CS 326
Representation Invariants and Abstraction Functions

Stephen Freund

1

Announcements

- Due Today!
  - Lab 1
  - HW 2
- Due Thursday!
  - HW 3
- Due next week
  - HW 4 (on today's material)
- Lab 2 on Thursday
  - Writing our first iOS app
  - No prelab

2

CS326 Method Specifications

```
/**
 ...

 **Requires**: none (can omit in this case)

 **Modifies**: self

 **Effects**: Changes the first occurrence of oldValue to newValue

 - Parameter oldValue: element to replace.
 - Parameter newValue: what to replace it with.
 - Returns: The first index where oldValue is found, or nil
   if it does not occur in the list.
 */
func replace(_ oldValue: T, with newValue: T) -> Int? {
  for i in 0..<count {
    if get(i) == oldValue {
      set(i, to: newValue)
      return i
    }
  }
  return nil
}
```

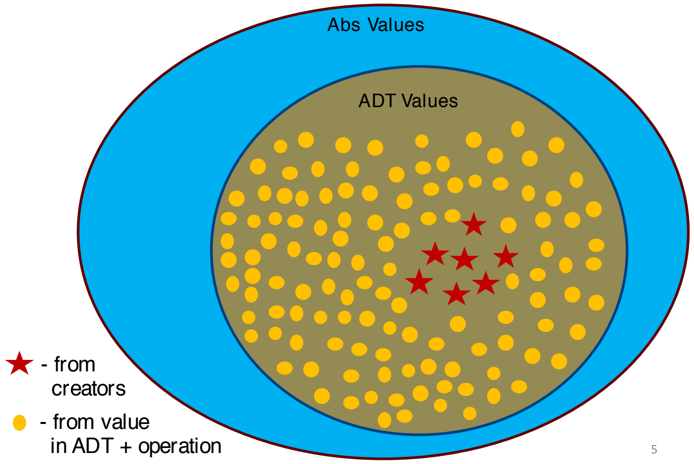
3

IntSet Spec

```
/// Overview: An IntSet is a mutable,            } Overview
/// unbounded set of integers.
/// A typical IntSet is { x1, ..., xn }.          } Abstract State
class IntSet {

  /// **Effects**: makes a new IntSet = {}         } Creator
  public init()

  /// - Returns: true if and only if element in self   } Observer
  public func contains(_ element: Int) -> Bool

  /// **Modifies**: self                            } Mutators
  /// **Effects**: self_post = self_pre U { element }
  public func add(_ element : Int)

  /// **Modifies**: self
  /// **Effects**: self_post = self_pre - { element }
  public func remove(_ element : Int)
```

4

ADTs and Specs



IntSet Implementation. Ok?

```
class IntSet {

  private var elems = [Int]()

  public func contains(_ element: Int) -> Bool {
    return elems.contains(element)
  }

  public func add(_ element : Int) {
    elems.append(element)
  }

  public func remove(_ element : Int) {
    if let index = elems.firstIndex(of:element) {
      elems.remove(index)
    }
  }
}
```


IntSet Implementation. Ok?

```
class IntSet {

  private var elems = [Int]()

  public func contains(_ element: Int) -> Bool {
    return elems.contains(element)
  }

  public func add(_ element : Int) {
    elems.append(element)
  }

  public func remove(_ element : Int) {
    if let index = elems.firstIndex(of:element) {
      elems.remove(index)
    }
  }
}
```

```
let s = IntSet()
s.add(3)
s.add(3)
s.remove(3)
assert !s.contains(3)
```


IntSet Rep Invariant

```
class IntSet {
  // Rep Invariant: elems has no duplicates
  private var elems = [Int]()

  public func contains(_ element: Int) -> Bool {
    return elems.contains(element)
  }

  public func add(_ element : Int) {
    elems.append(element)
  }

  public func remove(_ element : Int) {
    if let index = elems.firstIndex(of:element) {
      elems.remove(index)
    }
  }
}
```

Rep Invariant for ADT

Client

```
/// ...
/// A typical IntSet
/// is { x1, ..., xn }.
class IntSet {
```

Implementer

```
class IntSet {
  var elems : [Int]
  ...
```

[1, 1, 1]
[1, 4, 3]
[2, 1]
[1, 3, 4]
[1, 2]
[]
[3, 4, 1]
[1, 1, 2]
[1, 3, 4, 3]

9

Rep Invariant for ADT

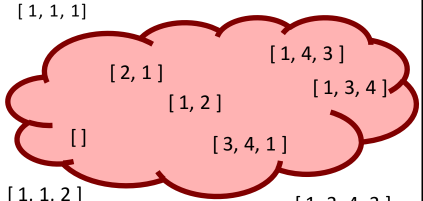
Client

```
/// ...
/// A typical IntSet
/// is { x1, ..., xn }.
class IntSet {
```

Implementer

```
class IntSet {
  var elems : [Int]
  ...
```

[1, 1, 1]



[1, 1, 2]
[1, 3, 4, 3]

Representation Invariant: RI(self) = { self.elems has no duplicates }

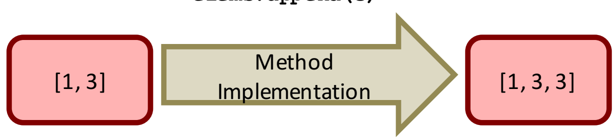
Rep Invariant Must Be Preserved

RI(Object) ✔ RI(Object') ✔

11

object.add(3)

```
add(3) ->
  elems.append(3)
```



RI(Object) ✔ RI(Object') ✘

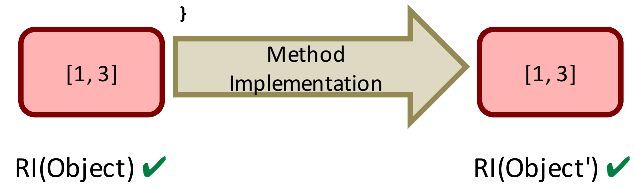
12

IntSet Rep Invariant

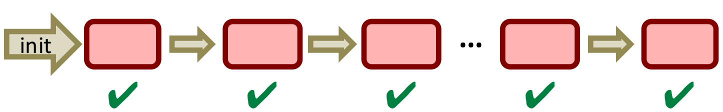
```
class IntSet {
  // Rep Invariant: elems has no duplicates
  private var elems = [Int]()

  public func contains(_ element: Int) -> Bool {
    return elems.contains(element)
  }

  public func add(_ element : Int) {
    if (!contains(element)) {
      elems.append(element)
    }
  }

  public func remove(_ element : Int) {
    if let index = elems.index(of:element) {
      elems.remove(index)
    }
```


object.add(3)

```
if !elems.contains(3) {
  elems.append(3)
}
```



Rep Invariant Must Be Preserved



Another Example

```
class Account {
   var balance : Int

   // history of all transactions
   var transactions : [Transaction]
   …
}
```

Real-world constraints:

- balance ≥ 0
- balance = Sum { t.amount | t in transactions }

Implementation-related constraints:

- forall t in transactions, tcompletionDate != nil

Checking the Rep Invariant

Rule of thumb: check on entry *and* on exit (why?)

```
public func remove(_ element : Int) {
  checkRep()
  if let index = elems.firstIndex(of:element) {
    elems.remove(index)
  }
  checkRep()
}

…
// Verify that elems contains no duplicates.
private func checkRep() {
  for i in 0..<elems.count {
    assert elems.firstIndex(of: elems[i]) == i
  }
}
```

17

IntSet V2

```
class IntSet {

  // Rep Invariant: elems has no duplicates
  private let elems = MutableList<Int>()

  public func contains(_ element: Int) -> Bool {
    checkRep()
    return elems.contains(element)
  }

  public func add(_ element : Int) {
    checkRep()
    if (!contains(element) {
      elems.append(element)
    }
    checkRep()
  }

  public func remove(_ element : Int) {
    checkRep()
    elems.remove(element)
    checkRep()
  }
}
```

```
class MutableList<T> {
  var count : Int
  func get(index: Int) -> T
  func set(index: Int, to: T)
  func append(_ e: T)
  func remove(_ e: T)
  func contains(_ e: T) -> Bool
}
```

18

IntSet V3

```
class IntSet {

  // Rep Invariant: elems has no duplicates
  private let elems = MutableList<Int>()

  public func contains(_ element: Int) -> Bool {
   checkRep()
   return elems.contains(element)
  }

  public func add(_ element : Int) {
   checkRep()
   if (!contains(element) {
     elems.append(element)
   }
   checkRep()
  }

  public func remove(_ element : Int) {
   checkRep()
   elems.remove(element)
   checkRep()
  }

  /// - Returns: A list containing the members of self
  public func getElements() -> MutableList<Int>() {
    return elems
  }
```

19

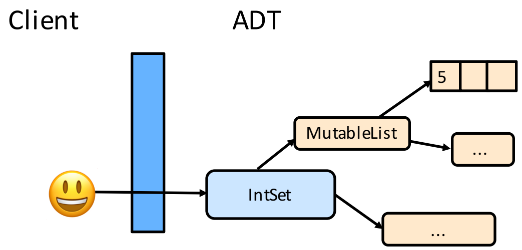
IntSet V3

```
class IntSet {

  // Rep Invariant: elems has no duplicates
  private let elems = MutableList<Int>()

  public func contains(_ element: Int) -> Bool {
   checkRep()
   return elems.contains(element)
  }

  public func add(_ element : Int) {
   checkRep()
   if (!contains(element) {
     elems.append(element)
   }
   checkRep()
  }

  public func remove(_ element : Int) {
   checkRep()
   elems.remove(element)
   checkRep()
  }

  /// - Returns: A list containing the members of self
  public func getElements() -> MutableList<Int>() {
    return elems
  }
```

```
let s = IntSet()
s.add(5)
let elems = s.getElements()
elems.add(5)
s.remove(5)
assert !s.contains(5)
```

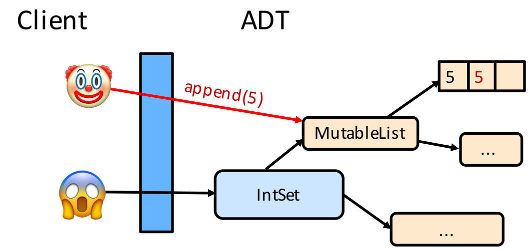
20

`private` Is Not Enough



21

`private` Is Not Enough



22

Solution 1: Copy In. Copy Out.

(assume `Point` is a mutable ADT)

```
public class Point {
  var x : Int
  var y : Int
}
```

```
public class Line {

  private var s : Point
  private var e : Point

  public init(s : Point, e : Point) {
    self.s = Point(s.x, s.y)
    self.e = Point(e.x, e.y)
  }

  public var start : Point {
    return Point(self.s.x, self.x.y)
  }

  ...
}
```

23

Shallow Copy

What's the bug (assuming `Point` is a mutable ADT)?

```
class PointSet {

  private var points = MutableList<Point>()

  public getElements() -> MutableList<Point> {
    let result = MutableList<Point>()
    for p in points {
      result.append(p)
    }
    return result
  }
}
```

24

Shallow Copy

What's the bug (assuming `Point` is a mutable ADT)?

```
class PointSet {

  private var points = [Point]()

  public getElements() -> [Point] {
    return points
  }
}
```


Deep Copy

```
class PointSet {
  private var points = MutableList<Point>()
  public getElements() -> MutableList<Point> {
    let result = MutableList<Point>()
    for p in points {
      result.append(Point(x: p.x, y: p.y))
    }
    return result
  }
}

class PointSet {
  private var points = [Point]()
  public getElements() -> [Point] {
    return points.map { p in new Point(x:p.x, y:p.y) }
  }
}
```


Solution 2: Immutable ADTs

```
public class Point {
  let x : Int
  let y : Int
}
```

```
public struct Point {
  let x : Int
  let y : Int
}
```

```
(immutable Point)

public class Line {

  private let s : Point
  private let e : Point

  public init(s : Point, e : Point) {
    self.s = s
    self.e = s
  }

  public var start : Point {
    return self.s
  }

  ...
}
```


Deep Copy Not Needed

(assuming `Point` is a immutable ADT)

```
class PointSet {

  private var points = MutableList<Point>()

  public getElements() -> MutableList<Point> {
    let result = MutableList<Point>()
    for p in points {
      result.append(p)
    }
    return result
  }
}
```

Deep Copy Not Needed

(assuming **Point** is a immutable ADT)

```
class PointSet {

  private var points = [Point]()

  public getElements() -> [Point] {
    return points
  }
}
```


Immutability and Design

- Advantages
  - Aliasing does not matter
  - No need to make copies with identical contents
  - Rep invariants cannot be broken
- Sometimes requires different/awkward design

```
public class MutableLine {

  func move(dx: Int, dy: Int) {
    self.s = Point(self.s.x + dx, self.s.y + dy)
    self.e = Point(self.e.x + dx, self.e.y + dy)
  }
}
```

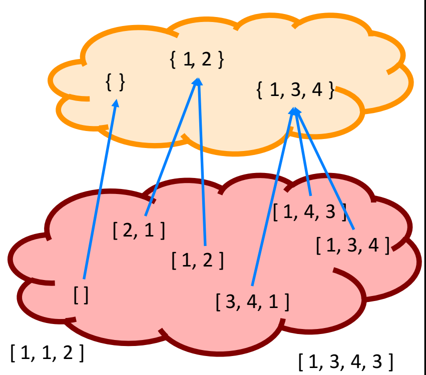

Abstract vs Concrete State of ADT

Client

```
/// ...
/// A typical IntSet
/// is { x1, ..., xn }.
class IntSet {
```

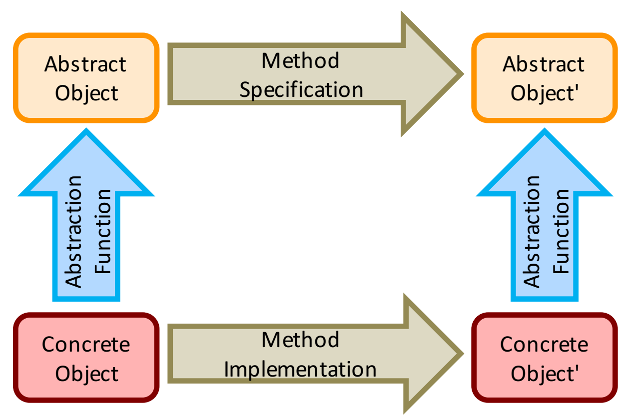
Implementer

```
class IntSet {
  var elems : [Int]
  ...
```



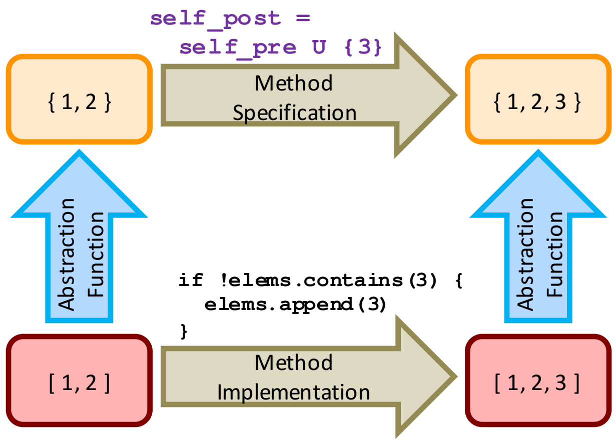
Abstraction Function: AF(self) = { x | x is contained in self.elems }

Transition Diagram



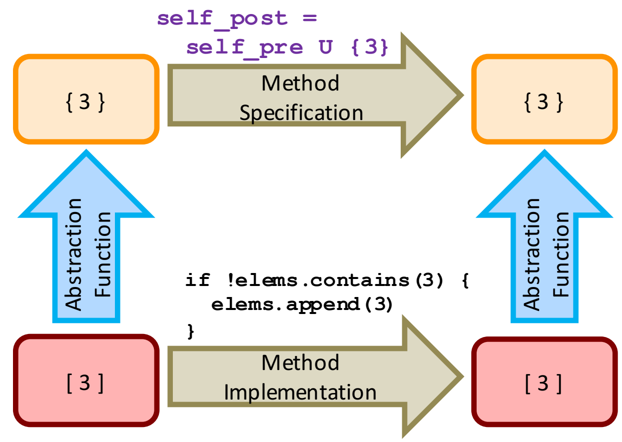
Each operation "does the right thing"

IntSet.add(3)



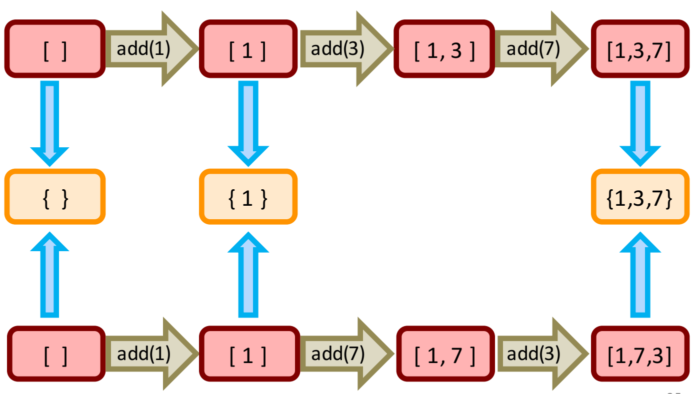
33

IntSet.add(3)



34

Abstract Equivalence



35

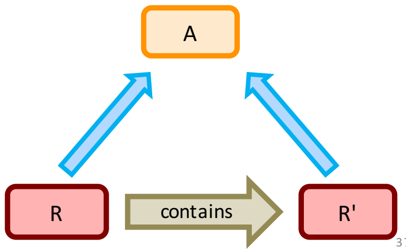
Benevolent Side Effects

```
/// - **Modifies**:
/// - Returns: true if and only if element in self
public func contains(_ element: Int) -> Bool {
    if let index = elems.index(of: element) {
      elems.swapAt(0, index)
      return true
    } else {
      return false
    }
}
```

36

Benevolent Side Effects

```
/// - **Modifies**: *still nothing*
/// - Returns: true if and only if element in self
public func contains(_ element: Int) -> Bool {
    if let index = elems.index(of: element) {
      elems.swapAt(0, index)
      return true
    } else {
      return false
    }
}
```



Writing AFs

- Domain: all concrete values satisfying Rep Inv.
- Range: Leverage math structures when possible

```
/// ...
/// A typical IntSet
/// is { x1, ..., xn }.
class IntSet {

  // AF(self) = { x | x in elems }

  // Rep Inv: No duplicates in elems

  var elems: [Int]

  ...
```


Writing AFs

- Domain: all concrete values satisfying Rep Inv.
- Range: Leverage math structures when possible

```
/**
 A Polygon is a plane figure that is bounded by a finite
 chain of at least 3 line segments closing in a loop, eg:

   (x0,y0)-(x1,y1), (x1,y1)-(x2,y2), ..., (xn,yn)-(x0,y0).

 where (x0,y0)-(x1,y1) denotes a line segment.
 */
class Polygon {

  // AF(self) = { pts[i]-pts[i+1] | i in 0..<pts.count-1 }
               U { pts[pts.count-1]-pts[0] }
  // Rep Inv: points.count >= 3
  var pts: [Point]
  ...
```


Writing AFs

- Domain: all concrete values satisfying Rep Inv.
- Range: Introduce names for pieces of abs state
  - often obvious and match public properties and observers

```
/**
 A point (x,y) on the Cartesian plan.

 **Specification Properties**:
   - x: horizontal coordinate
   - y: vertical coordinate
 */
class Point {
  // AF(self): x is self.x, y is self.y
  let x: Double
  let y: Double
  ...
```

Writing AFs

- Domain: all concrete values satisfying Rep Inv.
- Range: Introduce names for pieces of abs state

```
/**
 A URL represents the location of a resource on the network.

 **Specification Properties**:
   - protocol: either http or https
   - hostname: name of computer holding the resource
   - path: location of the resource on the host

 */
class URL {
  // AF(self): let "protocol://hostname/path" = urlString
  // Rep Inv: urlString is "well-formed"...
  let urlString : String
```

41

Make Abstract State Printable

- Domain: all concrete values satisfying Rep Inv.
- Range: Introduce names for pieces of abs state
- Write description property to show abs state:

```
class Point : CustomStringConvertible {
  // AF(self): x is self.x, y is self.y
  let x: Double
  let y: Double

  var description : String {
    return "(\(x), \(y))"
  }
  ...
```

```
let p = Point(3,4)
...
print(pt="\(p)")
// prints: pt=(3, 4)
```

42

Data Abstraction Summary

- **Rep Invariant:** Which concrete values represent abstract values?
- **Abstraction Function**: Which abstract value does a concrete value represent?

- Both are needed to reason about whether a module's implementation satisfies the specification

43