Java Comparability

Contents:

Introduction

Variables of the same declared type (such as int or float) may contain unrelated information. In this case we say that the two variables have different abstract data types. For example the integer variable x may represent the time in seconds since 1970 and the integer variable y may represent the outside temperature in Celsius. The variables x and y represent two different abstract types. Variables of the same abstract type are said to be *comparable*. Two variables are presumed to be of the same abstract type if they are used together in an expression (eg, x+y, x>y, x=y, etc). We often use the term 'interact' as a short-hand for this.

Comparability is important to dynamic invariant detection. Dynamic invariant detection looks for invariants between each possible combination of variables. If variables are not comparable, then it makes no sense to look for invariants between them. For example, using the definitions of x and y from above, it makes no sense to look for invariants between x and y (even though some (eg, x > y) are almost certainly true).

Our dynamic invariant detection tool (Daikon) looks for invariants over a subset of the variables, fields, and expressions in the target program. We refer to these as *Daikon variables*. We are interested in determining for each Daikon variable what other Daikon variables it is comparable to. Consider the following class:

```
class A {
  int x;
  int y;
  void m1(float f) {}
}
```

There are two Daikon class variables: A.x and A.y. There are also three Daikon variables associated with A.m1: A.m1.this.x, A.m1.this.y, and A.m1.f. A.x and A.y represent all of the values of x and y across all instances of A. The m1 variables represent all of the values of x, y, and f across all calls to m1().

Currently, we make use of the static tools Ajax (Java) and Lackwit (C) to determine comparability. Unfortunately, both of these tools are slow, buggy, and can handle only a relatively small subset of programs. It is Mike's opinion that the design/implementation of these tools makes fixing their shortcomings impossible (or at least extremely difficult).

We are thus interested in developing a comparability analysis of our own. We are considering both static and dynamic approaches. The attributes of each approach and possible design approaches are discussed below.

Definition of Comparability

The definition of comparability presented above is an intuitive one. Many definitions are possible, but a thorough understanding of the goal is necessary in order to understand how to achieve it.

Here is a possible more formal definition: comparability captures bidirectional dataflow reachability. Dataflow reachability indicates when one value can affect another. It indicates which definitions flow to (or affect) which uses; equivalently, it determines, for a given use, which definitions might have defined the value that is obtained at the use. We wish to relate all definitions and uses that are (transitively) related.

Here are some questions about the definition, with proposed answers in italics.

1. If fields in an object interact *before* that object is assigned to a variable, are the corresponding fields of the variable comparable?

   For example, in `a.x += a.y; c = a;` Is c.x comparable to c.y?

   Yes, previous information should be included in comparability

2. If the fields of an object interact *after* a variable is assigned away from that object, are the corresponding fields in the variable comparable?

   For example, in `c = a; c = null; a.x += a.y;` Is c.x comparable to c.y?

   No. Variable c's fields never interacted. Interactions that occur after an object is referenced by a variable are not relevant.

3. Is comparability between formal parameters determined only by the code within the method? Or, if two actual parameters are comparable are the corresponding formal parameters comparable as well?

   Consider the method `m1(A a, B b)`. Formal parameters a and b do not interact within m1(). If we call m1() with actual parameters a1 and b1 that are comparable, do the formal parameters become comparable?

   I think that the comparability of actual parameters should flow to the formal parameters. This implies that if the parameters to a method are comparable in any call that the formal parameters are comparable in all calls.

4. If the actual parameters interact *after* the call to the routine, are the formal parameters still related?

   For example, consider `m1(a1, b1); m1(a2, b2); a1 < b1;` Are the formal parameters for m1 comparable?

   Again, I think they are. This is not exactly the same situation as previously discussed for interactions that occur after a variable refers to an object. A variable can be assigned away from an object, but the formal parameter is in some sense always related to its actual parameters.

5. If two formal parameters become comparable because their actual parameters in one call are comparable (see previous issue), does this imply that all actual parameters should become comparable?

   For example, consider:

   ```
   int a, b, c, d;
   a == b;
   m(a,b);
   m(c,d);
   ```

   Are c and d comparable?

No. There was no interaction between c and d. A similar situation exists if the formal parameters interact in only some calls (due to a branch). Only in the calls where they actually interact should the corresponding actual parameters become comparable.

6. If two variables ever point to the same object, should that make each of their fields comparable?

   For example, consider `a1 = a2`. Does that make a1.x and a2.x comparable?

   It seems like it should make them comparable. Or you could argue that it really makes a1 and a2 comparable, and has no impact on the fields of a1 and a2.
7. There is a hierarchy of Daikon variables at different program points. Can Daikon variables which are children of the same parent have different comparability?

   For example, in class A with fields x and y and methods m1() and m2(), the Daikon variables A.m1.this.x and A.m2.this.x are both children of A.x. The same is true for field y. Can A.m1.this.x and A.m1.this.y be comparable while A.m2.this.x and A.m2.this.y are not?

   The answer to this (and several other issues) depends on the definition of comparability. If comparability is defined as capturing bidirectional dataflow reachability then it is reasonable for the comparability of instance variables to be different at different methods. There may be no instances of A in m2() where this.x and this.y were comparable. But there may be other instances of A in m1() where this.x and this.y were comparable.

   If, however, two variables are considered to be comparable if they are of the same abstract type, then it wouldn't seem to be possible for two children of the same parent to have different comparability (because each child would have the same abstract type).

8. Are fields in a class always comparable with themselves across different instantiations?

   For example, should a1.x and a2.x always be comparable since they are the same field in the same class?

   These should not be comparable. There are many circumstances where a field can be used to hold completely unrelated values. We shouldn't assume the values of a field are always related.
9. Does assignment (alone) make two variables comparable?

   For example, consider the following:

```
Class T {
  int a = 7;
  int b = 15;
  void m1() {
    a = b;
    b = 0;
  }
  void swap() {
    int tmp = a;
    a = b;
    b = tmp;
  }
}
```

   At the end of m1() or swap() are a and b comparable?

Under either definition of comparability it seems that a and b should be comparable as a result of the assignment. If comparability is based on abstract types, then the types for a and b would be the same based on the assignment. If it is based on bidirectional dataflow reachability, it seems that the data from b flowed into a and thus they should be comparable.

10. How do constants affect comparability?

    For example, does `a1.x = 7; a2.x = 7;` imply that a1.x and a2.x are comparable?

    Certainly for common constants such as 0, 1, -1, not-a-number, etc, assigning like constants should not imply comparability. It is not clear what should happen for more unusual numbers. Certainly if the number represents some common symbolic value it should imply comparability.

Static Analysis

This approach tracks data flow statically through the entire program. Conceptually, it is very similar to the type analysis used by Alan and Adam in Jiggetai, except that sets of variables are flowed rather than sets of types.

The approach can be completely sound, but it must make conservative assumptions. The result is that it may be imprecise (depending, of course, on the program). It's my feeling (based partially on the results from Lackwit) that this imprecision may be a significant problem (i.e., a significant number of variables may be indicated as comparable that are not). For example, consider the following code:

```
Class A {
  int x;
  int y;
}

List alist = new ArrayList();
A a1 = new A(); A a2 = new A(); A a3 = new A();
alist.add (a1); alist.add (a2); alist.add (a3)
int z = 7;
A a4 = alist.get(1);
z += a4.x
```

It is unlikely that a static analysis will be able to determine which A was retrieved from the list. While only a4.x and a2.x are truly comparable to z, a static analysis will probably conclude that a1.x, a2.x, a3.x, a4.x are comparable to z (and each other).

The advantage of a static approach is that it is not necessary to run the program and the results don't depend on the test suite. We also may be able to take advantage of Alan and Adams work to implement this relatively quickly.

Dynamic Analysis

A dynamic analysis executes the program over some set of input data (perhaps a test suite). All operations are tracked and when two values interact they become comparable. This would probably be implemented by rewriting the Java byte codes to include extra operations to track comparability. It could also be implemented via the JVM debugging interface, but our feeling is that this would be too slow (since many statements would be changed, this would require virtually single stepping through the program).

This document presents two different possible approaches to dynamically determining comparability, called "tracking variables" and "tracking objects". Each approach requires certain information to be tracked about each value (allocated object or primitive) in the program. Each approach also keeps a separate data structure that maps Daikon variables to comparability information, using the tracked information about each value.

The presentation makes two simplifying assumptions. First, we assume that there is some mechanism to associate the tracked information with each object (for example, each object could have extra space allocated within it for the tracked information, or a separate table mapping objects to the tracked information could be used). Second, we assume that primitives can be treated in the same way. The "Maintaining tracking information for each value" section discusses implementation strategies.

Tracking variables

As mentioned, our goal is to determine comparability for Daikon variables. Unfortunately, operations within the program do not take place on Daikon variables. When an operation occurs in the program, it is not known which Daikon variables (if any) are involved. This approach keeps track, for each object, of the set of Daikon variables that refer to it. This is referred to as the object's *variable set*. The variable set for each object is maintained as the program is running. Any time an interaction between two objects occurs, all of the variables in the corresponding variable sets are marked as comparable.

For example, in the following code, the object created by 'new A()' is referenced by both a1 and a2. The variable set for that object is thus {a1, a2}

```
A a1 = new A();
A a2 = a1;
```

The variable set for each object is maintained as follows. When an object is created, its variable set is initialized to its classname. In the example above, the variable set for the object created by 'new A()' is initialized to "A".

Variable sets are also updated when an assignment statement (var = obj) is executed. Any Daikon variable names associated with left hand side of the assignment must be removed from the variable set of the object currently referenced by var. The same names also must be added to the variable set of object. The names are also recursively applied to any fields within the object up to Daikon's dereference limit.

Determining the Daikon variable names associated with the left hand side (lhs) depends on the type of the lhs. If the lhs is a program variable (not a field), its Daikon variable name is the fully qualified name of the program variable if that name matches a Daikon variable. If the lhs is a field expression, the object containing the field may itself be referred to by multiple Daikon variable names. The Daikon variable names for the lhs are thus formed by appending the field name to each element of the container objects variable set.

Operational Details

There are two primary data structures:

- varcomp: comparability information for variables

  For each program point, a union-find data structure whose domain is the set of Daikon variables at that program point. For instance, varcomp[m1:enter] is a union-find data structure for the Daikon variables in scope at entry to method m1. Varcomp[ppt] has the following operations:

merge (var1, var2)	-	merges the comparability sets for the two variables
merge_vs (vs1, vs2)	-	merges the comparability sets for all of the variables in the variable sets vs1 and vs2.
add_literal (literal)	-	Adds the specified literal as a Daikon variable and returns its name. Only one name will be allocated for identical literals.

- objvars: variable set for each object

  Each object contains the set of Daikon variables that currently reference it in the program.

  add_var (variable) - Adds the specified variable to the objects variable set. Also for each field in the object field.add_var (variable + "field name")

  rm_var (variable) - Removes the specified variable from the objects variable set. Also for each field in the object field.rm_var (variable + "field name")

The following details what the algorithm needs to do for each construct in the program:

- Object creation (new Class())

  Add an entry to the map from the newly created object (obj) to the classname of the object:

  ```
  obj.objvars.add_var (Class.getName())
  ```

- integer literal

  If the literal is interesting (eg, "7"), it is given a fully qualified name (eg, Literal.7) and added to varcomp table.

  ```
  name = varcomp[Global].add_literal (literal)
  obj.objvars.add_var (name)
  ```

  If the literal is uninteresting, its variable set is empty.

- Assignment (x = y)

  If program variable x is a Daikon variable, update the variable sets for x and y.

  ```
  x.objvars.rm_var (fully qualified name of x)
  y.objvars.add_var (fully qualified name of x)
  ```

  If x is a field reference (z.f = y)

  ```
  foreach varname in z.objvars
      z.f.objvars.rm_var (varname + "f")
      y.objvars.add_var (varname + "f")
  ```

- Operation (x+y, x==y, etc)

  The variable set for the result is set to the union of the variable sets of the two operands. Also, each variable in the variable sets of x and y is made comparable.

  ```
  (x+y).objvars = union (x.objvars, y.objvars)
  varcomp.merge_vs (x.objvars, y.objvars)
  ```

- Scope entry - nothing

- Scope exit - nothing

- Procedure entry

The name of each formal parameter (p) must be added to the variable set of each actual parameter.

```
actual.objvars.add_var (formal parameter name)
```

- Procedure exit

  Nothing is done here. The formal parameter name is *not* removed from objvars so that future interactions on that object will be recorded on formal parameter.
- Procedure return - nothing

Variable Tracking Issues

- If an object can be referred to by the same variable name more than once (such as in a recursive call), will the variable set be maintained correctly.

  I think the solution to this is to allow the same name to exist multiple times in the variable set. This would allow the object to keep track of how many times a particular name referenced it. Another possible solution would be to create variable names that are specific to the stack frame so that multiple refernces to the same variable names (in different stack frames) could be kept.

  Yet another approach would be to simply not add a variable name to the variable set more than once. Formal parameters and locals are the only variables that are created on each recursive call. Formal parameters are never removed from an objects variable set and locals are not Daikon variables.

- If actual parameters become comparable *after* a call to a routine, how do we mark the formal parameters as comparable (see question in the definition section).

  Each object that is passed to a formal parameter adds the formal parameter to its variable set. Unlike a normal variable, the relation to the formal parameter is never removed (as there is nothing analogous to assigning the variable a new value). Thus if the comparability of the actual parameter is ever changed, it will be recorded on the formal parameter.

Tracking objects

In this approach, each value (object or primitive) tracks its equivalence set of other objects that are comparable to it. This can be thought of as a tag, such that all comparable objects have the same tag. It is implemented as a reference into a union-find data structure.

For example, consider:

```
class A {
  int x;
  int y;

  void m1() {
  }
}

A a1 = new A();
double z = 705;
}
```

When ’new A()’ is executed an object of class A is created and assigned a new, unique tag. Fields such as x and y are not assigned tags -- but if they are assigned values, the values will (already) have tags. Likewise, ’double z = 705’ assigns a

value (including its tag) to variable z.

Whenever two values interact, their tags are merged (indicating that their values are in the same equivalence set) using the union-find algorithm.

There is a global table that contains comparability information for each Daikon variable. At each instrumented program point, each Daikon variable is evaluated (as an expression) to determine the tag of its value. Two variables are made comparable if the values they reference have the same tags. In the previous example, when m1() is called, the instrumented code will determine the value (and thus the tag) referred to by this.x and this.y.

Operational Details

There are two primary data structures.

- objsets: comparability information for objects

  A union-find data structure whose domain is the set of all values dynamically created by the program.

- varsets: comparability information for variables

  For each program point, a union-find data structure whose domain is the set of Daikon variables at that program point. For instance, varsets[m1:enter] is a union-find data structure for the Daikon variables at entry to method m1.

The procedure *update(ppt)* copies information from objsets to varsets[ppt]. Each pair of variables (eg, x and y) at program point ppt are compared:

```
update(ppt) {
  if find(x.tag) == find(y.tag)
      varsets[ppt].union ("x", "y")
}
```

The following details what the algorithm needs to do for each construct in the program:

- Object creation (new Class())

  Assign a new, globally unique tag to the just-created object.

  ```
  object.tag = next_tag()
  ```

- integer literal

  If the literal is interesting (eg, "7"), then allocate a tag and assign it to the value on each execution. The value could be a unique value for each instance of the same literal or shared among identical literals.

  If the value is uninteresting (eg, "0") either assign a new, globally unique tag value on each execution, or assign a specific tag value that means "uninteresting". This might reduce the number of tags that are ever created, but requires special-case code in the union-find data structure.

- Assignment (x = y)

  Simply assign the tags (x.tag = y.tag). This means that previous values referenced by x are not related to its new value.

- Operation (x+y, x==y, etc)

Merge the tags for x and y and let the resulting value use x.tag (or y.tag, it doesn't matter).

```
union (x.tag, y.tag)
(x+y).tag = x.tag
```

- Scope entry - nothing
- Scope exit - nothing
- Procedure entry - nothing
- Procedure exit - nothing
- Procedure return - nothing

At the end of the run: For each program point ppt, read off comparability information from varsets[ppt]. Use a bottom-up algorithm to compute comparability for non-leaf program points. Make a final pass across all program points so that if two variables are comparable at a higher point, any versions of them at a lower program point are also related. (For instance, m1.this.x and m1.this.y would be comparable iff m2.this.x and m2.this.y were.)

Object Tracking Issues

- If actual parameters become comparable *after* a call to a routine, how do we mark the formal parameters as comparable (see question in the definition section).

  There are a couple of possible solutions. First, the tags associated with the formal parameters of each call to a method could be retained. At the end of the run, each could be examined to see if they were equivalent. Second, the program could be analyzed at the end to see if any of the actual parameters to a method were ever themselves comparable. If they were, then the formal parameters become comparable.

- An alternative approach would track not objects but references to objects, and would produce different results. Consider the following example:

```
A a0 = new A();     // creates object o1
A a1 = new A();     // creates object o2
A a2 = a0;
A a2 = a1;
A a3 = new A();     // creates object o3
if (a1 == a3) ...
```

  If tags are applied to objects, the tags for o2 and o3 are merged on the last line, but o1 never interacts with the other objects and is thus never merged. The merged objects (o1 and o2) are referenced by a1, a2, and a3 and thus these variables are comparable. Object o1 is referenced by a0 and since o1 is disjoint, a0 is not comparable to any of the other variables.

  Alternatively, if tags are applied to references (eg, a0, a1, a2, and a3), all of the references are merged (in the same equality set) by the end of the code

Maintaining tracking information for each value

We have assumed that we can add extra information (a union-find tag, or a list of variables) to each value computed by a program.

For objects, this is easy: place the information in a field, or create a separate table mapping objects to the tracked information could be used. In both approaches, the tracked information goes away when the object is garbage-collected -- so long as the separate table uses weakly held keys, as Java's WeakHashMap does. (Likewise, union-find data structures permit elements to be added and removed at will, and this will be done automatically by the garbage collector.)

Adding extra information to a primitive value is more involved. One approach is to convert all primitives to wrappers (e.g., int becomes Integer) and treat them like other objects.

A more efficient implementation is to create a shadow variable for each variable of primitive type. For every operation on the real variable, the instrumenter inserts corresponding operations on the shadow variable.

In the context of the Java Virtual Machine, an elegant implementation of the shadow variables is to maintain a shadow stack. For each primitive value on the main stack, the shadow stack contains its tracked information. Whenever a primitive value is pushed onto the main stack (for any reason), its tracked information is pushed onto the shadow stack. Because procedure call and return use the stack, the tracked information accompanies primitives automatically, without any special mechanism. The same is true for intermediate results of computations, like "(a+b)*c".

Issues/Questions

- Which definition of comparability - abstract data type or bidirectional dataflow reachability is more appropriate for our purposes?

Table of Contents