EQ: Checking the Implementation of Equality in Java

Chandan R. Rupakheti, Daqing Hou
Department of Electrical and Computer Engineering
Clarkson University, Potsdam, New York 13699-5722
{*rupakhcr, dhou*}*@clarkson.edu*

***Abstract*—Objects in Object-Oriented languages such as Java are required to implement an equality predicate using the `equals(Object)` method in order to be compared with each other. This is particularly important when these objects interact with lists, sets, and maps from the Java Collection Framework. There are several considerations that must be taken in the implementation of this method, which, if ignored, will lead to subtle bugs. We present a tool called EQ that analyzes the source code to find such bugs.**

***Keywords*-EQ; Object Equality; Abstraction Recognition; Path-Based Analysis; Model Finding; Soot; Alloy; Java**

I. Introduction

Java objects that interact with the Collection Framework are expected to implement a predicate for equality comparison by overriding `equals(Object)`. Such implementations must respect the three properties of equivalence relation [1]: *reflexivity*, *symmetry*, and *transitivity.* And they must return `false` when `null` is passed as an argument.

The `equals()` methods become especially error-prone when the type hierarchy containing them evolves. To get them right, a programmer must simultaneously take into account several factors, including type testing, state representation, and the evolution of the type hierarchy. During evolutionary development, programmers often forget about their previous intent about a type hierarchy while adding a new type. As a result, they create bugs that can be hard to detect even for an experienced programmer [1], [9].

Existing checkers handle equality related bugs inadequately (Section IV). To address this, we have developed a tool called EQ for finding bugs from `equals`. The key to EQ's design is to model the equality logic and the correctness specification in Alloy and use the Alloy Analyzer to find counterexamples [4]. Because in practice, objects to be considered equal are almost always assigned to the same type hierarchy, EQ produces one Alloy model per type hierarchy. [2] In this paper, we offer only a high-level overview of EQ's design (Section II), but full details can be found in [10]. Our focus is to demonstrate EQ's usefulness and how it can be used to detect subtle equality bugs (Section III).

II. Approach Overview

EQ processes one type hierarchy at a time to produce an Alloy model. For each `equals()` method in the hierarchy, EQ first enumerates all of its inter-procedural paths. These paths are then passed through a pipeline of pattern detectors, which identify equality related abstractions in the paths. Finally, the identified patterns, or abstractions, are translated to Alloy. EQ invokes the Alloy Analyzer to process the model, and counterexamples are reported inside Eclipse.

Six pattern detectors have been developed. These patterns cover *type testing*, *simple state comparison* (e.g., fields), and *composite state comparison* (e.g., array, list, set, and map comparisons). For example, the code in Listing 1 is automatically translated to the Alloy model in Listing 2. In this case, the *simple state* detector translates lines 5 and 8 in Listing 1 to lines 11 (also 12) and 14 in Listing 2, respectively. Similarly, the *type testing detector* translates line 6 of Listing 1 to line 13 of Listing 2.

EQ is designed as a plugin. It uses three frameworks: Soot [11], Eclipse's Java Development Tools (JDT) [3], and Alloy [4]. JDT is used to search classes that override `equals` and to construct the type hierarchies for them.

III. Case Study

In this section, we demonstrate EQ in an evolutionary development where `equals()` is co-evolved with a `Point` type hierarchy. This example has been used in previous research to show common problems in `equals()` [1], [6].

A. A Single Type

Consider the `Point` class in Listing 1. In general, implementations of `equals` comprise of a type test (line 6) followed by state comparisons (line 8). To guard against the possibility of the parameter being `null`, a nullity test is also performed (line 5).

Listing 1. The `Point` class.

```
1  public class Point {
2   private double x, y;
3   public Point(double x, double y) {this.x = x; this.y = y;}
4   public boolean equals(Object o) {
5    if(o == null) return true; // Nullity test
6    if (o.getClass() != Point.class) return false; // Type test
7    Point that = (Point)o;
8    return this.x==that.x&&this.y==that.y; // State comparison
9   } ... // Getters and other operations
10 }
```

[1]http://tinyurl.com/java-equals. All URLs verified on July 1, 2011.

[2]For performance, EQ considers only type hierarchies below `Object`.

[3]http://eclipse.org/jdt/

Given the implementation of Listing 1, EQ produces the Alloy model of Listing 2. Lines 1-4 in Listing 2 show the definition for the type hierarchy in Alloy. `Object` (line 1) is the root of the type hierarchy. It is inherited by `Point` (line 4). The reference for a Java object is modeled using a `ref` field in the `Object` signature. The fact at line 2 specifies that two objects are the same thing if they have the same reference. The fact at line 3 specifies that all *non-null* objects have positive reference. The `Null` signature at line 6 models `null` in Java with `ref` equal to `0` (line 7).

Listing 2. Alloy model for `Point`'s type hierarchy.

```
1  abstract sig Object {ref : Int} // Type hierarchy root
2  fact{all a, b : Object | (a.ref = b.ref) => (a = b)}
3  fact{all a : Object − Null | a.ref > 0}
4  sig Point extends Object {x : Int, y : Int}
5
6  lone sig Null extends Object {} // Nullity modeling
7  fact{all a : Null | a.ref = 0}
8
9  pred Object :: equals(that: Object) { // Equality Predicate
10  (this in Point) =>
11    (that.ref = 0) or  // Condition on the 1st true path
12    ((that.ref != 0) and // Conditions on the 2nd true path
13    (that in Point) and
14    (this.x = that.x) and (this.y = that.y))
15 }
16 // Properties of equivalence relation
17 assert nullity{all a:Object−Null|all b:Null|!a.equals[b]}
18 assert reflexive{all a:Object−Null|a.equals[a]}
19 assert symmetric{all a,b:Object−Null|a.equals[b]<=>b.equals[a]}
20 assert transitive{all a,b,c:Object−Null|a.equals[b] and b.equals[c]
21                                              => a.equals[c]}
22 // Checking the four properties in small scopes
23 check nullity for 2
24 check reflexive for 2
25 check symmetric for 3
26 check transitive for 4
```

Since equality is a predicate, only true-returning paths are translated into Alloy. There are two true-returning paths in Listing 1: [5] and [!5, !6, 7, 8], where the nodes prefixed with '!' represent the false branches. The predicates detected on a path are conjoined to make a conjunctive formula. Formulas for multiple paths are further disjoined to make a disjunction (Listing 2). Finally, if a type hierarchy contains multiple concrete types, the disjunctions for all `equals` are joined with mutually exclusive implications (e.g., Listing 4, lines 2 and 6) under one `equals` predicate in the model.

The specification for the equivalence relation is shown at lines 17-20 of Lisiting 2. Lines 23-26 are Alloy commands for checking these properties. Note that the small scopes are sufficient to reveal violations of these properties [4].

Alloy Analyzer reports the violation of *nullity* as shown in the counterexample of Figure 1. The counterexample means that for a non-null object `a` (`ref=6`), of type `Point(x=6, y=-8)`, its `equals` will *not* return `false` when a `Null` (`ref=0`) is passed as an argument. This is

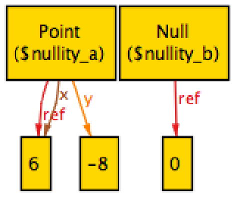


Figure 1. An Alloy Analyzer counterexample for the violation of nullity in `Point`.

caused by the typographical error at line 5 of Listing 1, which should have returned `false`. After this is fixed, Alloy Analyzer reports no more violations. Note that this is not an artificial error. In fact, the `equals` method of `org.jfree.data.xy.WindDataItem` in *JFreeChart 1.0.13* has a similar error.

B. Subtyping

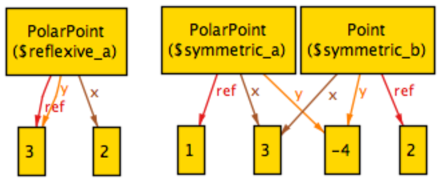
Let us extend the cartesian `Point` by a representation in the polar coordinates (Listing 3). `PolarPoint` uses the same internal representation of `Point` (i.e., `x` and `y`) to expose a new state (i.e., `r` and θ). It also inherits the implementation of `Point.equals()` for equality. When EQ is run again, the type hierarchy definition of Listing 2 will be updated to include `PolarPoint` and the `equals` predicate will change as shown in Listing 4. The rest of the model remains the same.

Listing 3. A `PolarPoint` that uses `Point`'s internal representation.

```
public class PolarPoint extends Point {
 public PolarPoint(double r, double theta)
 {super(r ∗ Math.cos(theta), r ∗ Math.sin(theta));}
 public double getTheta() {return Math.atan2(this.y, this.x);}
 public double getR() {return ...;}
}
```

Listing 4. Alloy model for the new `Point`'s type hierarchy.

```
1 pred Object :: equals(that: Object) {
2  (this in Point − PolarPoint ) => // Definition for Point
3    ((that.ref != 0) and // Constraints on the true path
4    (that in Point − PolarPoint) and
5    (this.x = that.x) and (this.y = that.y))
6  else (this in PolarPoint) => // Definition for PolarPoint
7    ... // Same as Point's due to inheritance
8 }
```



(a) Reflexivity (b) Symmetry

Figure 2. The violations related to `Point` and `PolarPoint`.

Alloy Analyzer now reports the violations of *reflexivity* and *symmetry* in Figures 2(a) and 2(b). The meaning of the counterexamples can be illustrated as follows:

```
PolarPoint a = new PolarPoint(...); // (r, theta) for (x=2, y=3)
a.equals(a); // returns false (Broken reflexivity)

PolarPoint a = new PolarPoint(...); // (r, theta) for (x=3, y=−4)
Point b = new Point(3, −4);
a.equals(b); // returns true
b.equals(a); // returns false; hence symmetry broken.
```

These violations are caused by the rigid `Point.class` type testing at line 6 of Listing 1 which allows only `Point` but not its subclasses to pass the type test. Since `PolarPoint` is intended to be a subtype of `Point`, using an `o instanceof Point` type testing expression will allow both `Point` and `PolarPoint` to form an equivalence relation, thus allowing for inter-class equality. We call such a hierarchy *type-compatible* [9].

The inheritance of `Point` by `PolarPoint` in this case is an example of subtyping where a `PolarPoint` can be substituted for a `Point`. However, inheritance may be used not only for subtyping but also for implementation reuse.

C. Implementation Reuse

Listing 5. Implementation of `Point3D`.

```
1 public class Point3D extends Point {
2  private double z;
3  ... // Constructor, getters and other operations
4  public boolean equals(Object o) {
5   if(!(o instanceof Point3D)) return false;
6   Point3D that = (Point3D)o;
7   return super.equals(o) && this.z == that.z;
8  }
9 }
```

Listing 6. Predicate for `Point` type hierarchy involving Listings 1 and 5.

```
1 pred Object :: equals(that: Object) {
2  (this in Point − Point3D) => // For Point
3   ((that in Point) and (that.ref != 0) and
4   (this.x = that.x) and (this.y = that.y))
5  else (this in Point3D) => // For Point3D
6   ((that in Point3D) and (that.ref != 0) and
7   (this.x = that.x) and (this.y = that.y) and (this.z = that.z))
8 }
```

Consider the `Point3D` class (Listing 5) that extends `Point` by adding a `z`-dimension. Given the type hierarchy of `Point` and `Point3D`, EQ generates the equality predicate shown in Listing 6. Note that in this case we assume that the type testing logic for `Point` (lines 5 and 6 of Listing 1) has been replaced by `if(!(o instanceof Point)) return false;`. The `instanceof` test at line 5 of Listing 5 also guarantees the non-nullness as expressed at line 6 of Listing 6. Alloy Analyzer now reports a violation of symmetry. The violation is caused by the `instanceof` type testing expression in the two classes. A `Point3D` is an instance of a `Point` but not vice versa, causing the violation. In fact, this is the most common violation found in *JDK 1.5*, *JFreeChart 1.0.13*, *Apache Lucene 3.0*, and *Apache Tomcat 6.0*. The results of running EQ on these projects can be found online [4].

Liskov and Guttag propose the implementation in Listing 7 to fix the symmetry violation [6] (pp. 182). Essentially, this implementation ignores the `z`-dimension when two `Points` are compared but includes it when two `Point3Ds` are compared. This design maintains symmetry but at the cost of transitivity (Figure 3). The counterexample means that for objects `a`|`Point3D(7,7,-6)`, `b`|`Point(7,7)`, and `c`|`Point3D(7,7,2)`, `a.equals(b)` returns `true`, `b.equals(c)` returns `true`, but `a.equals(c)` returns `false`. The design and implementation of equality is not always simple, and at times may even confuse experts.

Listing 7. Implementation of `Point`'s hierarchy in [6] (pp. 182).

```
// For Point
public boolean equals(Object p) {
 if (p instanceof Point) return equals((Point)p);
 return false;
}
// Overloaded definition
public boolean equals(Point p) {
 if (p==null || x != p.x || y !=p.y ) return false;
 return true;
}

// For Point3D
public boolean equals(Object p) {
 if (p instanceof Point3D) return equals((Point3D)p);
 return super.equals(p);
}
// Overloaded for Point
public boolean equals(Point p) {
 if (p instanceof Point3D) return equals((Point3D)p);
 return super.equals(p);
}
// Overloading for Point3D
public boolean equals(Point3D p) {
 if (p==null || z != p.z) return false;
 return super.equals(p);
}
```

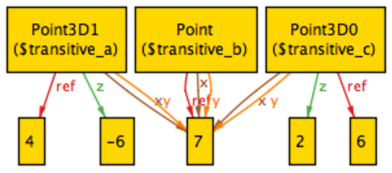


Figure 3. Transitivity violation in `Point`'s type hierarchy.

D. Solutions for Implementation Reuse

Bloch suggests using composition over inheritance for the implementation reuse as a fix to this problem ([1], Item 7).

[4]http://eqchecker.sourceforge.net/

In this case, a `Point3D` would compose a `Point` object and start a new type hierarchy. We, on the other hand, believe that equality should not be the only determining factor for choosing composition over inheritance. Hence, we have proposed two different solutions to this problem in [9]: making a type hierarchy *type-incompatible* or *hybrid*. We demonstrate that EQ can check these solutions as well for the desired properties.

A *type-incompatible* hierarchy will not allow its types to be equal to each other, that is, a `Point` cannot be equal to a `PolarPoint` or a `Point3D`. This can be implemented by replacing `o instanceof Type` with `o.getClass()==this.getClass()` in `Point` and `Point3D`. However, this is a restrictive design. It preserves equivalence by not allowing classes in a type hierarchy to be considered equal.

The *hybrid* hierarchy allows a type hierarchy to contain both *type-compatible* and *type-incompatible* sub-hierarchies. Implementation of the hybrid hierarchy can be achieved by using the *template method* design pattern (Listing 8). The `eq` template method provides a means for the participating classes to perform a bidirectional equality test (line 5) that the regular `equals` lacks. Since the bidirectional test of `eq` will allow two types to be equal only when this is permitted by the both sides, all of the equivalence properties can be preserved with this design. The Alloy model is shown in Listing 9, which passes EQ's checking.

Listing 8. Hybrid implementation of `Point`'s type hierarchy.

```
1  // Implementation for Point
2  public final boolean equals(Object o) {
3   if(!(o instanceof Point)) return false;
4   Point that = (Point)o;
5   return this.eq(that) && that.eq(this);
6  }
7  protected boolean eq(Point that)
8  { return x == that.x && y == that.y;}
9
10 // PolarPoint will inherit Point's equals() and eq()
11 // Point3D overrides eq but inherits equals()
12 protected boolean eq(Point p) {
13  if(!(p instanceof Point3D)) return false;
14  ... // return comparisons of x, y, and z for the two Point3Ds
15 }
```

Listing 9. Predicate for the hybrid hierarchy of Listing 8.

```
pred Object :: equals( that: Object ) {
 (this in Point − PolarPoint − Point3D) =>
  ((that in Point − Point3D) and (that.ref != 0) and
  (this.x = that.x) and (this.y = that.y) and // this.eq(that)
  (that.x = this.x) and (that.y = this.y))      // that.eq(this)
 else (this in PolarPoint) =>
   ... // Same as Point's due to inheritance
 else (this in Point3D) =>
  ((that in Point3D) and (that.ref != 0) and
  (this.x = that.x) and (this.y = that.y) and (this.z = that.z) and
  (that.x = this.x) and (that.y = this.y) and (that.z = this.z))
}
```

IV. Related Work

Several past work have been directed toward the investigation of the design and implementation of equality [1], [6], [7], [9]. Others extend Java to relation types [12] or use annotations to automatically generate the implementation of `equals` [3], [8]. We, on the other hand, check the correctness by directly analyzing the available source code.

Existing static analysis tools such as FindBugs detect only a small subset of the violations that EQ can when type hierarchies are involved. Others such as JLint, PMD, and ESC/Java ignore equality completely.

Program verification tools such as [2], [5] reason about one function at a time using Hoare triples. It would require significant effort to expand them to handle equality where multiple `equals` must be considered at the same time.

V. Conclusion

We have illustrated the manifestation of equality-related problems in the context of an evolutionary development. Much care is required to prevent or detect these problems. Once committed, they can be hard to spot even for experts. Under these circumstances, a tool such as EQ can be a useful aid. EQ is open-sourced at http://eqchecker.sourceforge.net.

References

[1] J. Bloch, *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

[2] X. Deng, J. Lee, and Robby, "Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems," in *ASE*, 2006, pp. 157–166.

[3] N. Grech, J. Rathke, and B. Fischer, "JEqualityGen: Generating Equality and Hashing Methods," in *GPCE*, 2010, pp. 177–186.

[4] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[5] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *TACAS*, 2003, pp. 553–568.

[6] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[7] S. Nelson, D. J. Pearce, and J. Noble, "Understanding the impact of collection contracts on design," in *TOOLS*, 2010, pp. 61–78.

[8] D. Rayside, Z. Benjamin, R. Singh, J. P. Near, A. Milicevic, and D. Jackson, "Equality and Hashing for (Almost) Free: Generating Implementations from Abstraction Functions," in *ICSE*, 2009, pp. 342–352.

[9] C. R. Rupakheti and D. Hou, "An Empirical Study of the Design and Implementation of Object Equality in Java," in *CASCON*, 2008, pp. 111–125.

[10] ——, "An Abstraction-Oriented, Path-Based Approach for Analyzing Object Equality in Java," in *WCRE*, 2010, pp. 205–214.

[11] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework," in *CASCON*, 1999, pp. 125–135.

[12] M. Vaziri, F. Tip, S. Fink, and J. Dolby, "Declarative Object Identity Using Relation Types," in *ECOOP*, 2007, pp. 54–78.