Principles of Software Construction: Objects, Design, and Concurrency

Functional Correctness – A Broader Perspective

Christian Kästner
Charlie Garrod

© 2012-14 C Kaestner, C Garrod, J Aldrich, and W Scherlis
Learning Goals

- Writing bug reports
- Apply Hoare-style verification to object-oriented programs
- Reason about inheritance with behavioral subtyping
- Apply static analysis tools
- Understand the tradeoffs among testing, formal verification and static analysis
Bug Reports
Reporting Defects

- Reproducible defects
  - Easier to find and fix
  - Easier to validate
  - Increased confidence

- Simple and general
  - More value doing the fix

- Non-antagonistic
  - State the problem
  - Don't blame
Social Issues in Defect Reporting

- There are differences between developer and tester culture
- Acknowledge that testers often deliver bad news
- Work hard to detect defects locally
  - Easier to narrow scope and responsibility
  - Less adversarial
- Don’t measure performance in terms of defect reports
Defect Tracking

- Always track defects and issues
- Issue: Bug, feature request, or query
  - May not know which of these until analysis is done, so track in the same database (Bugzilla, github)
- Provides a basis for measurement
- Provides a basis for division of effort
- Facilitates communication
  - Organized record for each issue
  - Ensures problems are not forgotten
Bug Tracking on GitHub

• Every GitHub project has its own issue tracker (and wiki); enable in project settings
Formal Verification of Object-Oriented Programs
Formal Verification

- Proving the correctness of an implementation with respect to a formal specification, using formal methods of mathematics.
- Formally prove that all possible executions of an implementation fulfill the specification.
- Manual effort; partial automation; not automatically decidable.
Formal Specifications

```java
/*@ requires len >= 0 && array != null && array.length == len; */
@ ensures \result == \sum int j; 0 <= j && j < len; array[j];
int total(int array[], int len);
```

Advantage of formal specifications:

* runtime checks for free
* basis for formal verification
* assisting automatic analysis tools

JML (Java Modelling Language) as specifications language in Java (inside comments)
Recap: Hoare-Style Verification

• Formal reasoning about program correctness using pre- and postconditions

• Syntax: \{P\} S \{Q\}
  - P and Q are predicates
  - P is the precondition
  - S is a program
  - Q is the postcondition

• Semantics
  - If we start in a state where P is true and execute S, then S will terminate in a state where Q is true
Recap: Hoare-Logic Rules

Assignments
{ P[E/x] } x:= E { P }

Composition
{ P } S { Q } { Q } T { R }
----------------------------------
{ P } S; T { R }

If statement
{ B & P } S { Q } { !B & P } T { Q }
----------------------------------------------
{ P } if (B) S else T { Q }

While loop with loop invariant P
{ P & B } S { P }
-----------------------------------
{ P } while (B) S { !B & P }

Consequence
P -> P' { P } S { Q } Q -> Q'
--------------------------------------------
{ P' } S { Q' }
Hoare Triples – Examples

- \{ \text{true} \} \ x := 5 \ \{ \ \}
- \{ \ \} \ x := x + 3 \ \{ \ x = y + 3 \ \} 
- \{ \ \} \ x := x * 2 + 3 \ \{ \ x > 1 \ \} 
- \{ x=a \} \text{ if } (x < 0) \text{ then } x := -x \ \{ \ \} 
- \{ \text{false} \} \ x := 3 \ \{ \ \} 
- \{ x < 0 \} \text{ while } (x!=0) \ x := x-1 \ \{ \ \}
Hoare Triples – Examples

• \{ \text{true} \} \ x := 5 \ \{ \ x = 5 \ \} 

• \{ \ x = y \} \ x := x + 3 \ \{ \ x = y + 3 \ \} 

• \{ \ x > -1 \} \ x := x * 2 + 3 \ \{ \ x > 1 \ \} 

• \{ \ x = a \} \ \text{if} \ (x < 0) \ \text{then} \ x := -x \ \{ \ x = |a| \ \} 

• \{ \ \text{false} \} \ x := 3 \ \{ \ x = 8 \ \} 

• \{ \ x < 0 \} \ \text{while} \ (x != 0) \ x := x-1 \ \{ \ \} 
  \text{• no such triple!}
Recap: 122 midterm

```c
int find_peak_bin(int[] A, int n)
//@requires 0 < n && n <= length(A);
//@requires is_peaked(A, 0, n);
//@ensures 0 <= result && result < n;
//@ensures gt_seg(A[result], A, 0, result);
//@ensures gt_seg(A[result], A, result+1, n);
{
int lower = 0;
int upper = n-1;
while (lower < upper)
//@loop_invariant ____________________________ ;
//@loop_invariant ____________________________ ;
{
int mid = lower + (upper-lower)/2;
//@assert ______________ ; /* optional */
if (A[mid] < A[mid+1])
  lower = mid+1;
else // @assert ______ ; /* optional */
  upper = mid;
}
//@assert _________________________ ; /* optional */
return lower;
```
Class Invariants

- Properties about the fields of an object
- Established by the constructor
- Should always hold before and after execution of public methods
- May be invalidated temporarily during method execution

```java
public class SimpleSet {
    int contents[];
    int size;

    //@ ensures sorted(contents);
    SimpleSet(int capacity) { ... }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean add(int i) { ... }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean contains(int i) { ... }
}
```

```java
public class SimpleSet {
    int contents[];
    int size;

    //@invariant sorted(contents);
    SimpleSet(int capacity) { ... }

    boolean add(int i) { ... }

    boolean contains(int i) { ... }
}
```
Behavioral Subtyping (Liskov Substitution Principle)

Let $q(x)$ be a property provable about objects $x$ of type $T$. Then $q(y)$ should be provable for objects $y$ of type $S$ where $S$ is a subtype of $T$.

Barbara Liskov

- An object of a subclass should be substitutable for an object of its superclass

- Known already from types:
  - May use subclass instead of superclass
  - Subclass can add, but not remove methods
  - Overriding method must return same or subtype
  - Overriding method may not throw additional exceptions

- Applies more generally to behavior:
  - A subclass must fulfill all contracts that the superclass does
  - Same or stronger invariants
  - Same or **stronger** postconditions for all methods
  - Same or **weaker** preconditions for all methods
Behavioral Subtyping (Liskov Substitution Principle)

abstract class Vehicle {
    int speed, limit;
    //@ invariant speed < limit;

    //@ requires speed != 0;
    //@ ensures |speed| < |old{speed}|
    void break();
}

class Car extends Vehicle {
    int fuel;
    boolean engineOn;
    //@ invariant fuel >= 0;

    //@ requires fuel > 0 && ! engineOn;
    //@ ensures engineOn;
    void start() { … }

    void accelerate() { … }

    //@ requires speed != 0;
    //@ ensures |speed| < |old{speed}|
    void break() { … }
}

Subclass fulfills the same invariants (and additional ones)
Overridden method has the same pre and postconditions
Behavioral Subtyping (Liskov Substitution Principle)

class Car extends Vehicle {
    int fuel;
    boolean engineOn;
    //@ invariant fuel >= 0;

    //@ requires fuel > 0 && ! engineOn;
    //@ ensures engineOn;
    void start() { ... }

    void accelerate() { ... }

    //@ requires speed != 0;
    //@ ensures |speed| < |\old{speed}|
    void break() { ... }
}

class Hybrid extends Car {
    int charge;
    //@ invariant charge >= 0;

    //@ requires (charge > 0 || fuel > 0)
        && ! engineOn;
    //@ ensures engineOn;
    void start() { ... }

    void accelerate() { ... }

    //@ requires speed != 0;
    //@ ensures |speed| < |\old{speed}|
    //@ ensures charge > \old{charge}
    void break() { ... }
}

Subclass fulfills the same invariants (and additional ones)
Overridden method start has weaker precondition
Overridden method break has stronger postcondition
Behavioral Subtyping (Liskov Substitution Principle)

class Rectangle {
    int h, w;
    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }
    //methods
}

class Square extends Rectangle {
    Square(int w) {
        super(w, w);
    }
}

Is Square a behavior subtype of Rectangle?
Behavioral Subtyping (Liskov Substitution Principle)

```java
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //methods
}

class Square extends Rectangle {
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}

Is Square a behavior subtype of Rectangle?
Behavioral Subtyping (Liskov Substitution Principle)

```java
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h = h; this.w = w;
    }

    void scale(int factor) {
        w = w * factor;
        h = h * factor;
    }
}

class Square extends Rectangle {
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}
```

Is Square a behavior subtype of Rectangle?
Behavioral Subtyping (Liskov Substitution Principle)

class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }

    void setWidth(int neww) {
        w=neww;
    }
}

class Square extends Rectangle {
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}

Is Square a behavior subtype of Rectangle?
Behavioral Subtyping (Liskov Substitution Principle)

class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }

    void setWidth(int neww) {
        w=neww;
    }
}

class Square extends Rectangle {
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}

class GraphicProgram {
    void scaleW(Rectangle r, int factor) {
        r.setWidth(r.getWidth() * factor);
    }
}

With these methods, Square is not a behavior subtype of Rectangle
Formal Verification of Object-Oriented Programs

- Analogue to verification of imperative programs
- Class invariants simplify specifications
- Behavioral subtyping ensures substitutability

- Proof of correctness
  - All possible executions will fulfill the formal specifications
  - Pen and paper proof
  - Support for partially automated proofs available (full automation not possible)
Static Analysis
public class CartesianPoint {
    private int x, y;
    int getX() { return this.x; }
    int getY() { return this.y; }
    boolean equals(CartesianPoint that) {
        return (this.getX() == that.getX()) &&
               (this.getY() == that.getY());
    }
}
FindBugs

public boolean equals(CartesianPoint p)
{
    return (p.x==this.x) && (p.y==this.y);
}

0 errors, 2 warnings, 0 others

Description

FindBugs Problem (Of concern) (1 item)
   CartesianPoint defines equals and uses Object.hashCode()

FindBugs Problem (Scary) (1 item)
   CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)

Bug Info

CartesianPoint.java: 12

Bug: CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)

This class defines a covariant version of the equals() method, but inherits the normal equals(Object) method defined in the base java.lang.Object class. The class should probably define a boolean equals(Object) method.

Confidence: Normal, Rank: Scary (8)
Pattern: EQ_SELF_USE_OBJECT
Type: Eq, Category: CORRECTNESS (Correctness)
public final class CartesianPoint {

    private int X, Y;

    CartesianPoint(int x, int y) {
        this.X = x;
        this.Y = y;
    }

    public int getY() {
        return Y;
    }

    public int getX() {
        return X;
    }

}
Static Analysis

- Analyzing code without executing it (automated inspection)
- Looks for bug patterns
- Attempts to formally verify specific aspects
- Point out typical bugs or style violations
  - NullPointerExceptions
  - Incorrect API use
  - Forgetting to close a file/connection
  - Concurrency issues
  - And many, many more (over 250 in FindBugs)
- Integrated into IDE or build process
- FindBugs and CheckStyle open source, many commercial products exist
Example FindBugs Bug Patterns

- Correct equals()
- Use of ==
- Closing streams
- Illegal casts
- Null pointer dereference
- Infinite loops
- Encapsulation problems
- Inconsistent synchronization
- Inefficient String use
- Dead store to variable
Bug finding

```java
public Boolean decide() {
    if (computeSomething()==3)
        return Boolean.TRUE;
    if (computeSomething()==4)
        return false;
    return null;
}
```

**Bug:** FBTest.decide() has Boolean return type and returns explicit null

A method that returns either Boolean.TRUE, Boolean.FALSE or null is an accident waiting to happen. This method can be invoked as though it returned a value of type boolean, and the compiler will insert automatic unboxing of the Boolean value. If a null value is returned, this will result in a NullPointerException.

**Confidence:** Normal, **Rank:** Troubling (14)
**Pattern:** NP_BOOLEAN_RETURN_NULL
**Type:** NP, **Category:** BAD_PRACTICE (Bad practice)
Abstract Interpretation

• Static program analysis is the **systematic examination** of an **abstraction of a program’s state space**

• Abstraction
  ▪ Don’t track everything! (That’s normal interpretation)
  ▪ Track an important abstraction

• Systematic
  ▪ Ensure everything is checked in the same way

Details on how this works in 15-313
Comparing Quality Assurance Strategies
<table>
<thead>
<tr>
<th></th>
<th>Error exists</th>
<th>No error exists</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Error Reported</strong></td>
<td>True positive (correct analysis result)</td>
<td>False positive (annoying noise)</td>
</tr>
<tr>
<td><strong>No Error Reported</strong></td>
<td>False negative (false confidence)</td>
<td>True negative (correct analysis result)</td>
</tr>
</tbody>
</table>

**Sound Analysis:**
- reports all defects
  - no false negatives
  - typically overapproximated

**Complete Analysis:**
- every reported defect is an actual defect
  - no false positives
  - typically underapproximated

**How does testing relate? And formal verification?**
Defects reported by Sound Analysis

All Defects

Defects reported by Complete Analysis

Unsound and Incomplete Analysis
The Bad News: Rice's Theorem

"Any nontrivial property about the language recognized by a Turing machine is undecidable."

Henry Gordon Rice, 1953

- Every static analysis is necessarily incomplete or unsound or undecidable (or multiple of these)
- Each approach has different tradeoffs
Soundness / Completeness / Performance Tradeoffs

- Type checking does catch a specific class of problems (sound), but does not find all problems.

- Compiler optimizations must err on the safe side (only perform optimizations when sure it's correct; -> complete).

- Many practical bug-finding tools analyses are unsound and incomplete:
  - Catch typical problems
  - May report warnings even for correct code
  - May not detect all problems

- Overwhelming amounts of false negatives make analysis useless.

- Not all "bugs" need to be fixed.
Testing and Proofs

- **Testing**
  - Observable properties
  - Verify program for one execution
  - Manual development with automated regression
  - Most practical approach now
  - Does not find all problems (unsound)

- **Proofs (Formal Verification)**
  - Any program property
  - Verify program for all executions
  - Manual development with automated proof checkers
  - Practical for small programs, may scale up in the future
  - Sound and complete, but not automatically decidable

- So why study proofs if they aren’t (yet) practical?
  - Proofs tell us how to think about program correctness
  - Important for development, inspection, dynamic assertions
  - Foundation for static analysis tools
  - These are just simple, automated theorem provers
  - Many are practical today!
Testing, Static Analysis, and Proofs

• **Testing**
  - Observable properties
  - Verify program for one execution
  - Manual development with automated regression
  - Most practical approach now
  - Does not find all problems (unsound)

• **Static Analysis**
  - Analysis of all possible executions
  - Specific issues only with conservative approx. and bug patterns
  - Tools available, useful for bug finding
  - Automated, but unsound and/or incomplete

• **Proofs (Formal Verification)**
  - Any program property
  - Verify program for all executions
  - Manual development with automated proof checkers
  - Practical for small programs, may scale up in the future
  - Sound and complete, but not automatically decidable

What strategy to use in your project?
Quality Assurance Summary

- Reporting and tracking bugs/issues
- Select a quality assurance strategy for functional correctness
- Testing can find faults in specific executions
- Formal verification (Hoare-style pre/post-conditions) can ensure correctness of all executions
  - Class Invariants and Behavioral Subtyping
- Static analysis can find issues for classes of problems
- Soundness vs. Completeness vs. Automation