Reasoning About ADTs, Assertions and Exceptions

Based on material by Michael Ernst, University of Washington

Announcements

- Exam 1 on Friday October 6th
  - Closed book/phone/laptop
  - 2 cheat pages allowed (handwritten or typed)
  - 1 double-sided sheet or 2 single-sided
- Reasoning about code, Specifications, ADTs
  - Includes Reasoning about ADTs but not Assertions and Exceptions
- Review slides and practice test available off Announcements page
- Go to office hours on Wednesday and Thursday!!

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 representation really means
    - E.g., array \([2, 3, -1]\) represents \(-x^2 + 3x + 2\)
    - How the data structure is to be interpreted

Outline of Today’s Class

- Static reasoning about ADTs
  - Proving that rep invariant holds
- Dynamic “reasoning”: assertions
  - We know that a fact (e.g., rep invariant) must hold, but proving it is either hard or impossible
- Exceptions

How to Design Your Code

- The hard way: Start hacking. When something doesn’t work, hack some more
- The easier way: Plan carefully
  - Write specs, rep invariants, abstraction functions
  - Write tests (first!), reason about code, refactor
  - Less apparent progress at first, but faster completion times, better product, less frustration
How to Verify Your Code

- The hard way: hacking, make up some inputs
- An easier way: systematic testing
  - Black-box testing techniques (more next time)
  - High white-box coverage (more next time)
  - JUnit framework
- Also: reasoning, complementary to testing
  - Prove that code is correct
    - Implementation satisfies specification
    - Rep invariant is preserved
  - We will write informal proofs

Uses of Reasoning

- Goal: show that code is correct
  - Verify that the implementation satisfies its specification. Hard!
    - Forward reasoning: show that if precondition holds, postcondition holds
    - Backward reasoning: compute weakest precondition, then show stated precondition implies the weakest precondition
  - Today: prove that rep invariant always holds. (1) This is sometimes easy, sometimes hard. (2) We will use informal proofs.

Goal: Show that Rep Invariant Is Satisfied

- Prove that all objects permitted by our implementation satisfy the rep invariant
  - Sometimes easier than testing, sometimes harder. How would you “test” rep invariant?
  - You should know how to use it appropriately

Verify that Rep Invariant Is Satisfied

- We have infinitely many objects, but limited number of operations
- How do we prove all objects satisfy rep invariant?
  - Induction!
    - Consider all ways to make a new object
  - Consider all ways to modify an existing object

Benevolent Side Effects in Observers

- An implementation of observer `IntSet.contains`
  ```java
  boolean contains(int x) {
    int i = data.indexOf(x);
    if (i == -1)
      return false;
    // move-to front optimization
    // speeds up repeated membership tests
    Integer y = data.elementAt(0);
    data.set(0,x);
    data.set(i,y);
    return true;
  }
  ```
  - Mutates rep (even though it does not change abstract value), must show rep invariant still holds

Ways to Make New Objects

- Infinitely many objects but limited number of operations!
**Induction**

- **Base step**
  - Prove rep invariant holds on exit of public constructors and for objects emitted by producers
- **Inductive step**
  - Assume rep invariant of this object holds on entry of method
  - Then prove rep invariant holds on exit
  - Intuitively: there is no way to make an object, for which the rep invariant does not hold

**The IntSet ADT**

```java
/**
 * Overview: An IntSet is a mutable set of integers. E.g., { x1, x2, ..., xn }, {}.
 * There are no duplicates in the set.
 */
public class IntSet {
    // Rep invariant:
    // data has no nulls and no duplicates
    private List<Integer> data;
    public IntSet() {
        data = new ArrayList<Integer>();
    }
    public void add(int x) {
        if (!contains(x)) data.add(x);
    }
    public void remove(int x) {
        data.remove(new Integer(x));
    }
    public boolean contains(int x) {
        return data.contains(x);
    }
}
```

**Proof. IntSet Satisfies Rep Invariant**

- **Base case: constructor**
  ```java
  public IntSet() {
      data = new ArrayList<Integer>();
  }
  Rep invariant trivially holds
  ```

- **Inductive step: for each method**
  - Assume rep invariant holds on entry
  - Prove rep invariant holds on exit

**Inductive Step, contains**

- Rep invariant: data has no nulls and no duplicates
- ```java
  public boolean contains(int x) {
      return data.contains(x);
  }
  ```
- List.contains does not change data, so neither does IntSet.contains
  - Therefore, rep invariant is preserved.
- Why do we even need to check contains?

**contains with Benevolent Side Effects**

```java
boolean contains(int x) {
    int i = data.indexOf(x);
    if (i == -1) return false;
    // move-to-front optimization
    // speeds up repeated membership tests
    Integer y = data.elementAt(0);
    data.set(0, x);
    data.set(i, y);
    return true;
}
```
**Inductive Step, remove**

Rep invariant: data has no nulls and no duplicates

```java
public void remove(int x) {
    data.remove(new Integer(x));
}
```

- **ArrayList.remove** has two behaviors
  - Leaves **data** unchanged
  - Removes an element
    - (Only addition can violate rep invariant)

Therefore, rep invariant is preserved

**Inductive Step, add**

Rep invariant: data has no nulls and no duplicates

```java
public void add(int x) {
    if (!contains(x))
        data.add(x);
}
```

- Case 1: \(x\) is in **data**
  - **data** is unchanged, thus rep invariant is preserved
- Case 2: \(x\) is not in **data**
  - New element is not null or a duplicate, thus rep invariant holds at exit

**Reasoning About Rep Invariant**

- Inductive step must consider all possible changes to the rep!

**Review Problems: IntMap Specification**

The Overview:

```java
/** An IntMap is a mapping from integers to integers.
 * It implements a subset of the functionality of Map<int,int>.
 * All operations are exactly as specified in the documentation
 * for Map.
 *
 * @specfield pairs = { <k1, v1>, <k2, v2>, <k3, v3>, ... }
 */
```

**Review Problems: IntStack Specification**

```java
/**
 * An IntStack represents a stack of ints.
 * It implements a subset of the functionality of Stack<int>.
 * All operations are exactly as specified in the documentation
 * for Stack.
 *
 * @specfield stack = [a_0, a_1, a_2, ..., a_k]
 */
```
**Review Problems: IntStack Specification**

```java
interface IntStack {
    /** Pushes an item onto the top of this stack.
     * If stack_pre = [a_0, a_1, a_2, ..., a_(k-1), a_k]
     * then stack_post = [a_0, a_1, a_2, ..., a_(k-1), a_k, val]. */
    void push(int val);

    /** Removes the int at the top of this stack and returns that int.
     * If stack_pre = [a_0, a_1, a_2, ..., a_(k-1), a_k]
     * then stack_post = [a_0, a_1, a_2, ..., a_(k-1)]
     * and the return value is a_k. */
    int pop();
}
```

**Review Problems: IntStack using IntMap**

```java
class Willy'sIntStack implements IntStack {
    private IntMap theRep;
    private int size = 0;

    public void push(int val) {
        size = size+1;
        theRep.put(size, val);
    }

    public int pop() {
        int val = theRep.get(size);
        theRep.remove(size);
        size = size–1;
        return val;
    }
}
```

**Review Problems: Reasoning about WillyIntStack**

- **Base case**: Prove rep invariant holds on exit of constructor
- **Inductive step**: Prove that if rep invariant holds on entry of method, it holds on exit of method
  - push
  - pop

For brevity, we ignore empty/full stack

**Review Problems: Willy's IntStack**

```java
class IntStack {
    private IntMap theRep = new IntMap();
    private int size = 0;

    public void push(int val) {
        size = size+1;
        theRep.put(size, val);
    }

    public int pop() {
        int val = theRep.get(size);
        theRep.remove(size);
        size = size–1;
        return val;
    }
}
```

**Review Problems: WillyIntMap using IntStack**

```java
class WillysIntMap implements IntMap {
    private IntStack theRep;

    public boolean put(int key, int value) { ... }
    public void remove(int key) { ... }
    public boolean containsKey(int key) { ... }
    public int get(int key) { ... }
}
```

**Review Problems: Willy's IntStack**

- Willy decided to add this new method
  ```java
  public Map elements() {
      return theRep;
  }
  ```

  He classified it as observer and went on to prove that it preserved the rep invariant
  - What can go wrong?
Practice Defensive Programming

- Check
  - Precondition
  - Postcondition
  - Rep invariant
  - Other properties we know must hold
- Check \textit{statically} via reasoning
  - "Statically" means before execution
  - Works in simpler cases (the examples we saw), can be difficult in general
  - Motivates us to simplify and/or decompose our code!

Check \textit{dynamically} via assertions

- What do we mean by "dynamically"?
  - At run time
- Assertions, supported by Java since 1.4
  - `assert index >= 0;`
  - `assert coeffs.length-1 == degree : "Bad rep"`
  - `assert coeffs[degree] != 0 : "Bad rep"`
- Write assertions, as you write code
- Aside: not to be confused with JUnit method such as `assertEquals`

**Assertions**

- \texttt{java} runs with assertions disabled (default)
- \texttt{java -ea} runs Java with assertions enabled
- Always enable assertions during development. Turn off in rare circumstances

```
assert (index >= 0) && (index < names.length);
```

**When NOT to Use Assertions**

- Useless:
  - `x = y+1; assert x == y+1;`
- When there are side effects
  - `assert list.remove(x);`
  - // Better:
    - `boolean found = list.remove(x); assert found;`
- How can you test at runtime whether assertions are enabled?

Outline of Today’s Class

- Static reasoning about ADTs
  - Proving rep invariants
- Dynamic reasoning: assertions
- Exceptions
  - Basics
  - Uses of exceptions

Failure

Some causes of failure
1. Misuse of your code
   - Precondition violation
2. Errors in your code
   - Bugs, rep exposure, many more
3. Unpredictable external problems
   - Out of memory
   - Missing file
   - Memory corruption
What to Do When Something Goes Wrong?

- Fail friendly, fail early to prevent harm

Goal 1: Give information
- To the programmer, to the client code

Goal 2: Prevent harm
- Abort: inform a human, cleanup, log error, etc.
- Retry: problem might be temporary
- Skip subcomputation: permit rest of program to continue
- Fix the problem (usually infeasible)

Preconditions vs. Exceptions

- A precondition prohibits misuse of your code
  - Adding a preconditions weakens the spec
  - A precondition ducks the problem
  - Behavior of your code when precondition is violated is unspecified!
  - Does not help clients violating precondition of your code
  - Removing the precondition requires specifying the behavior. Strengthens the spec
  - Example: specify that an exception is thrown

Which One Is Better?

Choice 1:
// modifies: this
// effects: removes element at index from this
// throws: IndexOutOfBoundsException if index < 0 || index >= this.size
public void remove(int index) {
    if (index >= size() || index < 0)
        throw new IndexOutOfBoundsException("Info...");
    else
        // remove element at index from collection
}

Choice 2:
// requires: 0 <= index < this.size
// modifies: this
// effects: removes element at index from this
public void remove(int index) {
    // no check, remove element at index
}

Square Root, With Precondition and Assertions

// requires: x >= 0
// returns: approximation to square root of x
public double sqrt(double x) {
    assert x >= 0 : "Input must be >=0";
    double result;
    ... // compute result
    assert(Math.abs(result*result - x) < .0001);
    return result;
}

Square root, Specified for All Inputs

// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
public double sqrt(double x) throws IllegalArgumentException {
    double result;
    if (x < 0)
        throw new IllegalArgumentException("...");
    ... // compute result
    return result;
}
Better: Square root, Specified for All Inputs

Client code:
```java
try {
    y = sqrt(-1);
} catch (IllegalArgumentException e) {
    e.printStackTrace();  // or take other action
}
```

Exception is handled by catch block associated with nearest dynamically enclosing try
Top-level handler: print stack trace, terminate program

Throwing and Catching

- Java maintains a call stack of methods that are currently executing
- When an exception is thrown, control transfers to the nearest method with a matching catch block
  - If none found, top-level handler
- Exceptions allow non-local error handling
  - A method far down the call stack can handle a deep error!

Propagating an Exception up the Call Chain

```java
// throws: IllegalArgumentException if no real solution exists
// returns: x such that ax^2 + bx + c = 0
double solveQuad(double a, double b, double c)
    throws IllegalArgumentException {
    ... // no need to catch it here
    return (-b + sqrt(b*b - 4*a*c))/(2*a);
}
```

Informing Client of a Special Outcome

- Special value
  - null – Map.get(x)
  - -1 – List.indexOf(x)
  - NaN – sqrt of negative number
- Special value are difficult to use
  - Hard to distinguish from real values
  - Error-prone: programmer forgets to check result?
  - The value is illegal and will cause problems later
- Ugly
- Better solution: exceptions

Two Distinct Uses of Exceptions

- Failure
  - Unexpected by your code (code is written so that failure does not happen)
  - Usually unrecoverable. If condition is left unchecked, exception propagates up the stack
- Special results
  - Expected by your code
  - Unknowable for client of your code
  - Always check and handle locally. Take special action and continue computing

Java Exceptions: Checked vs. Unchecked Exceptions

- Checked exceptions. For special results
  - Library: must declare in signature
  - Client: must either catch or declare in signature
  - It is guaranteed there is a dynamically enclosing catch
- Unchecked exceptions. For failures
  - Library: no need to declare
  - Client: no need to catch
  - RuntimeException and Error
Java Exception Hierarchy

Don’t Ignore Exceptions

- An empty catch block is poor style!
- Often done to hide an error or get to compile

```java
try {
    readFile(filename);
} catch (IOException e) {} // do nothing on error
```

- At a minimum, print the exception

```java
} catch (IOException e) {
    e.printStackTrace();
}
```

Exceptions, review

- Use an exception when
  - Checking the condition is feasible
  - Implementation is used in a broad or unpredictable context

- Use a precondition when
  - Checking would be prohibitive
  - Implementation is used in a narrow context in which calls can be checked

- Avoid preconditions because
  - Program can fail in an uninformative or dangerous way if caller violates the precondition
  - Program should fail as early as possible to prevent harm

- Use checked exceptions most of the time
- Handle exceptions sooner than later