Equality and Hashing for (almost) Free: Generating Implementations from Abstraction Functions

Derek Rayside, Zev Benjamin, Rishabh Singh,
Joseph P. Near, Aleksandar Milicevic and Daniel Jackson
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{drayside, zev, rishabh, jnear, aleks, dnj}@csail.mit.edu

Abstract

In an object-oriented language such as Java, every class requires implementations of two special methods, one for determining equality and one for computing hash codes. Although the specification of these methods is usually straightforward, they can be hard to code (due to subclassing, delegation, cyclic references, and other factors) and often harbor subtle faults. A technique is presented that simplifies this task. Instead of writing code for the methods, the programmer gives, as a brief annotation, an abstraction function that defines an abstract view of an object's representation, and sometimes an additional observer in the form of an iterator method. Equality and hash codes are then computed in library code that uses reflection to read the annotations. Experiments on a variety of programs suggest that, in comparison to writing the methods by hand, our technique requires less text from the programmer and results in methods that are more often correct.

1 Introduction

Determining equality between objects in an object-oriented language is tricky. The subtle pitfalls that face programmers have been described in numerous tutorials (see, *eg*, [5, 16]), and automated fault finders often specifically target equality errors [7, 10, 17].

This paper pursues a complementary goal: generating equality code automatically to make it correct by construction. The technique proposed is simple and can work (by using reflection, *eg*) in a standard Java environment, requiring no special tools for preprocessing or execution.

In our framework, two objects are regarded as equal if they represent the same abstract value. The programmer specifies an abstraction function that maps objects to their abstract counterparts, which is then used at runtime by reflective library code to compute and compare the abstract values.

The programmer can still, of course, make a mistake by giving the wrong abstraction function. Our experiments suggest, however, that the equality faults that plague programs arise not because programmers fail to understand what they should be comparing, but rather because of (a) simple errors implementing the comparison, or (b) subtle errors related to subclassing, delegation, cyclic references, *etc* (§5). Both kinds of errors are eliminated in our technique.

Furthermore, the most insidious outcome of defective equality implementations is that basic equivalence properties – symmetry, reflexivity and transitivity – are violated [5]. Even if the abstraction function fails to match the programmer's intent, our technique still guarantees that equality will be an equivalence, and that – in Java terms – every class will satisfy the 'object contract'. This property follows from the observation that *any* total function induces an equivalence relation on its domain. That is, the equality $\approx$ defined by

$$a \approx b \;\Leftrightarrow\; f(a) = f(b)$$

is an equivalence by definition. Moreover, for any equality $\approx$, a function f exists with this property (in particular, the function f that maps each object to its equivalence class). The abstraction function approach is therefore also complete in the sense that a function exists for any notion of equality we might need.

Abstraction functions have long been advocated as essential (but informal) documentation [13], are included in specification languages (for example, as

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

'model fields' in JML [12]), and are exploited by verification tools (such as ESC/Java [6]). Exploiting abstraction functions for equality thus gives them additional leverage, increasing the benefit they bring. Furthermore, to minimize the cost of writing an abstraction function, we chose to use a scheme that makes use of code already written. Rather than specifying model fields, we assume that every object can be represented as a sequence of objects. The programmer simply provides an iterator that yields this sequence, along with a directive indicating whether the order of elements in the sequence is significant. Because this iterator has usually been written anyway, the marginal cost of providing an abstraction function is often just the directive alone.

In order to evaluate our technique, we took some programs with existing implementations of equality, and replaced them with new implementations using abstraction functions instead. We then compared the code sizes of the two versions, and tested both versions using the Randoop tool [17] for conformance to the object contract and the vendor test suites for regressions. In short, we found that the new versions were both smaller and more robust. Each of the programs we examined had faults in their implementations of equality, none of which are present in our annotated versions.

2 Equality and Abstraction

In an object-oriented language, two objects that are distinct may nevertheless need to be treated as equivalent. In the physical world, there is no such notion; each object is unique and related to others only by sharing some of its properties. But the execution of a program usually produces multiple copies of the same conceptual object and it is necessary to be able to determine when these copies represent the same underlying object.

Equality can be defined in various ways, according to its intended usage. A common approach says that two objects are equal if and only if they are *observationally equivalent*: that is, any sequence of operations on the two objects will produce the same results. In a language with data abstraction, an object is observed through its public methods, so comparing concrete representations is too strong.

The notion of observational equivalence, while conceptually straightforward, does not lead directly to a generic implementation, since each datatype typically has its own set of observer methods. An alternative approach regards each object as representing some abstract value, obtained by applying an abstraction function [9] to the concrete object. Since the abstract value presumably captures exactly the observable properties, we can determine whether two objects are observationally equivalent simply by testing mathematical equality of their abstract values.

To realize this idea, we require each datatype to declare a standard observer method that implements the abstraction function by mapping the object to a sequence (concretely, an Iterator). Using a sequence gives uniformity, but does 'bias' [11] the representation of the abstract value. We therefore allow the user to indicate with annotations that certain features of this sequence – such as the order of elements – are to be ignored.

3 Object Contract Specification

This section outlines the properties we expect of implementations of equals and hashCode, and which are guaranteed by the correct use of our technique. Since these properties are consistent with the Java object contract, our technique allows all legal uses of equality in Java programs that satisfy it. In addition, our technique supports the more stringent approach of Liskov and Guttag [13].

Observational equivalence requires that equality be a mathematical equivalence:

reflexive:	$x.\mathsf{equals}(x)$
symmetric:	$x.\mathsf{equals}(y) \Leftrightarrow y.\mathsf{equals}(x)$
transitive:	$x.\mathsf{equals}(y) \wedge y.\mathsf{equals}(z) \Rightarrow x.\mathsf{equals}(z)$

It further stipulates that equal objects be substitutable:

$x.\mathsf{equals}(y) \Rightarrow f(x).\mathsf{equals}(f(y))$

for all observer methods f; and that objects that can be distinguished by observation are not equal:

$\neg x.\mathsf{equals}(y) \Rightarrow \neg f(x).\mathsf{equals}(f(y))$

for some observer f.

The Java object contract for hashCode stipulates that it be consistent with equals:

$x.\mathsf{equals}(y) \Rightarrow (x.\mathsf{hashCode}() == y.\mathsf{hashCode}())$

As in Java, comparisons to null return false. In addition, equals and hashCode always terminate and are side-effect free. Following Scheme [1, §11.5], two cyclic structures are equal when the (possibly infinite) tree unfoldings of their abstract state are element-wise equal.

Liskov and Guttag [13] note that equality should also be temporally consistent so that the result of a comparison of two objects doesn't change over time. This approach requires that mutable objects be compared by reference. Java's object contract has a weaker (and more complicated) notion of temporal consistency,

which conveniently allows two collections to be regarded as equals if they contain the same elements, but which can create problems when mutable objects act as keys in associative containers. Our technique supports both approaches.

4 Semantics

The heap of an executing program is usually viewed as a graph of object references linked by labeled edges corresponding to fields. To enable a more uniform traversal of the abstract state, we view the abstract heap as a similar graph, but with edges labeled by integer indices (as in [1]) so that the elements of collections can be treated in the same way as the named components of fixed size objects.

The abstraction function, $\mathcal{A}$, therefore maps a concrete object to a sequence of *parts*, representing the objects referenced by these indexed outgoing edges.

The abstract contents of an object are not sufficient to characterize it. Objects of different types, even with the same content, are often considered unequal. For example, consider an employee object and a bank account object: the employee has only a social security number, and the bank account has only an account number. Even if these two numbers are the same, the two objects are not equals. On the other hand, it is sometimes desirable to equate objects of different types. In Java, two sets containing the same elements are considered equal even if one is implemented as a HashSet and the other as a TreeSet. We therefore allow the user to associate each Java class X with an *equality type* $\mathcal{E}(\mathsf{X})$, which will be a supertype of X (or X itself). The employee and bank account classes would have different equality types, but HashSet and TreeSet share the Set equality type. For convenience, we overload $\mathcal{E}$, and write $\mathcal{E}(x)$ for the equality type associated with the class of an object x.

Each equality type T has an associated boolean ordering property, $\mathcal{O}(T)$. If the order of the elements returned by the abstraction function for T is signficant, then $\mathcal{O}(T)$ is true; otherwise false.

Equality. Two objects x and y are equals if the following conditions hold:

1. same equality type:
   $\mathcal{E}(x) = \mathcal{E}(y)$
2. same overall number of parts:
   $\mathcal{A}(x).\mathsf{size}() = \mathcal{A}(y).\mathsf{size}()$
3. each part occurs the same number of times:
   $\forall i : 0..\mathcal{A}(x).\mathsf{size}() \mid$
   $\mathcal{N}(\mathcal{A}(x), \mathcal{A}(x)[i]) = \mathcal{N}(\mathcal{A}(y), \mathcal{A}(x)[i])$,
   where $\mathcal{N}(s, e)$ counts the occurrences of element e in sequence s
4. if ordered, parts occur in the same order:
   $\mathcal{O}(\mathcal{E}(x)) \Rightarrow$
   $\forall i : 0..\mathcal{A}(x).\mathsf{size}() \mid \mathcal{A}(x)[i].\mathsf{equals}(\mathcal{A}(y)[i])$

These conditions alone make equality an equivalence relation (provable by induction over the structure of $\mathcal{A}$).

Hashing. The hash code of an object is a composition of the hash codes of its parts. Let $\mathcal{A}_H(x)$ be the subsequence of parts in $\mathcal{A}(x)$ that are used for hashing – from the empty sequence (which will result in many hash collisions) to $\mathcal{A}(x)$ itself (which may make computing the hash code expensive).

Further, let $\otimes$ be the hash composition function. If $\neg\mathcal{O}(\mathcal{E}(x))$, then $\otimes$ must be associative and commutative; otherwise the function recommended by Bloch [5] may be used. Hashing is then defined recursively as follows:

$$\mathcal{H}(x) = \begin{cases} k & \mathsf{isCyclic}(\mathcal{A}_H(x)) \\ \mathsf{reduce}(\otimes, \mathsf{map}(\mathcal{H}, \mathcal{A}_H(x))) & \text{otherwise} \end{cases}$$

where k is some pre-defined constant.

So long as the implementation of $\mathcal{A}$ is deterministic, side-effect free, terminates, and faithfully represents the abstract state, these conditions for equality and hashing – as implemented in our technique – will guarantee the conditions of the contract discussed in §3.

5 Why Equality and Hashing are Hard

Achieving observational equality without breaking the object contract is known to be tricky (*eg*, [5]). This section discusses five areas of difficulty, and notes how the systematic technique presented in this paper resolves them.

Simple Errors. Programming requires organizing many details, some of which are often overlooked. In the programs we studied, simple errors (such as implementing equals but forgetting to implement hashCode) were common. Additionally, as programs evolve, maintenance of equality and hashing methods is frequently neglected. When a class is modified by a change to its fields, its equals and hashCode methods (as well as the equals and hashCode methods of its subclasses) should likely be updated, too.

If these methods are generated automatically, as they are in our technique, programmers are no longer burdened with the repetitive task of maintaining them, and their correctness is guaranteed.

Subclassing. The equals method of a subclass will usually need to reference the fields of the superclass, but these may not even be in scope. A common approach is to call the superclass equals method, but because of the asymmetry of dynamic dispatch, this may cause violations of the symmetry condition of the object contract.

This dilemma is easily resolved in our technique by assigning distinct equality types to the subclass and superclass.

Delegation. Decorator/wrapper classes are used to add features to an existing class. This pattern is often used to make a thread-safe class from a thread-unsafe one. For example, a common implementation of equals in such a class is:

```
boolean equals(Object that) {
  synchronized (this) { return this.w.equals(that); } }
```

If the wrapped implementation of equals is faulty, then this implementation may exhibit a reflexivity violation. We can guard against such violations by introducing a reference equality check:

```
boolean equals(Object that) {
  if (this == that) return true;
  synchronized (this) { return this.w.equals(that); } }
```

Programmers often forget to include this reference equality check in their wrapping classes.

The code generated by our technique includes this extra check and avoids this violation. Delegation can also lead to symmetry violations, which are similarly handled.

Objects of Different Classes. Sometimes objects should be regarded as equal even when they belong to different classes. The JDK Collections contain many such cases: a HashSet, for example, can be compared to a TreeSet, and the two will be considered equal if they contain the same set of elements, even though they use different implementations internally.

Unlike all other approaches to automating equality, our technique easily handles this case by allowing the programmer to assign the same equality type (in this case Set) to the two classes.

Cyclic Object Graphs Cyclic object graphs – which are not uncommon in object-oriented programs – are even more challenging for equality and hashing than they are for serialization. Programmers often try to sidestep the problem by defining equality only on fields that do not contain cycles (violating our condition that $\mathcal{A}$ be faithful to the abstract state), or they simply allow a comparison to overflow the stack.

Proper handling of equals and hashCode in the presence of cycles is, however, not impossible. Using an approach similar to that of Eiffel [14], our technique generates equals and hashCode methods that are guaranteed to terminate with a correct answer, even in the presence of cycles.

We test equality by assuming that the objects being compared are equals and searching for evidence to refute that assumption, traversing the object graph and adding new assumptions to a global table as we encounter new objects. If, in any step of the algorithm, no new assumptions are added and no assumption can be refuted, then the original objects must be equals.

When computing hashCode, if a cycle is detected within an object structure, a constant is returned as the hashCode for the entire structure. Thus, any cyclic structure will hash to the same constant value. This approach may lead to hash collisions, but it does respect the object contract.

6 Annotations

Our technique generates equals and hashCode implementations for classes annotated with an equality type and an abstraction function. Table 1 presents a summary of these annotations.

In the most common case, the @ConcreteEquality annotation can be used to say that the equality type is the class itself, and the abstraction function is an iterator over the fields of the object (an implementation of this iterator is provided by our implementation). For example, we annotate the simple class Point { int x,y; } with @ConcreteEquality.

If we wish to equate objects of different concrete classes, then we must introduce @EqualityTypeDefn and @AbstractionFunction annotations. For example, in the Java Collections, we can compare an ArrayList and a LinkedList for equality, and we will find that they are equal if they have the same contents in the same order. To indicate this, we add two annotations to the List interface: @EqualityTypeDefn(Ordering.Total) and @AbstractionFunction("iterator"). These annotations indicate that List will serve as the equality type for all classes that implement it, and that the ordering of the contents of a List is significant in determining equals. Our technique then generates equals and hashCode methods for

Table 1 Annotations for expressing equality types and abstraction functions

Annotation + Description	Example(s)
@EqualityTypeDefn Declares that this Java type represents an *equality type* and indicates whether or not its objects' parts are ordered.	@EqualityTypeDefn(Ordering.Total) interface List {...} @EqualityTypeDefn(Ordering.None) interface Set {...}
@AbstractionFunction Names the method that provides an iterator over the parts of this object.	@AbstractionFunction("iterator") interface Collection {...}
@ConcreteEquality Indicates that the abstract parts of object x are the objects its fields refer to.	@ConcreteEquality class Point { int x, y; }
@NotKey Used with @ConcreteEquality to exclude fields from $\mathcal{A}(x)$.	@ConcreteEquality class BankAccount { final int id; @NotKey double currentBalance; ... }
@ReferenceEquality Indicates that the unique identifier for this object should be used for equality, and so no other object can be equal to this one.	@ReferenceEquality interface Iterator {...}

all classes that implement the List interface. Each class must provide its own implementation of the abstraction function (*ie*, the iterator method), but the generated implementations of equals and hashCode are the same.

Similarly, we add the following annotations to the Set interface: @AbstractionFunction("iterator") and @EqualityTypeDefn(Ordering.None), indicating that the contents of a Set are not ordered.

Finally, to implement the Liskov and Guttag [13] approach to equality for mutable datatypes, the @ReferenceEquality annotation can be used.

Writing these annotations puts far less burden on the programmer than writing the equals and hashCode methods they replace. The most common case is covered by @ConcreteEquality; a custom iterator is needed only when the abstract and concrete states differ. Even when needed, however, the definition of this iterator is frequently simpler and less prone to error than the equals method it replaces. Moreover, the automatically generated methods are guaranteed to fulfill the object contract, eliminating many subtle bugs.

6.1 Subtypes and Equality Types

An equality type is assigned not only to the class or interface it explicitly annotates, but also to all subtypes. Any class lacking an ancestor marked as an equality type is treated as having its own unique equality type.

To ensure that this identifies a class's equality type unambiguously in the presence of multiple supertypes, we require that at most one supertype of a class be annotated with @EqualityTypeDefn.

Our prototype implementation includes a static analysis tool that enforces this condition at compile time. It would correctly reject a class that attempted to implement, for example, both Set and List. Though not generally enforced, the stipulation that this situation should be impossible is informally stated in the documentation of Collection.equals().

7 Evaluation

Four questions come to mind for evaluating our technique: (1) Is it sufficiently expressive to replace the equals and hashCode implementations that programmers write in practice? (2) What effect does it have on correctness? (3) Is it easier or more difficult than implementing equals and hashCode by hand? (4) How does our (prototype) implementation compare in performance to hand-coded methods?

To answer these questions, we annotated three widely used programs: the JDK Collections library v1.4 (a subset of java.util), the Apache Commons Collections library v3.2, and JFreeChart 1.0.0 (an open source plotting program that is a member of the DaCapo Benchmark Suite [4]).

7.1 Expressiveness

Our technique was able to replace almost all the equals and hashCode methods of the three benchmark programs, as shown by Table 3. Those that we were not able to replace were within classes that either in-

Table 2 Test results. 'Errors' are test cases that throw unexpected exceptions. 'Failures' are test cases that compute incorrect results.

	Randoop Tests					Vendor Tests				
	Total	*Original*		*Annotated*		*Total*	*Original*		*Annotated*	
		Errors	Failures	Errors	Failures		Errors	Failures	Errors	Failures
JDK	20661	1	184	0	0	1366	2	5	2	7
Apache	3415	0	7	0	0	2443	0	0	0	0
JFreeChart	59	2	57	2	0	1038	1	8	0	3

tentionally violated the object contract (such as IdentityHashMap) or used weak references to control interaction with the garbage collector.

Table 3 Number of manually written equals and hashCode implementations in original program, and remaining after code was converted to use our technique.

Case Study		Before	After
JDK	equals	24	2
	hashCode	24	3
Apache	equals	56	6
	hashCode	57	9
JFreeChart	equals	263	0
	hashCode	114	0

7.2 Correctness

To evaluate correctness, we ran two sets of unit tests on each of the benchmark programs: a set generated by the Randoop tool [17] and the standard set provided by the program's author.

The annotated versions of these programs are more correct than the originals. This is primarily shown in Table 2, where it can be seen that the annotated versions usually pass more test cases than the original versions. Where they do not, the test cases themselves either contain hard-coded constants tied to particular implementations or are faulty, as discussed below.

Table 7 classifies the faults we found according to the parts of the object contract violated (§3) and according to the reasons that writing correct implementations of equals and hashCode are difficult (§5). The application of our technique fixed all faults found.

JDK Collections Faults. We found the same faults in the JDK 1.4 collections that Pacheco et al. [17] found previously: namely that some of the wrapper sets and maps had reflexivity and symmetry faults.

There is an interesting case where our technique forced us to restructure some of the JDK code, from an original design that produced some surprising behavior to a more conventional design. Originally, the IdentityHashMap.Entry class implemented both the Iterator and Map.Entry interfaces. When one called next() on the iterator, it returned itself in the guise of a Map.Entry. A programmer retaining a reference to this entry from a previous iteration of the loop would be surprised to find that its key and value were now the key and value for the current iteration. This design saved allocating a Map.Entry object for each iteration of the loop, at the cost of surprising mutation behavior (which is documented as Bug #6312706 in the Sun/Java bug database).

Our technique forced us to split this class into two classes, because Iterator and Map.Entry are different equality types, and so no single class may implement both of them.

Failing JDK Collections test cases. Our annotated JDK Collections code fails two test cases that the original code passed: BitSet.hashCodeTests and Vector.ToStringTests. Both of these test cases have hard-coded values (such as hash codes and string encodings) that are computed differently (but still correctly) in our implementation.

Apache Collections Faults. Delegation issues led to symmetry faults in SynchronizedBuffer.equals() and ReferenceMap.equals(), and a reflexivity fault in MapBackedSet.equals().

PriorityBuffer.equals() was not implemented, even though it was clear from the usage that an implementation was expected.

JFreeChart Faults. We found 32 classes in JFreeChart that implement equals but not hashCode.

AbstractObjectList.equals() was actually performing a prefix test, rather than an equality test, because it neglected to compare the size of the lists. This caused symmetry and transitivity failures. Furthermore, the equals and hashCode implementations in this class failed to terminate on cyclic inputs.

ShapeUtilities.equals(GeneralPath,GeneralPath) had a copy-paste error that resulted in it comparing the first argument to itself, and so it always returned true.

XYImageAnnotation had a 'todo' indicating that the image field should be serialized. After de-serialization, the image field was null, and so the equality test failed.

Range had a fault caused by the special primitive double value NaN (not-a-number). The programmer mistakenly expected == to be reflexive, when in fact it is not: all boolean operations involving NaN return false, including NaN==NaN. java.awt.geom.Point2D, which is used by JFreeChart, had a similar fault.

We found 14 classes in JFreeChart that neglected parts of their abstract state in their equals and hashCode implementations in order to avoid potential cycles.

Faulty JFreeChart test cases. We found faults in five of JFreeChart's vendor-supplied unit tests. Interestingly, these tests did not exhibit failures when run against the original JFreeChart code because of the fault in AbstractObjectList.equals(): two wrongs made a right. Once we fixed the fault in AbstractObjectList, we found that these test cases failed. Upon inspection, we discovered that the test cases had incorrect expectations about which objects are equals. The numbers in Table 2 reflect the corrected versions of these tests cases. The five test cases in question were:

- XYAreaRendererTests.testSerialization()
- StackedXYAreaRenderer.testSerialization()
- CombinedRangeCategoryPlotTests.testCloning()
- CombinedDomainCategoryPlotTests.testCloning()
- CategoryPlotTests.testCloning().

Failing JFreeChart test cases. As shown in Table 2, our annotated version of JFreeChart still fails three tests – which also failed when run with the original code.

XYImageAnnotationTests.testSerialization() fails because of the previously mentioned serialization/equality fault in XYImageAnnotation.

XYPlotTests.testGetDatasetCount() has expectations that are inconsistent with the behavior of the XYPlot constructor.

SegmentedTimelineTests.testMondayThroughFridayTranslations() imprecisely transforms one representation of time to another.

Observations. On the basis of this, albeit limited, sample of programs we might infer that:

1. When programmers wish to simply compare the concrete state of two objects, the kinds of mistakes they make amount to simple oversights, rather than the more subtle issues discussed in §5.

2. When an abstract view is needed in order to compare objects of different concrete classes, programmers tend to make mistakes due to the more subtle difficulties, most often delegation.

3. When equality makes use of an iterator, mistakes occur in the code that calls the iterator, rather than in the iterator itself.

4. Cyclic object graphs are not handled well. The two most common approaches that we observed were: (1) allowing the stack to overflow, and (2) neglecting parts of the abstract state that may lead to a cycle (but whose omission violates the condition that $\mathcal{A}$ is faithful to the abstract state (§3)).

If these observations hold in general, they suggest that our technique provides appropriate automation and focuses programmer effort in the right places. We did not observe a single case of a programmer implementing an iterator incorrectly, which is the one thing our technique requires the programmer to do. All of the faults we observed were due to simple oversights or to subtle issues concerning the implicit definition of equality types or the use of abstraction functions. In our technique, all of these issues are handled mechanically.

7.3 Ease of Use

To assess ease of use, we measured the number of annotations that we added (Table 4) to each benchmark program, as well as the number of classes and methods added and removed (Table 5).

The total number of annotations that we added is similar to, or less than, the net number of methods removed, suggesting that the use of our technique is certainly not more difficult than implementing equals and hashCode by hand. Moreover, assuming that all methods are equally difficult to write, and that annotations are easier to write than methods, these numbers suggest that it is in fact easier to use our annotations. This conclusion is consistent with our subjective experience.

From an ease of use perspective, there seem to be three cases: where the programmer requires (1) concrete equality or reference equality; (2) explicit equality types

Table 4 Number of annotations added to benchmark programs.

	@ReferenceEquality	@ConcreteEquality	@NotKey	@EqualityTypeDefn	@AbstractionFunction	Total
JDK	23	2	0	4	5	34
Apache	37	18	0	4	3	62
JFreeChart	0	130	27	8	4	169

where abstraction functions (iterators) have already been defined; or (3) explicit equality types where abstraction functions have not already been defined. In the first two cases, our technique is clearly easier to use than implementing equals and hashCode by hand. The first case is also the most common. In the third, and least common, case, there does not seem to be significant difference in the level of effort required.

For example, in JFreeChart many of the methods added were actually unrelated to our technique. Fully half were concerned with JFreeChart's special treatment of the equality of java.awt.GradientPaint objects. Normally this extra code would not be required to use our technique.

The third usability case, where the abstraction function was not already implemented, was indeed rare. Of the four @EqualityTypeDefn annotations we added to the JDK Collections, three of them already had iterators defined (Set, List, Map), and only one required us to implement the abstraction function ourselves (Map.Entry). The Map.Entry case required some work: 9 classes implement the Map.Entry interface, and 35 of the 50 methods we added were concerned with Map.Entry objects.

In JFreeChart, @EqualityTypeDefn is applied only to MonthDateFormat, QuarterDateFormat, and various type-specific subclasses of AbstractObjectList. Only the first two of these required manually implementing an abstraction function. These two cases were, subjectively, easier than the Map.Entry case in the JDK Collections.

7.4 Performance

Our prototype is implemented using Java reflection. A faster implementation would statically generate code. Such an implementation is clearly feasible, but would require more engineering effort, and would be less flexible for research purposes.

Table 5 Changes to benchmark programs. 'Base' indicates the number of classes or methods in the original program. '+' indicates classes or methods added to switch to our technique. '−' indicates classes or methods removed.

	Classes			Methods		
	Base	+	−	Base	+	−
JDK	159	7	2	1202	50	72
Apache	728	11	0	5544	39	121
JFreeChart	1158	4	0	8960	30	377

We have made some modest efforts to tune the performance of our prototype. It turns out that some operations, such as field access and method dispatch, execute reasonably efficiently via reflection. Other operations, such as examining annotations and the type hierarchy are slower. Caching the results of these operations improved the performance of our prototype significantly.

Table 6 compares the benchmark program execution times before and after our annotations were added. For all three benchmarks, we measured the time taken to run both test suites (Randoop tests and vendor/author tests). Unit tests tend to exercise equals and hashCode more than a regular program execution would, so these probably provide an upper bound on the change in performance. For JFreeChart, we also ran its "large" workload from the DaCapo benchmark suite [4]. The DaCapo benchmark suite is specifically designed to assess the performance impacts of techniques that transform programs, and so this workload gives a more realistic indication of what one might expect to see on a regular program run.

Table 6 shows that the unit test execution times vary from a speedup of 13% to a slowdown of 104%, with an average slowdown of 23%. However, the slowdown on the more realistic DaCapo workload was a mere 2%, which would be quite acceptable for many real-world applications.

The results reported in Table 6 were computed on a single-core 3.6GHz Pentium4 with 3GB RAM.

8 Related Work

Automatic generation of concrete equality comparisons is not uncommon: there are tools (*eg*, Eclipse), libraries (*eg*, Apache Commons Lang), annotations (*eg*, [15]), built-in language features (*eg*, [1, 14, 16, 18]), and

Table 6 Performance evaluation

Times in milliseconds	Randoop Tests			Vendor Tests			DaCapo Input			*Averages*
	Before	*After*	*Change*	*Before*	*After*	*Change*	*Before*	*After*	*Change*	
JDK	7374	7699	+4%	4410	3837	-13%	.	.	.	-4%
Apache	2231	2514	+13%	2479	5049	+104%	.	.	.	+58%
JFreeChart	669	986	+47%	20382	21217	+4%	21355	21827	+2%	+18%
Averages			+21%			+32%			+2%	+23%

language extensions (*eg*, [3]) that accomplish this goal.

Most of these approaches are for functional languages or work only for record-like structures within object-oriented languages.

Static code generation techniques can be fragile if the programmer needs to remember to regenerate the code each time a new field is added to a class. Approaches based on annotations, reflection, built-in language features, and language extensions are robust in this respect.

Many of these approaches are also fragile in the face of cyclic object references; notable exceptions include Eiffel [14] and Scheme [1]. We borrow the idea of treating the tree-unfolding of cyclic structures from Scheme and the strategy of using an assumption set to compute equality of cyclic structures from Eiffel.

Grogono and Sakkinen [8] and Vaziri et al. [19] both have a concept of abstract state. However, neither of these approaches allow objects of different concrete classes to be considered equals. The two crucial elements of our technique that facilitate these kinds of comparisons are explicit equality types and rich abstraction functions.

Grogono and Sakkinen [8] also distinguish fields as 'essential' or 'accidental' and use this distinction to specify object ownership properties that are taken into account when cloning objects.

Vaziri et al. [19] distinguish fields as either part of the object's identity or not. The identity of an object must be immutable, whereas the other fields may refer to mutable objects. Object construction is controlled so that there can never be two different objects with the same type and the same identity. In other words, Vaziri et al. [19] adopt the Liskov and Guttag [13] view of the consistency of equality, but provide some linguistic mechanisms to make this discipline easier for practicing programmers to follow.

Object mutability (or immutability) plays an important role in the ideas of Liskov, Guttag, Grogono, Sakkinen, and Vaziri et al. We think programmers would be well-advised to follow their ideas. However, in this paper, we do not offer any guidance on how programmers should use mutable objects.

The object contract and difficulties in implementing it have been discussed extensively elsewhere. Abiteboul and Bussche [2], for example, present three formalizations of deep equality and prove that they are equivalent. Their main concern is deep equality in the presence of cyclic object graphs. Bloch's book [5], which is a standard reference for Java programmers, explains other common pitfalls (such as asymmetry due to subclassing) and how to avoid them. Odersky et al. [16] also discuss common pitfalls, and provide some recipes for implementing equals and hashCode manually.

Finally, we note that Grogono and Sakkinen [8] consider four different notions of equality, including a topological notion of equality that is not discussed elsewhere. They use the term 'structural equality' for this topological notion, and use the term 'deep equality' for what we refer to as 'concrete equality' and some other authors refer to as 'structural equality.'

9 Conclusions

Implementing equals and hashCode by hand is tedious and error-prone. We have proposed a simple technique for defining equality in terms of abstraction functions, and have shown that – at least for three non-trivial programs – it is sufficiently expressive to replace the majority of handwritten equals and hashCode methods and results in fewer bugs.

In the common case of concrete equality, the only annotation required is @ConcreteEquality. Manual implementations of concrete equality tend to have simple, rather than subtle, errors. Mechanization is clearly feasible in this case, as demonstrated by a variety of other languages and techniques. Our work lends further support for this feature being included in every object-oriented language.

Enhancing a language with explicit equality type declarations (and an associated consistency checker) might achieve many of the correctness improvements that we

observed above, albeit without the advantage of automatically generating method implementations.

The unique contribution of our technique is support for the case where equality is to be defined over an abstract state that differs from the concrete state. In these cases, we observed that programmers tend to make more subtle errors in their manual implementations of equals and hashCode. These errors seem to involve either the implicit definition of equality types, or the usage of the abstraction function – but not the definition of the abstraction function itself, which programmers appear to be able to implement correctly. By making equality types explicit and mechanizing the usage of abstraction functions, our technique completely avoids the errors that programmers make in practice, while still providing sufficient expressive power and, in most cases, requiring less effort than manual implementations.

Acknowledgments

Thanks to Carlos Pacheco for his help with Randoop and the JCK; to Greg Dennis, Jonathan Edwards, Eunsuk Kang, Rob Seater, Kuat Yessonov for helpful discussions; and to the anonymous reviewers for suggestions that improved the paper.

This research was funded in part by the National Science Foundation under grant 0541183 (Deep and Scalable Analysis of Software).

References

[1] Revised^6 Report on the Algorithmic Language Scheme, September 2007. URL `http://r6rs.org`.

[2] Serge Abiteboul and Jan Van Bussche. Deep equality revisited. In T.W. Ling, A.O. Mendelzon, and L. Vieille, editors, *Deductive and Object-Oriented Databases*, volume 1013 of *LNCS*, pages 213–228, 1995.

[3] David F. Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency—Practice and Experience*, 15(3–5):185–206, March–April 2003. doi: 10.1002/cpe.653.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In Peri Tarr and William R. Cook, editors, *Proc.21stOOPSLA*, October 2006.

[5] Joshua Bloch. *Effective Java*. Addison-Wesley, 2001.

[6] D.L. Detlefs, K.R.M. Leino, G. Nelson, and J.B. Saxe. *Extended Static Checking*. Compaq, Systems Research Center, 1998.

[7] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a SAT solver. In Ivica Crnkovic and Antonia Bertolino, editors, *Proc.6thESEC/FSE*, pages 195–204, Dubrovnik, Croatia, September 2007.

[8] Peter Grogono and Markku Sakkinen. Copying and comparing: Problems and solutions. In Elisa Bertino, editor, *Proc.14thECOOP*, volume 1850 of *LNCS*, pages 226–250, Cannes, France, June 2000.

[9] C. A. R. Hoare. Proof of correctness of data representation. In Friedrich L. Bauer and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *LNCS*, pages 183–193, 1975. ISBN 3-540-07994-7.

[10] David Hovemeyer and William Pugh. Finding bugs is easy. In Geoff Cohen, editor, *Proc.OOPSLA Onward!*, Vancouver, British Columbia, Canada, October 2004.

[11] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Inc., 1990.

[12] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, April 2003. URL `http://www.jmlspecs.org`.

[13] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986. ISBN 978-0262121125.

[14] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992.

[15] Microsoft Research. The F# 1.9.6 Draft Language Specification, September 2008.

[16] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, November 2008.

[17] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In Wolfgang Emmerich and Gregg Rothermel, editors, *Proc.29thICSE*, Minneapolis, MN, 2007.

[18] Simon Peyton Jones. Haskell 98 Language and Libraries, Revised Report, December 2002. URL `http://www.haskell.org/onlinereport/`.

[19] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In Erik Ernst, editor, *Proc.21stECOOP*, volume 4609 of *LNCS*, pages 54–78, Berlin, Germany, July 2007. ISBN 978-3-540-73588-5.

Table 7 Classification of faults found (and fixed)

	Object Contract Violations (§3)									**Difficulties** (§5)					
	equals							hash							
Fault Location / Description	*reflexivity*	*symmetry*	*transitivity*	*non-null consistency*	*termination*	*well-foundedness*	*side-effect free*	*faithfulness*	*equals consistent termination*	*side-effect free*	*oversight*	*evolution*	*subclassing*	*delegation*	*different classes*	*cycles*
JDK 1.4 Collections																
SynchronizedSet.equals(Object)	1	1												1	1	
UnmodifiableSet.equals(Object)	1	1												1	1	
SynchronizedMap.equals(Object)	1	1												1	1	
UnmodifiableMap.equals(Object)	1	1												1	1	
Apache Commons Collections 3.2																
MapBackedSet.equals(Object)	1	1												1	1	
SynchronizedBuffer.equals(Object)	1	1												1	1	
ReferenceMap.equals(Object)	1	1			1	1				1					1	1
PriorityBuffer.equals(Object)							1				1				1	
JFreeChart 1.0.0																
Classes implementing equals but not hashCode[1]									32		32					
Classes ignoring abstract state/with cycle faults[3]							14									14
AbstractObjectList.equals(Object)		1	1		1	1	1		1		1					
ShapeUtilities.equals(GeneralPath,GeneralPath)							1				1					
XYImageAnnotation				1							1					
Range.equals(Object)	1										1					
Totals	8	8	1	1	2	2	17		32	2	37			6	8	15

1. The following JFreeChart classes implement equals but not hashCode: BlockBorder, ChartRenderingInfo, Colorblock, CombinedDomainXYPlot, ContourPlot, DefaultBoxAndWhiskerCategoryDataset, DefaultDrawingSupplier, EntityCollection, FastScatterPlot, HistogramDataset, ItemLabelPosition, JFreeChart, KeyToGroupMap, LegendGraphic, LegendItemCollection, MeterPlot, MiddlePinNeedle, PiePlot3D, PieSectionLabelGenerator, PinNeedle, PlotRenderingInfo, SimpleHistogramBin, SpiderWebPlot, StackedBarRenderer, StandardXYToolTipGenerator, StandardXYZURLGenerator, SymbolicXYItemLabelGenerator, Task, TaskSeriesCollection, ThermometerPlot, TimeTableXYDataset, XYPlot.

2. The equals implementations in the following JFreeChart ignore some part of their abstract state in order to avoid potential cycles: CategoryPlot, CategoryTableXYDataset, ChartRenderingInfo, CombinedDomainCategoryPlot, CombinedDomainXYPlot, CombinedRangeCategoryPlot, CombinedRangeXYPlot, DefaultTableXYDataset, IntervalXYDelegate, PlotRenderingInfo, TableXYDataset, XYPlotTests, XYSeriesCollection.