Reading Report
Java and Scala's Type Systems are Unsound

Tanguy Retail
Anastasios Doumoulakis

November 2, 2016

Contents

1 Introduction **3**
1.1 Java's Type System 3
1.2 Scala's Type System 4

2 Wildcards Mechanics **6**
2.1 Covariance and Contravariance 6
2.2 Wildcard Capture 6

3 Unsoundness in Java **7**
3.1 Explicit null pointers 8
3.2 Implicit null pointers 9

4 Unsoundness in Scala **9**

5 Discussion About Unsoundness **10**

6 Potential Solutions **12**

7 Conclusion **12**

Abstract

In this reading report, we analyze the article "Java and Scala's Type Systems are Unsound : The Existential Crisis of Null Pointers". It was written by Nada Amin and Ross Tate for OOPSLA 2016 (Object-Oriented Programming, Systems, Languages, and Applications) that took place in Amsterdam during the ACM SIGPLAN conference on Systems, Programming, Languages and Applications : Software for Humanity (SPLASH).

In the article the authors show a fatal flaw in the type systems of Java and Scala which results in their unsoundness. We will reproduce, in this report, the examples shown in the paper but also explain in more detail the reason behind the unsoundness of Java and the Wildcard Mechanics. We also discuss about the nature of unsoundness in general and glance over the solution proposed in the paper.

1 Introduction

Java is one if not the most used language in the world, especially in the professional world, and it has been that way for well over a decade. After its release in 1996 there have been regular releases of new versions (every two years or so) that brought incremental improvements to the language.

In 2004, Java 5 (the version 1.5 of Java codename "Tiger") was released and with it came a significant number of features that were added to the language. The most important features where autoboxing/unboxing which are automatic conversions between primitive types (such as int) and primitive wrapper classes (such as Integer), annotations and most importantly generics. But to understand generics we need to understand the basics of Java's type system.

1.1 Java's Type System

The most important characteristic of Java's type system is the inductive definition that defines what is a type. The primitive types in java are *byte, char, short, int, long, float, double, boolean* and *void.* Every class and interface in Java is also a type and if τ is a type then $\tau[\,]$ is also a type. We then define the relation of subtyping, every type is a subtype of the class `Object`, the notation we use for subtyping is $\tau <: \tau'$, it means that τ is a subtype of τ'

A crucial part of the unsoundness of Java's type system lies in the definition of the type of `null` that we call it `nil`. In Java `nil` is a subtype of every class and every interface, we also have `nil` $<: \tau[\,]$ if τ is a type.

The main new mechanism introduced the so called *generics*, this feature of Java 5 allows us to instantiate a class or an interface with type parameters. Defining a generic class with `class <T> Foo{...}` allows to write `new Foo<Integer>()` which will replace all occurrences of T with `Integer` in `Foo` when we instantiate it. Type parameterization is possible with classes or interfaces and additional conditions can be imposed to the class type-parameters, namely we can request that a type must inherit or be a super class of a particular class with`<T extends Number>` and `<T super Integer>`. This mechanism associated with wildcards which are explained in the next section are the cause of the unsoundness in Java.

We can also define generic methods the same way we define generic classes, this is done by writing `public <T> void foo(List<T> arg){...}`. Calling this method with `foo(new List<Integer>())` will replace all occurrences of `T` by `Integer` in `foo`.

Defining a type with parameters the way Java does, enables a type of polymorphism known as parametric polymorphism, but there are also other types of polymorphism in Java. Polymorphism is the ability (in programming) to present the same interface for differing underlying forms (data types), a polymorphic type is one whose operations can also be applied to values of some other type, or types.

We have seen the parametric kind but in Java we can also use the inheritance mechanism which enables subtype polymorphism. It is important to know the kinds of polymorphisms we use because their effects can be subtle and we will reference them during this report.

1.2 Scala's Type System

The article we analyze also demonstrate the unsoundness of Scala, this language is much more recent than Java but has gained a lot of popularity in recent years. The first version of Scala was released in 2004, the same year that Java 5 was released, and has since been steadily growing. Today it has become one of the most popular functional language and is highly appreciated in the professional world. Despite the fact that Scala code is meant to be compiled for the Java virtual machine (JVM), the unsoundness in Scala is unrelated to the unsoundness in Java Which is a good thing because it could have meant that the JVM was unsound which is thankfully not the case.

Scala is a functional language, that means its type system is an essential part of the language and finding that it is not sound is a serious problem. To understand where the unsoundness comes from we need, as we did for Java, to understand the basis behind the type system. Scala offers, like Java, parametric and subtype polymorphism but to understand the unsoundness we need only to understand one main feature of Scala's type system : path-dependent types.

Path-dependent types if a mechanism that allows a member in a `trait` to have an individualized type, local only to this instance of the `trait`. In Scala the keyword `trait` allows us to define a structure similar to an interface in Java. To illustrate we define a trait, `Singer`, that defines the ability to sing with a `Voice`, we also have the method `Sing` that, when given a voice and lyrics, sings these lyrics in this voice.

```
trait Singer {
   type Voice;
    def sing(v : Voice, lyrics : String) : Unit
}
```

We do not specify the definition of the type Voice, since `trait` is similar to a java interface, we simply state that it must exist when instantiate a `Singer`. But when given an immutable variable of type `Singer` named `s1`, the type `s1.Voice` will be associated to this instance only. This means that if we are given another immutable `Singer` named `s2`, we cannot use a value of type `s2.Voice` as value of type `s1.Voice`, this guarantees that every `Voice` given to the method `s1.sing` comes exclusively from `s1` and not any other `Singer`.

Similarly to type parameters, we can add constraints to type members, we can impose that a type must be a sub-type or a super-type of another type with the symbols `<:` and `>:`. Lastly we need to explain the use of the keyword `with` which allows *mixin* in Scala.

A mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes. In Scala we can use the previous example to define a `Bird` that sings, we see that `Bird` has mixed in all methods of the trait into its own definition as if class Bird had defined method sing() on its own. Scala allows traits to be combined using the keyword `with` but also to create an anonymous type when creating a new instance of a class. In the case of a `Person` class instance, not all instances can sing, but all `Actors` can. We can see the `class A extends B with C {...}` can be decomposed as `class A extends «B with C {...} »`

```
class Birds extends Singer
class Person
class Actor extends Person with Singer

val singingPerson = new Person with Singer
```

We now know enough about the type system of Scala to understand the origin of the unsoundness, in this next part we describe the last piece that we need to fully understand the unsoundness in Java. That last piece is the wildcard feature in parametrized types in Java.

2 Wildcards Mechanics

Earlier we presented the implementation of parametric polymorphism in Java through the help of generics, in this section we will look at a particular aspect of generics in Java, the use of wildcards.

2.1 Covariance and Contravariance

Wildcards are a way to enable interaction between the two types of polymorphism in Java, sub-type polymorphism and parametric polymorphism, those types of polymorphism were explained in the introduction. In Java wildcards encode a form of *Use-site variance*, this means that you can express co- and contravariance. It is called use-site variance, because the annotation is not placed where the type is declared, but where the type is used as opposed to *Declaration-site variance* where the type parameter is annotated where the generic type is declared as in C#.

Variance refers to how subtyping between more complex types (list of Cats versus list of Animals, function returning Cat versus function returning Animal, ...) relates to subtyping between their components. In Java, if a method wants a `List` of `Animal`, it can perform write or read operations on it and those operations can take an `Animal` from the `List` or give one to it.

If a method performs only read operation on a `List<Animal>` we can also give it a `List<Cat>` or `List<Dog>` without any adverse effects. This is known as a *covariant* use of the `List` of `Animal`. Covariant use of a `List` is written `List<? extends Animal>` in Java, this list can contain any sub-type of `Animal` thanks to the *explicit constrain* `extends`.

Suppose now we have a method that will write `Animals` in a `List`, this `List` can be of any super-type of `Animals` and the method will still work. This method is a *contravariant* use of the `List`. We can explicitly express contravariance in Java with the keyword `super`, writing `List<? super Animals>` means the list will accept any super-type of `Animals`.

2.2 Wildcard Capture

Now that we have shown how wildcards can be used in Java, we explain how *wildcard capture* works when wildcards interact with generic methods. Let's take the `foo` method defined earlier with `public <T> void foo(List<T> arg){...}`, we seen what happens when we call it with `foo(new List<Interger>())` but what if we call it with `foo(new List<? extends Integer>())`? The problem lies in the fact that `? extends Integer` is not a type in itself but a set of possible type constrained by the fact that they need to be a sub-type of `Integer`.

The solution comes as *wildcard capture*, during this process the wildcards are replaced by a "fresh" type variable, the type checking is then done using this new type variable. The *explicit constraints* put on the wildcard are reflected on this new type, but we can also have *implicit constraints*.

```java
class Test{
   static class Cat <N extends CatRace> { ... }

   public static <T> void pet(T toPet){ ... }

   public static void main(String[] args){
      Cat<?> cat = new Cat<>();
      pet(cat);
   }
}
```

In this example when type checking the call to `pet`, the type system replace the wildcard in `Cat<?>` by a fresh type variable, `cat` will have the type `Cat<V>`. There are no explicit constraints on `V`, but implicitly we now that it must be a sub-type of `CatRace` because the wildcard must be a valid type argument to the class `Cat`.
In short when type checking a generic method with an argument containing a wildcard, the type system replaces the wildcard with a new type argument and places the explicit and implicit constraints imposed on the wildcard.

We now know enough about Java's and Scala's type systems to be able to understand where the unsoundness comes from.

3 Unsoundness in Java

In this section we will demonstrate the unsoundness of Java with some concrete examples. What we call unsoundness is the fact that these examples respect the Java Language Specification and compile with the Java compiler but still throw a ClassCastException when we run them, we discuss in more detail the implication of having an unsound type system later in the report.

In all these examples the unsoundness is caused by a having constraints on wildcards impossible to satisfy and using `null` to pass the type checking of a method that implicitly cast an argument into a completely unrelated type. We separate the examples into two categories, the first where the null pointer is explicit and the other where it is implicit.

3.1 Explicit null pointers

Here is the first example :

```
class Example1{
   static class Constrain<D, C1 extends D> {}

    static class Bind <D> {
      <C extends D> D upcast(Constrain<D,C> constrain, C c){
         return b;
        }
    }

   static <D, C> D coerce(C c){
      Constrain<D, ? super C> constrain = null;
        Bind<D> bind = new Bind<>();
        return bind.upcast(constrain, c);
    }

    public static void main(String [] args){
    Dog result = Example1.<Dog, Cat>coerce(new Cat());
    }
 }
```

The most important part of this example to understand is the way implicit and explicit constrains impose an impossible to fulfill condition to a wildcard. We declare a class `Constrain<D, C1 extends D>`, in which the second type parameter must be a sub-type of the first. But in the method `coerce` we declare a variable `constrain` with the type `Constrain<D, ? super C>`, at this point we impose on the wildcard to also be a super-type of `C`. This means that when we call `upcast` and the type checker puts a fresh type parameter instead of the wildcard (`Constrain<D, X>` for example), the new type parameter `X` will have to be a subtype of `D` and a super type of `C` something resembling `Constrain<D, X extends D super C>`.

In the example all the type parameters that are `D` will become `Dog` and `C` will become `Cat`. When we call the method `coerce` the first thing we do is constrain a wildcard like we just described, we then instantiate `Bind`, this class will allow us to do the implicit cast. The cast will be done with the help of the method `upcast`, when calling this method we force `C` to be equal to whatever type the wildcard will give, this works only because `constrain` is `null` and will satisfy any type constrain we impose. This means that `upcast` takes `C c` that was given as an argument and implicitly casts it to a `D` with the return, which is possible because we imposed `C extends D` in the type parameters and because `C` is equal to the type parameter that the wildcard will give meaning `X extends D super C`.

The objective of this example is to trick the type system into believing that a `Cat` can become a `Dog`. We give a wildcard some impossible constrain and then tell the type checker the argument of a method is of that type, which allows us, with an additional constrain on the method, to cast any type into any other type.

3.2 Implicit null pointers

With the first example we tricked the type system into accepting a impossible condition on type parameter by explicitly giving it `null` which will satisfy any type constrain. But using the fact that Java initializes an instance field with null if no value is given, we can write another example that demonstrates the unsoundness of Java's type system without any explicit `null`.

```
class Example2<C, D> {
  class Constrain<B extends D> {}
  final Constrain<? super C> constrain;
  final D field;
  Example2(C c) {
    constrain = getConstrain();
    field = coerce(c);
  }
  <B extends D> D upcast(Constrain<B> constrain, B b) {
    return b;
  }
  D coerce(C c) {
    return upcast(constrain, c);
  }
  Constrain<? super C> getConstrain() {
    return constrain;
  }
  public static void main(String[] args) {
    Dog result = new Example2<Cat, Dog>(new Cat()).field;
  }
}
```

This example works in the same way as the one before but the null is implicitly given to `constrain` when the method `getConstrain()` returns it. Globally this example works in the same way as the first, `constrain` constrains the wildcard to be a sub-type of `Dog` and a super-type of `Cat` and `upcast` does the implicit cast of a `Cat` to a `Dog` and puts the result in a variable. We can point out that the exception won't be triggered if we don't assign `field` to a variable.

The article show multiple other example with small variations, for instance an example where there is are single parameter methods and single type parameter classes. In all examples the same technique is used to point out the unsoundness.

In the next section we show that Scala is also unsound with the approach we used for Java.

4 Unsoundness in Scala

The unsoundness in Scala can be shown with a method similar to the one used in the last section but the features used is the one that was explained in the introduction, *path-dependent types* and *mixin*. In this example the same `ClassCastException` is thrown inside the main method.

```
object unsound {
  trait LowerBound[T] {
    type M >: T;
  }
  trait UpperBound[U] {
    type M <: U;
  }
  def coerce[T,U](t : T) : U = {
    def upcast(ub : LowerBound[T], t : T) : ub.M = t
    val bounded : LowerBound[T] with UpperBound[U] = null
    return upcast(bounded, t)
  }
  def main(args : Array[String]) : Unit = {
    val zero : String = coerce[Integer,String](0)
  }
}
```

In this example is it much clearer where we institute impossible condition, when we instantiate `bounded` we explicitly say it should be a super-type of `T` and a sub-type of `U` with the line `val bounded : LowerBound[T] with UpperBound[U]`. Then in a manner similar to our java examples, we use a method named `upcast` that does the cast of any type into any other type. This method takes a super-type of `T` and a `T` and casts the second parameter into the type of the first, which is a legal thing to do because the first parameter is a super-type of `T`. But in our case we provide this function with a super-type of `T` and a sub-type of `U`, meaning that we can use the result of the function as a `U`.

The exception happens when we try to use the result of `coerce` and actually assign it to a variable forcing the cast. Just like in our Java example, we cannot provide a value that is a super-type of `Integer` and a sub-type of `String`, the only value that can satisfy those conditions is `null`.

We clearly see that, once again, the unsoundness is caused by resolving impossible constraints on type with the null value and using that to provoke a implicit cast. Both Java and Scala can have programs type check and compile but give a `ClassCastException` when we run them. In the next section we will discuss why it is important to have a sound type checker in a language.

5 Discussion About Unsoundness

Java and Scala have been unsound for 14 years, ever since Java 5 and the first release of Scale, in this section we look at the causes of the unsoundness in more detail and discuss about what could have been done to avoid this situation.

The first point the paper makes is about the detection of so-called *nonsense types*, those are the types we obtain when we put impossible condition on wildcard or in the case of Scala, when we use mixin in an unreasonable way. With wildcard capture we can easily encounter a type `V` that must satisfy `Integer <: V <: String` because of explicit and implicit constraints. Such a type would not be refused in the type checking even if the only value that can satisfy this type is `null`.

Nonsense types can also arise from other case such as trying to find the least common super-type of two types during the execution of the type checking algorithm. The paper argues that we can either reject nonsense types, but this approach will limit some features of the language like type arguments. Instead an alternative is to allow nonsense type but only when their existence does not affect the soundness of the type system. In the end the unsoundness arises from an unforeseen interaction between nonsense types and null values.

Wildcard capture and path-dependent types would not be causing unsoundness in Java and Scala if not for the existence of implicit nulls. Null values are useful in a language in many use cases (uninitialized values, optional argument, exceptional behavior, etc..) and removing them from the language would solve some problems but other solutions would have to be found in order to fill the lost functionality.

The Authors of the paper go at length to explain how the unsoundness we caused because of the unforeseen interaction between two features in a language, wildcards and null values for Java and path dependent types and null values for Scala. The problem resides in the fact that in large languages like Java there are so many features that it is almost impossible to check all the possible interactions with existing features when a new functionality is introduced. There have been formalization of Java with Featherweight Java but in the process the language was stripped of many feature to make it smaller including null values which doesn't help the problem at hand. Formalizing the entire language would be a huge task and would not guarantee soundness anyway.

Some methodologies are proposed in the paper to help with preventing unsound language features in the future.

- *Threat to Validity* : Communicate the properties that are the most susceptible to cause problems when generalized when dealing with minimization
- *Open Ended Calculi* : Formalizing part of a language and specifying the assumption made about the features that were not formalized
- *Minimizing Cross-cutting Features* : Remove features that has the least cross-cutting impact if it does not affect the proof of the language to focus on the more complex parts of the minimization.

Other aspect like mechanical verification and problems with the community values and publication process are also mentioned but these are beyond the scope of this report. Instead the last section of this report will present the fixes that are proposed to the languages to fix the unsoundness demonstrated earlier.

6 Potential Solutions

For Java the problem come in the specific case where during wildcard capture, type-argument inference captures the a type but no corresponding value is needed to complete the process. The proposed fix is to refuse a method invocation or check the argument for null values in cases where a null will make the type checker accept typically unacceptable constraints. The problem with this solution is that it his an algorithmic challenge to know in which cases a null value is the only solution to the type checking of a method. The fact that type constraints can be recursive, complicates the process to figure out when to forbid null values as arguments because of the subtyping that can provoke the capture of a wildcard.

This could be facilitated with disallowing subtyping with wildcard capture, which can be done if we forbid the use of `? super` in wildcards that have implicit constraints or banning implicit constraints altogether. But the solution must be mostly backward compatible and the authors state that introducing implicit constraints between so-called "materials" would satisfy backward compatibility.

In Scala the fix would be to forbid path dependent types where the path lead to a null pointer. This can only be done with a path validation during runtime which can be done with minimal overhead in the JVM.

7 Conclusion

All and all the unsoundness in Java's and Scala's type systems show a quirk in the conception of the type systems and not the JVM itself. The unforeseen interaction between feature caused the risk of having a program that type checks and compiles but throws a cast exception during runtime. This is unlikely because the features that are causing the unsoundness are rarely used in a way that can lead to this kind of problem but it is nonetheless possible.

The fact that the languages have been unsound for 14 years shows that there exist issues in the way that feature are tested and released, but it is understandable when we realize the size of languages such a Java. There is no perfect solution to prevent this kind of problem from arising again but research on language design and formalization could help with finding and preventing unsound features or combination of features to be integrated into programming languages.