Dynamic Loading and Type Safety with Java 2

Verchinine, A.

Dynamic loading and type safety with JAVA2

Alexia Verchinine
herisson@capet.iut-fbleau.fr
LAbORATORY OF AlGORITHMS, COMPLEXITY AND Logic
Informatique, Université Paris XII – 61, av du Général de Gaulle – 94010 Créteil FRANCE

May 20, 2005

Abstract

The topic of the paper concerns Java type safety. A new notion of class equality is suggested. With the help of this class equality notion, we show that there exists java virtual machine (JVM) that is not only statically but dynamically type safe. Moreover, S.Liang and G.Bracha constraints rules are satisfied for it.

1 Introduction

The problem of safe execution of a program was intensively studied especially for the JAVA programming language. The safety of language type system plays important role in these studies. In particular, statical type safety of JAVA language was proved in [DE97] and [vON99]. In computer science, a programming language is type safe when the language does not permit the programmer to treat a value as a type to which it does not belong. This generally requires that the language have a complete specification of its semantics; this in turn implies that programs written in that language follow the specification regardless of what machine it is run on.

Here “statical” means that everything has to be done at the compilation time. On the other hand, the problem of dynamical (at the execution time) type safety firstly formulated by V.Saraswat [Sar97] existed in all implementation of JAVA runtime environment until JAVA2. More precisely, V.Saraswat showed that the type system becomes inconsistent in a presence of dynamic class loading. Later, S.Liang and G.Bracha proposed a solution that had to repair the gap in the type system safety [LB98]. Their solution was accepted and declared as being used for JAVA2 virtual machine specification [LY99]. However, their solution cannot be applied formally due to an imprecision in the definition of equality. It will be shown in section 2 in more detail. Moreover, it is easy to show that the declared solution was not practically implemented in SUN’s second generation of JVM1 (it is shown in appendix A). Actually, it seems that practical solution to the V.Saraswat’s problem (at least in the last generation of JVM) consisted in restricting the possibility of dynamic class loading in rather severe way that makes class replacement in the run time almost unrealistic. In what follows, we show that some more sophisticated definition of class equality may permit to “relax” the above mentioned restriction. It is only the first step towards a general notion of dynamic class compatibility that in our opinion will allow an upgrade on the fly as it was conceived in the JAVA original design. Several promising consequences of our approach will be discussed in the conclusion.

2 Recall of Liang and Bracha constraint rules

Let’s call the problem of type system inconsistency in dynamic environment Saraswat’s problem. The solution to Saraswat’s problem proposed by S.Liang and G.Bracha is shown in figure 1. To apply it

1after sun jre-1.2
< C, L > notation indicates that class C is defined by classloader L.

1. if < C, L₁ > references a field : T field name;
   declared in a class < D, L₂ >, then we generate the constraints:
   \[ T^{L_{1}} = T^{L_{2}} \]

2. if < C, L₁ > references a method : T₀
   method name(T₁, ..., Tₙ);
   declared in a class < D, L₂ >, then we generate the constraints:
   \[ T_{0}^{L_{1}} = T_{0}^{L_{2}}, ..., T_{n}^{L_{1}} = T_{n}^{L_{2}} \]

3. if < C, L₁ > overrides a method : T₀
   method name(T₁, ..., Tₙ);
   declared in a class < D, L₂ >, then we generate the constraints:
   \[ T_{0}^{L_{1}} = T_{0}^{L_{2}}, ..., T_{n}^{L_{1}} = T_{n}^{L_{2}} \]

4. The constraint set \{T^{L_{1}} = T^{L_{2}}, T^{L_{2}} = T^{L_{3}}\} indicates that T
   must be loaded as the same type in L₁ and L₂, and in L₂ and L₃. Even if,
during the execution of the program, T is never
loaded in L₂, distinct version of T could not be loaded by L₁
and L₃.

Figure 1: Liang and Bracha constraint rules

formally, one needs to precise firstly what a classloader the letter L in the notation < C, L > stands for.
Secondly a precise meaning of C₁ = C₂ should be defined.

Even if it is not said explicitly that L stands for the so called “defining classloader” (which applies the
define() method to the class). It is not difficult to understand it. Otherwise, one can use delegation of
loading to reproduce the bug pointed out by V. Saraswat [Sar97].

As to the notion of class equality, it merits more detailed consideration.

3 Class equivalence

What could be the precise meaning of the sentence “let’s consider a class C” ? How the classes can be
identified ? At compile time everything we know is the class fully qualified name\(^2\) (FQN) and the class
location in the local or remote file system. But at run time, the class is already loaded and the only way
it can be identified is the reference given by its classloader. So in what follows “C” in the expression
“class C” is considered as a kind of pointer or reference. Now what a reasonable meaning can be given to
the phrase “class C₁ is equal to class C₂” ? Certainly if C₁ = C₂ (as pointers) both classes have the same
representation and they are identical. This kind of class equality is too strong – for example, you cannot
change source code of a method. On the one hand, it is clear that this notion of equality is safe. On the
other hand, it does not allow any modification, so no hope to perform “upgrade on the fly” or something
similar. Our aim is to give another notion – “class equivalence” – that would be much less restrictive but
still safe with respect to the type system. To do that some preliminary considerations are needed.

A class is thought of as a tuple : < FQN, Super, Mod, Fields, Methods >, where FQN is it’s fully
qualified name; Super is either itself (as defined in the JAVA language) if it exists or the symbol \( \omega \)
otherwise; Mod \( \in \{ \text{private, protected, public} \} \) is a class modifier; Fields is a set of triples < id,

\(^2\)see figure 2
Every package, class, interface, array type, and primitive type has a fully qualified name. It follows that every type except the null type has a fully qualified name.

- The fully qualified name of a primitive type is the keyword for that primitive type, namely, `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, or `double`.

- The fully qualified name of a named package that is not a subpackage of a named package is its simple name.

- The fully qualified name of a named package that is a subpackage of another named package consists of the fully qualified name of the containing package followed by "." followed by the simple (member) name of the subpackage.

- The fully qualified name of a class or interface that is declared in an unnamed package is the simple name of the class or interface.

- The fully qualified name of a class or interface that is declared in a named package consists of the fully qualified name of the package followed by "." followed by the simple name of the class or interface.

- The fully qualified name of an array type consists of the fully qualified name of the component type of the array type followed by "\[]".

Figure 2: Definition of Fully Qualified Names in [LY99], 2.7.5
mod, type> and Methods is a set of quadruples <id, mod, type, code> where id is the identifier of the field/methode; mod ∈ {private, protected, public}; type is an ordered sequence of return type and argument types of the method, and code is the method’s “program”.

As for the types we do not consider them as classes but we suppose that every non primitive type T is represented by a class that will be denoted ClT and every primitive type T is represented by a type.

3.1 Main idea

We say that two class are equivalent iff they have the same list of identifiers (for fields and methods). And associated types of these identifiers are equivalent (for methods, types are component wise equivalent).

3.2 Definitions

Notation: here Proj1 stands for projection on the first component

Definition 1 For primitive types C1, C2:
C1 ≈ C2 iff FQN(C1) = FQN(C2) (these are strings)
For classes C1, C2 (recall that in fact C1, C2 are pointers):
C1 ≈ C2 iff either C1 = C2 or:
1. FQN(C1) = FQN(C2) (these are strings)
2. Super(C1) ≈ Super(C2)
3. Mod(C1) = Mod(C2) (these are strings)
4. Proj1(Fields(C1)) = Proj1(Fields(C2)) (these are lists)
   and for every i ∈ Proj1(Fields(C1))
   if < i, m1, t1 > ∈ Fields(C1) and < i, m2, t2 > ∈ Fields(C2) then
   m1 = m2 (these are strings)
   and Clt1 = Clt2

5. Proj1(Methods(C1)) = Proj1(Methods(C2))
   and for each i ∈ Proj1(Methods(C1))
   if < i, m1, t1, c1 > ∈ Methods(C1) and < i, m2, t2, c2 > ∈ Methods(C2) then
   m1 = m2 and
   Clt1 = Clt2 component wise.

It may seem at the first glance that the definition above leads to an infinite loop at least on recursively defined classes. Now we shall show how to break the potential infinite loop.

Let C1, C2 be classes. To figure out whether or not C1 ≈ C2 we build a partially ordered set of constraints (call it the decision tree) in the following way. At the beginning there is only one constraint C1 ≈ C2 in the root of the ordered-tree. Every time we go down to fields or methods we add successors to the corresponding node and put there the required class/type equivalences as constraint – nodes are added in order which fields/methods appear in class. Once a constraint D1 ≈ D2 (and a node d) is added, we decide whether the node may be closed as defined just below.

Definition 2 A node d that contains a constraint D1 ≈ D2 is closed iff one of the following is true:

1. D1 and D2 are primitive types and have the same fully qualified name.
2. D1 = D2 (as ‘pointers’ i.e. they have physically the same bytecode)
3. There is a predecessor of d that contains the same constraint D1 ≈ D2 or D2 ≈ D1
4. All successors of d are closed.

The decision tree is closed iff it’s root is closed.

Note that the process of descision tree construction will always terminate because of finite number of types and classes which are known to the system at run time.

Definition 3 C1 ≈ C2 iff the decision tree for that constraint is closed.
4 Dynamic type safety of JAVA platform

4.1 JAVA’s virtual machine properties

We do not need a complete specification of JVM, several “natural” properties of JVM will be sufficient. We suppose that

1. Liang and Bracha constraints rules (which was recall in figure 1) applied with our definition of equality (cf 3.2).
2. Synchronously (i.e. at the compilation time) JAVA is type safe.
3. Pointers do not exist. There is only Reference type. So there is no direct memory access. There are two ways to access the memory : direct assignment or parameters substitution in methods call.
4. Dynamic loading is well defined, and a classloader which has applied the define() method cannot reapply this method for a class with the same fully qualified name.

4.2 Proposition

An implementation of the JAVA language which respect constraints defined in 4.1 is type safe.

4.3 Proof of JAVA2 dynamic type safety

Let’s suppose that it exists a class $C$ that compromised the type safety at the run time. This implies that it exists an instruction which assigns a variable $\beta$ of type $T_\beta$ to a variable $\alpha$ of type $T_\alpha$ provided $T_\alpha$ and $T_\beta$ are not the same. Obviously, syntactically (fully qualified name) $T_\alpha = T_\beta$. There is two possibilities:

1. $\beta$ is declared in the class $C$. According to S.Liang and S.Bracha constraints rules (our (1) in section 4.1), there is no constraint generated for this assignment. So $T_\beta$ was defined by the same classloader as $T_\alpha$. Since $T_\alpha$ and $T_\beta$ are syntactically identical, and the classloader never applies the method define() twice to the same fully qualified name. So $T_\alpha$ and $T_\beta$ have the same bytecode. Therefore, the type system cannot be compromised during execution of this program.

2. $\beta$ was not declared in the class $C$. Suppose $\beta$ was declared in some class $D$. There are two possibilities :

   (a) $D$ and $C$ have the same defining classloader. This case is similar to the case 1.
   (b) The defining classloader for $D$ (call it $L_D$) is not the same as the defining classloader for $C$ (call it $L_C$). The constraint : $T_\alpha^C \approx T_\beta^D$ is generated for the program execution.

Suppose that the assignment $\beta$ to $\alpha$ make the type system unsafe. This implies that : There exists an identifier $i$ (field or method) which is a member of $\alpha$ and a member of $\beta$. For this $i$, $\alpha.i = \beta.i$ is inconsistent, i.e.,

$$\exists i \in \left(\text{Proj}_1(\text{Fields}(T_\alpha)) \cup \text{Proj}_1(\text{Methods}(T_\alpha))\right) \cap \left(\text{Proj}_1(\text{Fields}(T_\beta)) \cup \text{Proj}_1(\text{Methods}(T_\beta))\right) : \alpha.i = \beta.i$$

is inconsistent. Here $\text{Proj}_1$ stands for projection on the first component.

Let $T_\alpha^i$ and $T_\beta^i$ be classes associated with $i$ respectively in $T_\alpha$ and $T_\beta$. Notice first that $T_\alpha^i$ and $T_\beta^i$ cannot be primitive types, if they were this would contradict the definition of equivalence $T_\alpha^C \approx T_\beta^D$. Hence, we find ourselves in the same situation as it was with $T_\alpha$ and $T_\beta$ without modifying any hypotheses. We cannot stop this recurrence. So there exist an infinite types chain that does not contain any primitive type. Therefore, there is a loop in this chain (supposing there is none implies a contradiction with our hypothesis of static type safety) and the inconsistency of type system is located in this loop.

Let $T_{\alpha,0}^\iota = T_{\beta,0}^\iota, \ldots, T_{\alpha,n}^\iota = T_{\beta,n}^\iota)$ be the minimal loop of this chain. Notice that $T_{\alpha,0}^\iota = T_{\beta,0}^{\iota(n+1)}$ and $T_{\beta,0}^\iota = T_{\beta,0}^{\iota(n+1)}$. By construction, there cannot be type inconsistency between each $(T_{\alpha,0}^\iota, T_{\beta,0}^\iota)$ — for each $(T_{\alpha,0}^\iota, T_{\beta,0}^\iota)$, we supposed inconsistency after in the chain. But we supposed that inconsistency is located in this loop... this is absurd.
5 Conclusion – Future work

We saw that theoretically, it is possible to define a JVM which guaranties type safety in dynamical environment and allows to the programmer to use class replacement with more freedom. One of interesting questions is how to define an extended class equivalence which allows more freedom than the equivalence we have defined – for example, allowing to permute declaration of fields or methods. This implies a modification of the JVM and have a particular look to the constant_pool at run time to avoid bad references that may appear after class replacement.

An other interesting direction consist in extending the equality relation to a kind of compatibility relation and even more to a notion of relative compatibility on the set of classes. In our opinion, the programing language type system is much more flexible than it may seem from the point of view of existing restrictive approaches.

Acknowledgments: I am thankful to A. Slissemko who draw my attention to the problem and whose remarks were very useful for my work.

A Practical proof of non-implementation of Liang-Bracha

This test shows that the solution to the Saraswat's problem proposed by S.Liang and G.Bracha which was declared to be taken into account in the JAVA virtual machine specification, is not implemented in the SUN's JVM until now (the test gives the same results on all the java virtual machine for Java2 platform). The idea is to load "physically the same bytecode" R in two different classloaders L_1 and L_2. Then we try to assign a variable of the type < R, L_1 > by the variable of the type < R, L_2 >. The bytecodes are identical so these two classes must be considered as equal. Whatever type system be taken. However, an exception is raise at run time (see A.2).

A.1 Version of jdk

herisson@merlin% java -version
java version "1.5.0_02"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_02-b09)
Java HotSpot(TM) Client VM (build 1.5.0_02-b09, mixed mode, sharing)

A.2 Execution

herisson@merlin% java tst RT
Hello...
Error java.lang.reflect.InvocationTargetException in tst.doIt.

A.3 Source code

DelegatingClassLoader.java
public class DelegatingClassLoader extends LocalClassLoader {
  public DelegatingClassLoader (String dir) {
    super(dir);
  }

  public synchronized Class loadClass(String name, boolean resolve) throws ClassNotFoundException {
    Class c;
    try {
      if (name.equals("RR") || name.startsWith("java.")) {
"
return this.findSystemClass(name);  
} else {  
return this.loadClassFromFile(name, true);  
}  
}  
catch (Exception d) {  
System.out.println("Exception " + d.toString() +  
" while loading " + name +  
" in DelegatingClassLoader");  
throw new ClassNotFoundException();  
}  
}  

**********  
LocalClassLoader.java  
**********  
import java.lang.*;  
import java.util.*;  
import java.lang.reflect.*;  
import java.io.*;  

public abstract class LocalClassLoader extends java.lang.ClassLoader {  
private String directory;  
public LocalClassLoader (String dir) {  
directory = dir;  
}  
protected Class loadClassFrom(String name, boolean resolve)  
throws ClassNotFoundException, FileNotFoundException {  
File target = new File(directory + name.replace('.', '/') +  
".class");  
if (! target.exists()) throw new java.io.FileNotFoundException();  
long bytecto = target.length();  
byte [] buffer = new byte[(int) bytecto];  
try {  
FileInputStream f = new FileInputStream(target);  
int readcount = f.read(buffer);  
f.close();  
Class c = defineClass(name, buffer, 0, (int) bytecto);  
if (resolve) resolveClass(c);  
return c;  
}  
catch (java.lang.Exception e) {  
System.out.println("Abort read: " + e.toString() +  
" in LocalClassLoader");  
throw new ClassNotFoundException();  
}  
}  

**********  
R.java  
**********  
public class R {  
public int r = 0;  

7
public class RR {
    public R get_R() {
        return new R();
    }
}

public class RT {
    public static void main() {
        int x;
        try {
            System.out.println("Hello...");
            RR rr = new RR();
            R r = rr.get_R();
            x = r.r;
            System.out.println(" r is " + x + ".");
            r.r = 300960;
            System.out.println(" r is set to " + r.r + ".");
            System.out.println("bye...");
        }
        catch (Exception e) {
            System.out.println("Exception " + e.toString() + 
                " in RT main");
        }
    }
}

import java.lang.reflect.*;

public class tst {
    DelegatingLoader loader;

    public void doIt(String argv[]) {
        try {
            if (argv.length < 1) {
                System.out.println("Usage: java tst <class>");
                return;
            }
            String target = argv[0];
            this.loader = new DelegatingLoader("./");
            Class c = this.loader.loadClass(target, true);
            Object [] arg = {};
            Class [] argClass = {};
            c.getMethod("main", argClass).invoke(null, arg);


```java
} catch (Exception e) {
    System.out.println("Error" + e.toString() + " in tst.doIt.");
}
public static void main(String argv[]) {
    tst t = new tst();
    t.doIt(argv);
}
```


References


