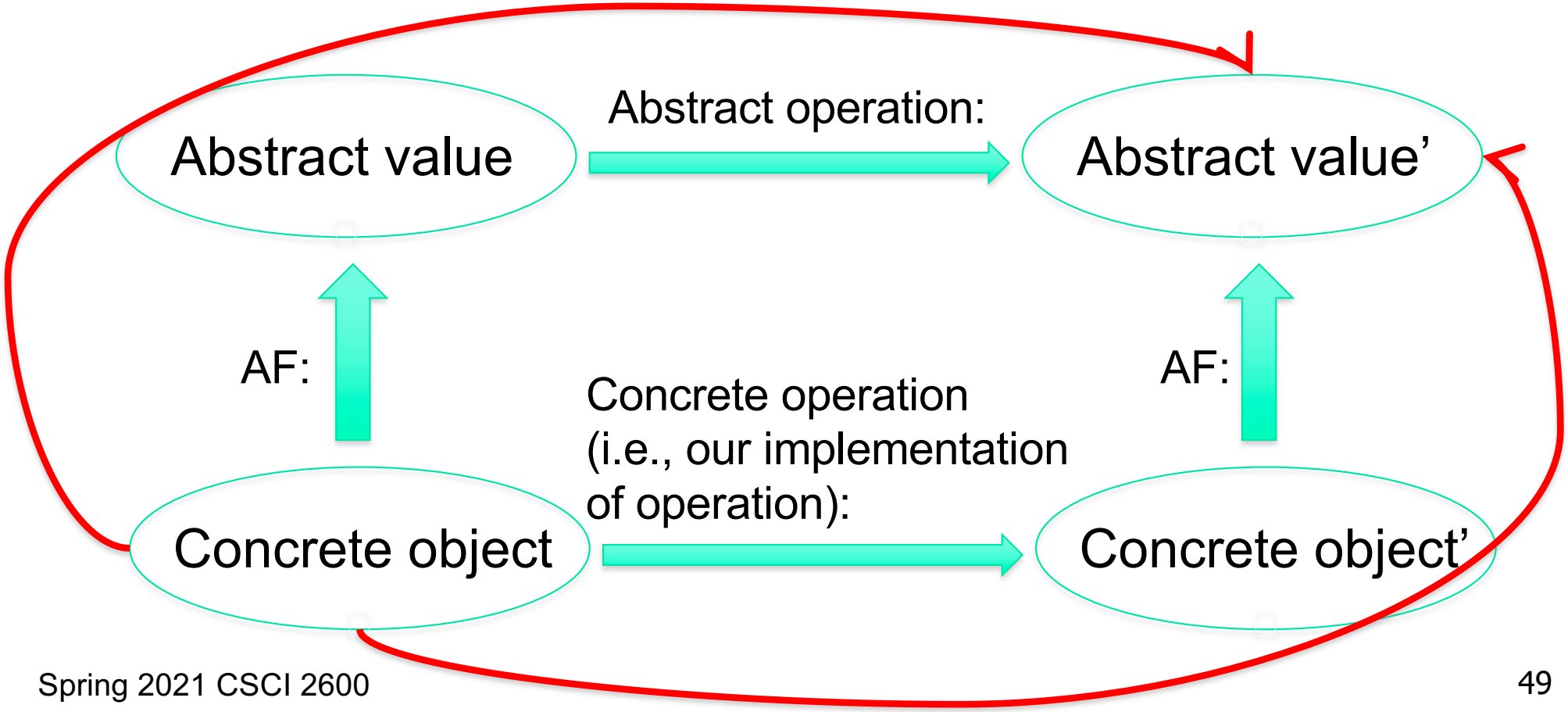
- Make a copy on the way in too:

```
public IntSet(ArrayList<Integer> elts) {
  data = new ArrayList<Integer>(elts);
  …
}
```

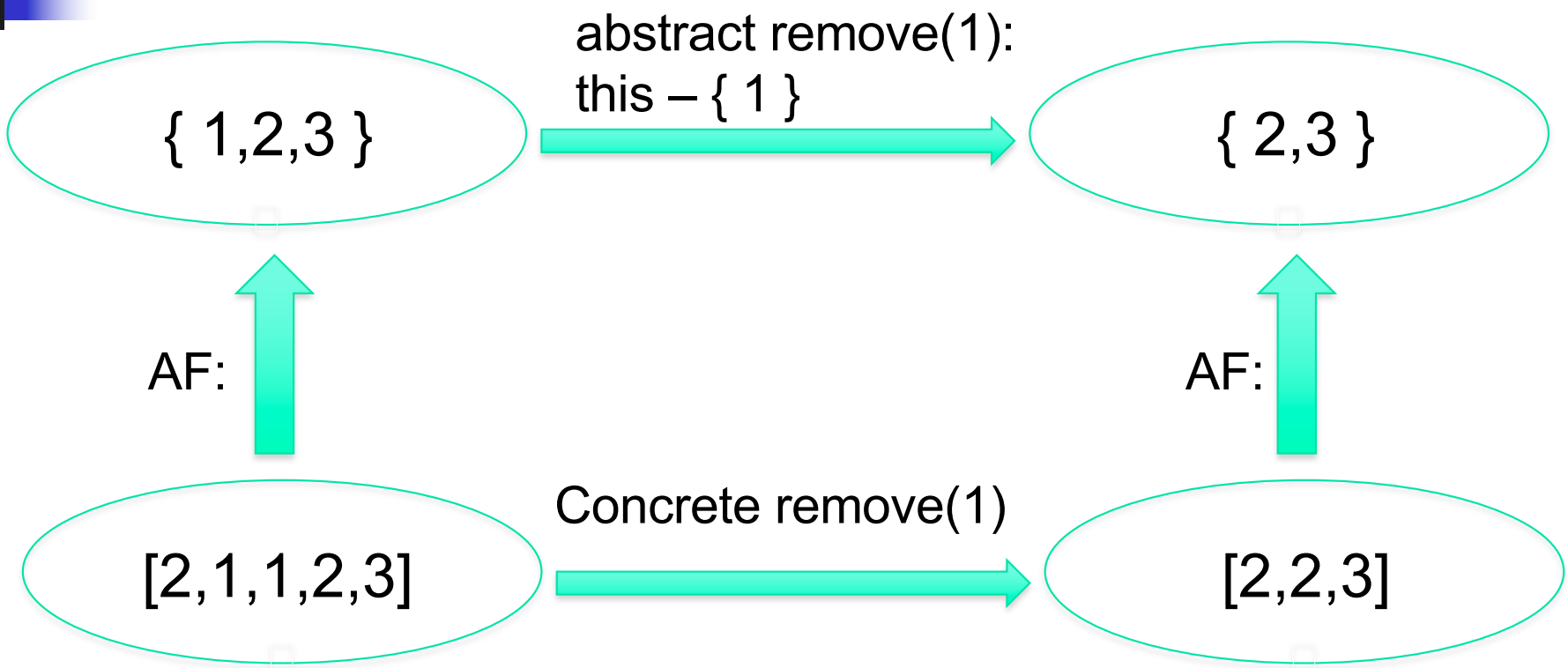
- Why?

Abstraction Function

- Abstraction function allows us to reason about correctness of the implementation



IntSet Example



Creating concrete object:
- Establish rep invariant
- Establish abstraction function

After every operations:
- Maintains rep invariant
- Maintains abstraction function