CS 326
Representation Invariants and Abstraction Functions
Stephen Freund

IntSet Spec

```swift
/// Overview: An IntSet is a mutable, unbounded set of integers. A typical IntSet is \{ x_1, \ldots, x_n \}.
class IntSet {
    /// - Returns: true if and only if element in self
    public func contains(_ element: Int) -> Bool
    /// **Modifies**: self
    /// **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)
```

Implementing a Data Abstraction (ADT)

To implement a data abstraction:
- Select the representation of instances, "the rep"
  - In Swift/Java/Python, typically instances of some class you define
- Implement operations in terms of that rep

Choose a representation so that:
- It is possible to implement required operations
- The most frequently used operations are efficient
  - But which will these be?
  - Abstraction allows the rep to change later
IntSet Implementation

```swift
class IntSet {
    private var elements = [Int]()

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

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

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

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

IntSet Rep Invariant

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

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

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

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

Rep Invariant for ADT

Client

```swift
/// ...  
/// A typical IntSet  
/// is [ x1, ..., xn ].
class IntSet {
    var elems: [Int]
    ...
}
```

Implementer

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

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

Rep Invariant Must Be Preserved

Concrete Object

```
RI(Object) ✔
```

Method Implementation

```
RI(Object') ✔
```

Concrete Object'
object.add(3)

IntSet Rep Invariant

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

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

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

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

object.add(3)

Rep Invariant Must Be Preserved
Another Example

```swift
class Account {
    var balance : Int
    // history of all transactions
    var transactions : [Transaction]
}
```

Real-world constraints:
- balance ≥ 0
- balance = ∑{t.amount | t in transactions}

Implementation-related constraints:
- forall t in transactions, t.completionDate != nil

Checking the Rep Invariant

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

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

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

IntSet V2

```swift
class IntSet {
    // Rep Invariant: elements has no duplicates
    private let elements = MutableList<Int>()
    public func contains(_ element: Int) -> Bool {
        checkRep()
        return elements.contains(element)
    }
    public func add(_ element : Int) {
        checkRep()
        if (!contains(element) {
            elements.append(element)
        }
        checkRep()
    }
    public func remove(_ element : Int) {
        checkRep()
        elements.remove(element)
        checkRep()
    }
    public func getElements() -> MutableList<Int> {
        return elements
    }
}
```

```swift
let s = IntSet()
```

```swift
s.add(5)
```

```swift
let els = s.getElements()
```

```swift
els.append(5)
```

```swift
s.remove(5)
```

```swift
assert !s.contains(5)
```

```
let s = IntSet()
```

```
let els = s.getElements()
```

```
s.append(5)
```

```
s.remove(5)
```

```
assert !s.contains(5)
```
**private Is Not Enough**

![Diagram showing concepts]

**Solution 1: Copy In. Copy Out.**

(assume Point is a mutable ADT)

```java
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)
    }
    ...
}
```

**Deep Copy**

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

```swift
class PointSet {
    private var points = MutableList<Point>()

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

**Deep Copy**

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

```swift
class PointSet {
    private var points = [Point]()

    public getElements() -> [Point] {
        return points
    }
}
```
Solution 2: Immutable ADTs

(immutable Point)

public class Line {
    private let s: Point
    private let e: Point

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

    public var start: Point {
        return self.s
    }
    ...
}

Deep Copy Not Needed

(assuming Point is immutable ADT)

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

Deep Copy Not Needed

(assuming Point is an immutable ADT)

class PointSet {
    private var points = [Point]()

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

Immutability and Design

public class Line {
    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**

```java
/// ...   
/// A typical IntSet is \{ x_1, \ldots, x_n \}.
class IntSet {
    var elems : [Int] ...
}
```

**Implementer**

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

**Abstraction Function:**

\[ AF(self) = \{ x \mid x \text{ is contained in self.elems} \} \]

### Commutativity Diagram

Each operation "does the right thing"
**Abstract Equivalence**

- Add(1)
- Add(3)
- Add(7)
- Add(1, 3, 7)
- Add(3, 7)
- Add(1, 3, 7)

**Benevolent Side Effects**

```swift
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

```swift
class IntSet {
    // AF(self) = { x | x in elems }
    // Rep Inv: No duplicates in elems
    var elems: [Int]
}
```

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.

```swift
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

```swift
/**
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
    ...
```

Make Abstract State Readable

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

```swift
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))";
    }
    // prints: p=(3, 4)
```