Java and Scala's Type Systems are Unsound *

The Existential Crisis of Null Pointers

Nada Amin
EPFL, Switzerland
nada.amin@epfl.ch

Ross Tate
Cornell University, USA
ross@cs.cornell.edu

Abstract

We present short programs that demonstrate the unsoundness of Java and Scala's current type systems. In particular, these programs provide parametrically polymorphic functions that can turn any type into any type without (down)casting. Fortunately, parametric polymorphism was not integrated into the Java Virtual Machine (JVM), so these examples do not demonstrate any unsoundness of the JVM. Nonetheless, we discuss broader implications of these findings on the field of programming languages.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

General Terms Design, Languages, Reliability, Security

Keywords Unsoundness, Java, Scala, Null, Existential

1. Introduction

In 2004, Java 5 introduced generics, i.e. parametric polymorphism, to the Java programming language. In that same year, Scala was publicly released, introducing path-dependent types as a primary language feature. Upon their release 12 years ago, both languages were unsound; the examples we will present were valid even in 2004. But despite the fact that Java has been formalized repeatedly [3, 4, 6, 9, 10, 18, 26, 38], this unsoundness has not been discovered until now. It was found in Scala in 2008 [40], but the bug was deferred and its broader significance was not realized until now.

This experience illustrates some potential lessons for our community to learn. For example, when researching a feature, we often develop a minimal calculus employing that feature and then verify key properties of that calculus. But these results provide no guarantees about how the feature in question will interact with the many other common features one might expect for a full language. The unsoundness we identify results from such an interaction of features. Thus, in addition to valuing the development and verification of minimal calculi, our community should explore more ways to improve our chances of identifying abnormal interactions of features within reasonable time but without unreasonable resources and distractions. Ideally our community could provide industry language designers with deep insights into how a given feature fits within the landscape of language design as a whole, all while maintaining our community's independence and productivity.

On that note, another lesson from this experience is that one well understood feature should be refined. In particular, the design of constrained generics is itself overconstrained. There is a notion of a well-formed generic type, in which all type arguments satisfy the constraints on the corresponding type parameters. Most languages simply reject types that are not well-formed, but our examples show that relying on well-formedness is not always sound. Furthermore, well-formedness is not always necessary for soundness, as we informally show is the case for constrained generics. Thus, in addition to exploring type systems in which everything is well-formed, our community should explore type systems in which ill-formed types can be accepted. Section 6 has more discussion on these lessons and others, and afterwards we informally present potential fixes to Java and Scala that are currently being discussed with the respective teams.

2. Java's Generics

Java generics are a form of *parametric polymorphism* [12, 33]. Classes and interfaces can be specified to have type parameters and instantiated to have type arguments. For example, `List<E>` is an interface for data structures implementing indexable lists of elements of type `E`.

Before the introduction of parametric polymorphism, Java already had *subtype polymorphism* as a key feature of the language. An `ArrayList` instance could be provided to

* This work is supported in part by the NSF under grant CCF-1350182, and by the ERC under grant 587327 DOPPLER.

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984004

code that expected a List, Collection, or Serializable. Interaction of subtype polymorphism and parametric polymorphism is non-trivial. As an attempt to enable powerful interaction of these features, Java 5 introduced wildcards [12].

Wildcards encode a form of *use-site variance* [39]. Suppose a method wants a List of Numbers. That method might get Numbers from the List and add Numbers to the List. But in many cases a method does only one of these two things. A method just *getting* Numbers from the List could just as easily be provided a List of Integers or a List of Floats and the method would still work perfectly well. This indicates that the method is a *covariant use* of the List of Numbers [36]; it works given any List of any *subtype* of Number. In Java, this is expressed using the type List<? extends Number>, where ? extends Number is an *explicitly* constrained wildcard that represents some unknown subtype of Number. On the other hand, a method just *giving* Numbers to the List could just as easily be provided a List of Objects and the method would still work perfectly well. This indicates that the method is a *contravariant use* of the List of Numbers [19]; it works given any List of any *supertype* of Number. In Java, this is expressed using the type List<? super Number>, where ? super Number is an explicitly constrained wildcard that represents some unknown supertype of Number.

Java also has generic methods, i.e. parametrically polymorphic methods, and so Java specifies how wildcards should interact with such methods. As an example, consider the following method signature:

```
<E> List<E> reverse(List<E> list) {...}
```

The <E> at the beginning of the signature indicates that the method works for any type E. Next, suppose a variable ns has type List<? extends Number>, and consider what the type of reverse(ns) should be. The challenge is that ? extends Number is not itself a type, and treating it as such would be unsound.

To type-check this, Java uses *wildcard capture*. A wildcard represents *some* type, but each wildcard can represent a different type. So, when an expression supplied to a generic method has a type with wildcard type arguments, Java *captures* the wildcard by introducing a fresh type variable that represents the unknown type described by the wildcard. Java then replaces each wildcard type argument in the type with its corresponding fresh type variable, and then type-checks the generic method using that modified type.

As an example, the following steps are taken to type-check reverse(ns) from above:

1. Determine that the type of ns is List<? extends Number>.
2. This type has a wildcard type argument, so introduce a fresh type variable, say X, for that wildcard.
3. Change the type argument to X, resulting in List<X>.
4. Type-check the call to reverse using List<X> as the type of its argument.

This process results in the type List<X>. But there is something unsatisfactory with this result. With ns, we knew that every value provided by the list is a Number. We reversed that list, so the new list should have the same property. But the type List<X> describes no such information, since X is just some arbitrary type variable that could represent anything. The fact is, there are actually two more steps in wildcard capture that are designed to address such concerns.

The first additional step incorporates the explicit constraints on the wildcard. The wildcard ? extends Number has an explicit constraint on it stating that the unknown type must be a subtype of Number. In the type-checking process, the explicit constraint on the wildcard is reflected by adding that constraint to the fresh type variable X. As a consequence, any value provided by a List<X> is known to be a Number because X is constrained to be a subtype of Number.

The second additional step incorporates the *implicit* constraints on the wildcard [12, 35]. To understand what implicit constraints are, consider the following class, noting in particular the constraint on the type parameter N:

```
class Numbers<N extends Number> extends List<N> {
  public N totalSumOfValues() {...}
}
```

Next, suppose nums has type Numbers<?>, and then consider what the type of reverse(nums) should be. If we apply the above extended process, we determine that its type is List<Y>, where Y is a fresh type variable. But Y is unconstrained because the wildcard in Numbers<?> has no explicit constraints on it. As a consequence, we no longer know that the values provided by the list are Numbers.

But even though the wildcard in Numbers<?> has no *explicit* constraints, we know that *implicitly* it is a subtype of Number. This is because the unknown type that the wildcard represents must at least be a valid type argument to Numbers, which implies it must be a subtype of Number due to the constraint on the corresponding type parameter N. Using this reasoning, Java also adds *implicit* constraints to the fresh type variable Y to reflect that it must be a valid type argument. Java looks at the type parameter corresponding to the wildcard and duplicates all constraints on that type parameter as constraints on the corresponding fresh type variable. In this way, Java is able to deduce that all the values provided by the list resulting from reverse(nums) are in fact Numbers.

These are the basic mechanisms for type-checking with wildcards. There are more mechanisms, such as subtyping, but these are sufficient for our example.

3. Unsoundness of Java

Now that we know the relevant features of Java, we use them to create a method coerce in Figure 1 that can turn any type into any type, without (down)casting. This code type-checks according to the Java Language Specification [12] and is compiled by javac, version 1.8.0_25. Stepping through the code, one can see that coerce simply returns the value of its

```
class Unsound {
  static class Constrain<A, B extends A> {}
  static class Bind<A> {
    <B extends A>
    A upcast(Constrain<A,B> constrain, B b) {
      return b;
    }
  }
  static <T,U> U coerce(T t) {
    Constrain<U,? super T> constrain = null;
    Bind<U> bind = new Bind<U>();
    return bind.upcast(constrain, t);
  }
  public static void main(String[] args) {
    String zero = Unsound.<Integer,String>coerce(0);
  }
}
```

Figure 1. Unsound valid Java program compiled by `javac`, version `1.8.0_25`

argument `t`. When executed, a `ClassCastException` is thrown inside the `main` method with the message "`java.lang.Integer` cannot be cast to `java.lang.String`". The reason is that the JVM does not directly support generics, implementing them through type erasure and type casts. This is fortunate since it causes the JVM to catch the error before it can directly cause severe problems, such as accessing memory associated with a field that is not actually present (though it might be possible that it can indirectly cause problems).

In order to understand the example, the critical type to look at is `Constrain<U,? super T>`. The wildcard in this type is explicitly constrained to be a supertype of `T`. Furthermore, the wildcard is implicitly constrained to be a subtype of `U` because the class `Constrain` states that its second type argument must be a subtype of its first. This means that the wildcard is constrained to be a supertype of `T` *and* a subtype of `U`.

Next, consider the method `upcast` of variable `bind`. The variable `bind` has type `Bind<U>`. This means its `upcast` method will return a `U` if provided a `Constrain<U,B>` and a `B` for some subtype `B` of `U`. It does so by using its stated constraint on `B` to upcast its second parameter to the return type. However, the type argument for `B` is not specified, so it must be inferred. Type-argument inference is done by first collecting the known assumptions and the necessary requirements, and then solving constraints in an attempt to determine if there is a valid type argument. We use Figure 2 to track these assumptions and requirements, and `bind.upcast`'s stated constraint on `B` forms Requirement e) in that figure. But to determine the remaining assumptions and requirements, one must first consider the arguments to the generic method.

We supply `bind.upcast` with the first argument `constrain` of type `Constrain<U,? super T>`. This type has a wildcard type argument, so Java captures the wildcard by introducing a fresh type variable, say `Z`. This wildcard is explicitly

a) Assumption	`T <: Z`	Explicit constraint on wildcard
b) Assumption	`Z <: U`	Implicit constraint on wildcard
c) Requirement	`B ≡ Z`	`constrain` is a `Constrain<U,Z>`
d) Requirement	`T <: B`	`t` is a `T`
e) Requirement	`B <: U`	Stated by `bind.upcast`

Figure 2. Type-argument inference for Figure 1

constrained to be a supertype of `T`, so Java declares that `Z` is assumed to be a supertype of `T`, forming Assumption a) in Figure 2. This wildcard is also implicitly constrained to be a subtype of `U`, so Java declares that `Z` is assumed to be a subtype of `U`, forming Assumption b) in Figure 2.

Java then type-checks `bind.upcast(constrain, t)` using `Constrain<U,Z>` as the type of `constrain`. Java uses that type to determine that the type parameter `B` of `upcast` must be syntactically identical to the type argument `Z`, forming Requirement c) in Figure 2. Java then determines that the type of `t` must be a subtype of `B`, forming Requirement d) in Figure 2. At this point, all the assumptions and requirements have been gathered, and the invocation type-checks if there exists a type argument for `B` that satisfies the requirements in Figure 2 under the assumptions in that figure.

Requirement c) informs us that there is at most one such type argument: `Z`. Furthermore, the remaining requirements correspond exactly to the assumptions. That is, the explicit constraint on the wildcard in `Constrain<U,? super T>` implies `t` is a valid second argument, and the implicit constraint on that wildcard implies the constraint stated by `bind.upcast` is satisfied. As a consequence, the method type-checks. Furthermore, even though type-argument inference in Java is undecidable, this type argument is identified by `javac`, version `1.8.0_25`, which consequently accepts and compiles the code in Figure 1. However, constraint solving is generally a non-deterministic process, and type-argument inference itself is non-deterministic [35], so the Eclipse Compiler for Java, `ecj` version `3.11.1.v20150902-1521`, and the current compiler for the upcoming Java 9, `javac` build 108, fail to type-check this code. They consider Requirements d) and e) first, instead of Requirement c), and recognize that the invocation can only type-check if `T` were a subtype of `U`. Intuitively, this is impossible, but this example shows that this intuition is incorrect. Note that their rejection of the unsound code in Figure 1 is merely an accident of the non-deterministic nature of constraint solving. They still accept the code in Figure 3, which is also unsound for similar reasons. On the other hand, the code in Figure 4 is an unsound Java program that all 3 compilers reject even though it is valid, illustrating how inconsistent the type-argument inference algorithms are.

Note that `bind.upcast` can only be executed to do the unsound coercion if we can actually provide a value of type `Constrain<U,? super T>`. Without knowing that `T` is a subtype of `U`, it is impossible to create such an instance.

```
class Unsound9 {
  static class Type<A> {
    class Constraint<B extends A> extends Type<B> {}
    <B> Constraint<? super B> bad() { return null; }
    <B> A coerce(B b) {
      return pair(this.<B>bad(), b).value;
    }
  }
  static class Sum<T> {
    Type<T> type;
    T value;
    Sum(Type<T> t, T v) { type = t; value = v; }
  }
  static <T> Sum<T> pair(Type<T> type, T value) {
    return new Sum<T>(type, value);
  }
  static <T,U> U coerce(T t) {
    Type<U> type = new Type<U>();
    return type.<T>coerce(t);
  }
  public static void main(String[] args) {
    String zero = Unsound9.<Integer,String>coerce(0);
  }
}
```

Figure 3. Unsound valid Java program compiled by `javac`, JDK9 build 108, and `ecj`, version `3.11.1.v20150902-1521`

So type-checking the method invocation could actually be sound since it might be unreachable. But in Java `null` inhabits *every* reference type, a property we call *implicit nulls*, as opposed to having a type be inhabited by `null` *only if* the type explicitly declares so, a property we call *explicit nulls*. Thus, although it is impossible to create an instance of `Constrain<U,? super T>`, one can simply use `null` to bypass the critical assumptions that were used to argue the soundness of these features, most notably implicit-constraint generation in wildcard capture.

4. Scala's Path-Dependent Types

Scala also has both parametric and subtype polymorphism. But to illustrate unsoundness, we rely more on its main feature: path-dependent types.

Path-dependent types allow values to have individualized types associated with them. For example, a graph object can have a *type member* that indicates the type of its vertices. A type member is denoted as in the following trait (where a trait is, for our purposes, akin to an interface):

```
trait Graph {
  type Vertex;
  def getNeighbors(v : Vertex) : List[Vertex]
}
```

This code does not specify what the implementation of `Vertex` is; it simply indicates that the `Graph` object has some

```
class UnsoundSpec {
  static class Constrain<A, B extends A> {}
  static <A,B extends A>
  A upcast(Constrain<A,B> constrain, B b) {
    return b;
  }
  static <T,U> U coerce(T t) {
    Constrain<U,? super T> constrain = null;
    return upcast(constrain, t);
  }
  public static void main(String[] args) {
    String zero = coerce(0);
  }
}
```

Figure 4. Unsound valid Java program

type of vertices. This type might be indices used to index an adjacency matrix, or it might be a class whose instances store a list of neighbors.

Given an immutable variable `g` of type `Graph`, there is an associated path-dependent type `g.Vertex` representing the type of the vertices of the graph `g`. This type is truly coupled with the specific variable `g`. Given another immutable variable `g2` of type `Graph`, values of type `g2.Vertex` cannot be used as values of type `g.Vertex` and vice versa. Thus, the implementation of `g` can guarantee that a vertex passed to `g.getNeighbors` is necessarily a vertex of `g` rather than some other `Graph`.

There are a few more details that need to be mentioned. As with type parameters, type members can be constrained to indicate they must be a subtype of something and/or a supertype of something. Also, Scala traits, being similar to interfaces, can be inherited multiple times, so Scala allows traits to be combined using the keyword `with` to represent values that inherit both traits. With these final details, we can demonstrate that Scala's type system is unsound.

5. Unsoundness of Scala

Now that we know the relevant features of Scala, we use them to create a method `coerce` in Figure 5 that can turn any type into any type without (down)casting. This code type-checks according to the Scala Language Specification [27] and is compiled by `scalac`, version `2.11.7`. Stepping through the code, one can see that `coerce` simply returns the value of its argument `t`. When executed on the JVM, a `ClassCastException` is thrown inside the `main` method just as with our Java example.

In order to understand the example, the critical type to look at is `LowerBound[T] with UpperBound[U]`. Instances of this type have a type member `M` that is *both* a supertype of `T` (because the instance satisfies `LowerBound[T]`) and a subtype of `U` (because the instance satisfies `UpperBound[U]`).

```
object unsound {
  trait LowerBound[T] {
    type M >: T;
  }
  trait UpperBound[U] {
    type M <: U;
  }
  def coerce[T,U](t : T) : U = {
    def upcast(lb : LowerBound[T], t : T) : lb.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)
  }
}
```

Figure 5. Unsound valid Scala program compiled by `scalac`, version `2.11.7`

Next, consider the function `upcast`. It takes a value whose type member is a supertype of `T`, and a value of type `T`, and then upcasts the latter value to the former's type member, taking advantage of the supertype constraint.

When we call this function using a value that is a `LowerBound[T]` *and* an `UpperBound[U]`, its type member happens to also be a subtype of `U`, so the returned value can subsequently be used as a `U`.

Note that this reasoning assumes that we can actually provide a `LowerBound[T] with UpperBound[U]`. Without knowing that `T` is a subtype of `U`, it is impossible to create such an instance. So this reasoning could all be sound. *But*, Scala also has implicit `null` values. So, although it is impossible to create such an instance, one can simply use `null` to bypass the critical assumptions that were used to argue the soundness of these features.

6. Lessons to be Learned

We have demonstrated that two major industry languages are unsound, both 12 years after having been released. Both of these languages have had significant involvement from academics, especially with respect to the features used in this paper. The following are some lessons to be learned from this experience.

6.1 Tolerating Nonsense Types

It might not make sense for algorithmic subtyping to be transitive in all settings. For example, suppose $\bot$ is a subtype of everything, and that $\top$ is a supertype of everything. It is unreasonable to expect an algorithm to report that `Integer` is a subtype of `String` whenever the context has a type variable `V` that is specified to be a supertype of $\top$ and a subtype of $\bot$ (note the reversal). Neither `Integer` nor `String` reference `V`, so one would not expect the algorithm to consider the bounds on `V`. Nonetheless, for algorithmic subtyping to be transitive, it would have to do precisely this due to the following derivation:

$$\texttt{Integer} \leq \top \leq \texttt{V} \leq \bot \leq \texttt{String}$$

Type variables like `V` can arise from what we call "nonsense" types. For example, `V` and its inconsistent constraints would arise from capturing the wildcard in the pseudo-type `Constrain<⊥,? super ⊤>` using the `Constrain` class from Figure 1.

Often one imposes restrictions on types to disallow nonsense types, where nonsense is whatever the designer at hand considers it to be. But this strategy can be fickle. For example, suppose a subtyping algorithm was proved correct under the assumption that all types in the derivation were valid. But type validation is an algorithm itself, and one that often relies on subtyping. Furthermore, due to designs such as F-bounded polymorphism [7], the relevant subtyping derivation may reference the type being validated. Thus we get a circular dependency between type validation and subtyping.

Another issue is that types arise from intermediate type-checking algorithms. For example, suppose we had intersection types and an invariant class `Array`. One might require all types to be a subtype of at most one instantiation of `Array` in order to deduce type equivalences. That is, no type should be a subtype of `Array<String>` and `Array<Integer>`, since it is impossible to belong to both. But consider the following pseudo-code, where `A` and `B` are type variables:

```
interface Property<T> { boolean holdsFor(T t); }
Array<A> as = ...;
Property<? super Array<A>> propa = ...;
Property<? super Array<B>> propb = ...;
boolean holds = (... ? propa : propb).holdsFor(as);
```

When determining the type of the conditional expression `(... ? propa : propb)`, the type-checker computes the join, i.e. least common supertype, of `Property<? super Array<A>>` and `Property<? super Array<B>>`. With intersection types, this join is `Property<? super Array<A>∩Array<B>>`. This contains the type `Array<A>∩Array<B>`, which violates the restriction because it is a subtype of both `Array<A>` and `Array<B>` by the definition of type intersection. This nonsense type is never written by the programmer but instead arises as a result of the join algorithm. If the algorithms used to type-check `holdsFor` are written assuming such nonsense types cannot exist, they might mistakenly and unsoundly accept the last line of code in the example.

Note that `Array<A>∩Array<B>` is inhabitable — the variable `A` could represent the same type as `B`. So nonsense types are not the same as uninhabitable types. They are simply violations of the (often implicit) assumptions of how the designer expects types to be used. But nonsense types might not always be written directly by programmers, and they may not be easy to identify. An insistence on preventing nonsense types is what most delayed the development of a core

calculus for Scala because nonsense types can so easily arise from even the core features of the language [2].

Thus, if one wants to reject nonsense types, they must do two things. First, they must check that the algorithm for identifying nonsense types works correctly even if nonsense types arise as the algorithm progresses, or prove that all types that arise during this process are in fact sensible. Second, they must prove that all types constructed intermediately during compilation are themselves sensible assuming the types they were constructed from are sensible. This is easily accomplished for many kinds of nonsense. For example, one can ensure that all names are in scope, and every use of a generic class has the correct number of type arguments.

But many of the more complex forms of sensibility, such as type-argument validation, do not easily satisfy these criteria. For such cases, the alternative we propose is simple: allow nonsense types except where soundness specifically relies on sensibility. For example, type-argument validation only needs be done when creating a `new` instance of a generic class and when inheriting a generic class. This is because only the implementations of the methods of the new instance rely on the specified requirements of the type arguments. Everything else simply relies on the fact that all instances have sound implementations of their methods. This can be proved by abstracting the signature and inheritance of all classes of a program into interfaces with no constraints on their type parameters. Except for `new` and inheritance, every mention of a class can be replaced with its associated interface and the program will still type-check and have the same semantics. Thus, `new` and inheritance are the only places that need type arguments to satisfy the constraints on the respective type parameters.

Language designers should consider both how to reject nonsense types *and* how to accept them before deciding on which approach to take. Different strategies are appropriate for different problems. In fact, changing from the former strategy to the latter strategy is what finally enabled the development of sound models for Dependent Object Types (DOT) [1, 30], the minimal calculus for Scala. The examples in this paper were found by applying experience with nonsense types to identify a point in the soundness proof for this minimal calculus that fails to generalize to null values. Thus understanding rather than ignoring nonsense types is critical to preventing unsoundness.

6.2 The Billion-Dollar Mistake

Implicit nulls, a key component of our examples, were invented by Tony Hoare, who refers to this invention as his billion-dollar mistake [17]. The feature has been a cause of many bugs in software. It adds a case that is easy to forget and difficult to keep track of and reason about. Interestingly, here it causes the same problem for the same reasons, but at the type level. The reasoning for wildcards and path-dependent types would be perfectly valid if not for implicit null values.

Many already would like to remove implicit nulls from industry. This finding gives us more impetus to do so. As type systems become more advanced, it becomes more important for types to provide guarantees that are not invalidated by implicit nulls.

Null values have their uses though, so simply casting them aside would be naïve. Instead, we need to understand their uses and provide features that sufficiently accommodate those use cases. For example, implicit nulls are useful for representing the absence of a returned value, the absence of an optional argument, unknown values, uninitialized fields, and exceptional behavior.

Our community has a variety of solutions for each of these use cases. Now is a good time to collaborate with industry teams to put these solutions into practice, since recent industry languages such as Ceylon, Kotlin, and Rust have already made nulls explicit in reaction to the problems that implicit nulls have caused [5, 22, 31]. But each solution our community provides has its own advantages and disadvantages, making different solutions appropriate for different languages. Our community should build better relationships with industry so that design teams know who might be interested in helping and would help them identify the solution best suited for the language at hand rather than the contact's own preferred solution. If our community wants to remove implicit nulls from industry, where appropriate, our community should make greater effort to appreciate the complex compromises that have to be made in practice so that our work can successful transfer out of academia.

6.3 Unforeseen Interactions

There have been many formalizations of subsets of and extensions to Java [3, 4, 6, 9, 10, 18, 26, 38], many of which were proved sound, but none of which found this error. The reason is that none of them handle the full feature set of Java.

Earlier work on Java had the goal of proving Java sound, making an effort to handle large subsets of Java [9, 26]. In particular, they made a point of considering features that already had a history of unsoundness, with mutable state being the most notorious such feature. But even these extensive formalizations considered only a subset of the Java language, still making them incomplete, and the Java language has since become even more complex.

The problem is that Java, along with many other major industry languages, is extremely large. It provides features for efficiency, for concurrency, for convenience, for maintainability, for interoperability, for security, for transportability, for serialization, for debugging, for fault tolerance, and so on. At present, researchers simply do not have the time to formalize languages and prove valuable theorems at this scale, or even to communicate these formalizations to each other. Furthermore, we need to explore new possibilities in addition to inspecting current standards, and we need to consider far more options than can ever become reality.

So an effort was made to minimize formalizations to just the subset of Java necessary to capture the core features of the language, making it easier to design extensions to Java and prove their soundness. This effort culminated in Featherweight Java (FJ) [18]. To demonstrate its extensibility, Igarashi et al. developed an extension of FJ with generics, resulting in Featherweight Generic Java. Wild FJ [38] and TameFJ [6] then extended Featherweight Generic Java with wildcards and existential types capable of expressing both explicit and implicit constraints. The concision of these calculi is what enabled our community to quickly develop and reason about these and many other extensions to Java. And, with the growth of proof assistants, minimization has become a valuable tool for addressing the massive effort involved in mechanical formalization and verification of these languages and extensions.

Unfortunately, in minimizing Java to a core calculus, null pointers were dropped. Every extension afterwards focused on proving the soundness of its new feature, so its creators did not add the feature of null pointers that was already deemed uninteresting. As a consequence, none of these extensions have been proved compatible with null pointers. And in fact, what we have just demonstrated is that both Wild FJ and TameFJ are not compatible with null pointers.

The issue is that inhabitants of existential types provide evidence that some type satisfying certain constraints exists, but null pointers provide no such evidence. Wild FJ, TameFJ, and Java make use of those constraints to implement use-site variance. But by making null pointers implicit, existential types being inhabited no longer guarantees anything. In our examples, we create a type whose evidence implies `T` is a subtype of `U` and use this evidence to convert from one to the other. We then falsify this evidence with a null pointer to instantiate the unsoundness. Thus, the unforeseen interaction of features has caused Java and Scala to be unsound.

6.3.1 On Methodology

The question at hand is how such critical oversights might be prevented in the future. This is not the first time an interaction of features has posed some of the most interesting challenges for a language. Possibly the most famous example is the interaction of parametric polymorphism and mutable state. Gordon, Milner, and Wadsworth actually identified this interaction in their initial publication of ML [11], which considers the full language in depth. Yet it took a decade to arrive at the modern solution known as the value restriction [37]. Still later, a similar interaction between mutable state and existential quantification led to unsoundness in Cyclone [15], which is important to Java's design for wildcards [34]. More specific to object-oriented programming, the interaction of subtyping and inheritance made Eiffel [24] unsound [8].

Complications are not always in the form of unsoundness. The interaction of parametric polymorphism and recursion makes type inference undecidable unless polymorphic recursion is disallowed [16, 21, 23, 25]. The interaction of subtyping and parametric polymorphism makes subtyping undecidable [28], even without impredicative polymorphism [20], and even without multiple-instantiation inheritance [14].

The issue is that we do not have some oracle that can tell us whether a minimization unintentionally elided a critical feature with unexpected interactions. And yet, we cannot afford to regularly handle the entirety of a language such as Java. Had Wild FJ and TameFJ been derived from Flatt et al.'s ClassicJava [10], they would have identified these concerns, but this would have also required more work without it being clear that this additional work would come with additional benefits.

Here we consider some possible methodologies that might help identify problematic interactions of features ahead of time without hindering the advancement of our field. These methodologies are not meant to be replacements for minimal calculi. Different methodologies are appropriate for different settings, and minimal calculi have already proven their value to our community. But we hope our community will explore the methodologies below and encourage the ones that seem successful and efficient. This is meant to start a discussion, or give stronger voice to an ongoing one, so we use Featherweight Java (FJ) as a running example to illustrate the methodologies and their potential place within this discussion. Featherweight Java is an interesting case because it, along with Generic Featherweight Java, is itself compatible with implicit nulls; only in extending it do these incompatibilities arise. So ideally these methodologies can help forecast problematic features for not only the calculus at hand, but even the extensions built upon that calculus.

Threats to Validity When reporting an experimental evaluation, it is standard practice in many communities to report the "Threats to Validity". That is, no experiment can provide a guarantee, and one must explicitly discuss how the given experimental results might fail to generalize. We could apply a similar standard to minimization: one must explicitly discuss the properties of the minimization that are most likely to fail to generalize, as well as which features are most likely to present unforeseen complications.

In the case of FJ, a threat to validity is that it has different values from Java, which has significant effect on the induction principles of the proof. `null` is one notable Java value that is absent from FJ, so the calculus and its soundness may not generalize to that feature. In this way, when researchers build upon a minimal calculus, they might have a better idea of which features absent from the calculus they should still consider in more detail, if only informally.

Open-Ended Calculi An alternative is to develop "open-ended" minimal calculi and proofs. That is, develop the syntax, typing rules, and semantics with the understanding that each of these components is incomplete. Similarly, prove the desired properties of the open-ended calculus without assuming the given operations in the calculus are the only op-

erations in the calculus. Of course, this is impossible to do in general, so one would need to specify the assumptions made about the remaining unknown features. These assumptions formally expose much of the threat to validity of the minimization, as well as allow researchers to know when they can simply reuse the proof without any customization. Note that one can state assumptions that are more general than is necessary to prove soundness of the closed minimal calculus. Thus this methodology might allow one to develop composable minimal calculi and proofs that one could be more confident will generalize to the full language.

In the case of FJ, this would first amount to adding ... to the grammar, typing rules, and semantics. Then, for the soundness theorem, one would add the requirement that each typing rule in ... concluding $\Gamma \vdash e : \tau$ can show that e reduces soundly given that each of the assumptions of the rule reduces soundly. Here reducing soundly means not getting stuck *and*, if reducing to a value, then then that value has type τ. Note that this requirement is stated in terms of values. In the case of FJ, the only values are class instances. But Java has more values. For example, Java has primitive values. But primitive values can never have a class type, making it clear that they would have to be wrapped to interact with FJ, and consequently these wrappers would satisfy the requirements of FJ and thereby be sound. On the other hand `null` can have any class type. Thus the open-ended methodology would inform the researchers that they should consider `null` at least informally, and then either add it to the calculus to strengthen its representation of the full language, or state it as a threat to validity for subsequent researchers to keep in mind. This methodology might clarify what exactly is guaranteed in the more general case, might better illustrate why the system is sound, and might simplify the process for extending to more features or adding new features by simply proving that the invariant is maintained with the new cases.

Minimizing to Cross-Cutting Features An entirely different approach is to change the goal of minimization. Whereas traditionally minimization has removed the complex features, one could alternatively minimize to remove the features with the least cross-cutting impact. That is, if a feature is simply another case in a proof, with no effect on the design of the proof, its invariants, or its guarantees, then remove it. This might significantly reduce the capability of the calculus, but such a reduction does not matter if that capability is not relevant to the question at hand. In this way, the discussion of the work could focus on the most challenging aspects of the language or proof, which the researchers' expertise is most relevant to, rather than the aspects that might easily be recreated by the reader or successor.

These two approaches will often disregard many of the same features. Access modifiers have no effect on evaluation and only restrict typability. Exceptions simply bypass computation, much like non-termination, and use and reduce to

```
class SingleParameters<Ignore> {
  static class Bind<A> {
    class Curry<B extends A> {
      A curry(B b) { return b; }
    }
    <B extends A>
    Curry<B> upcast(Constraint<B> constraint) {
      return new Curry<B>();
    }
    class Constraint<B extends A> {}
    <B> A coerce(B t) {
      Constraint<? super B> constraint = null;
      return upcast(constraint).curry(t);
    }
  }
  public static void main(String[] args) {
    Bind<String> bind = new Bind<String>();
    String zero = bind.<Integer>coerce(0);
  }
}
```

Figure 6. Unsound valid Java program with only single-parameter methods and single-type-parameter classes compiled by `javac`, version `1.8.0_25`

the same values as exceptionless programs in Java. So in the case of FJ, both these features would be ignored by both minimization strategies for soundness. However, in Java constructors are simply an initialization of all fields to `null` followed by a method initializing the fields, and methods with multiple parameters can be translated to single-parameter methods using currying. Even type parameters for classes can be curried, and Figure 6 shows that multiple parameters are unnecessary for unsoundness. On the other hand, `null` adds an exceptional case to the invariant of the proof, one that must be explicitly handled by the semantics of the language. Furthermore, `null` and mutable state are crucial to the design of Java, affecting how various features are implemented (such as constructors), and which features are omitted (such as closures that can access all variables in scope).

Thus in the case of FJ, this difference in methodology might amount to reintroducing `null` and mutable state, and removing constructors and simplifying methods to have exactly one parameter. In fact, this difference is necessary to accurately model object creation in Java, or at least the aspect of it that would be most concerning for soundness. In particular, because constructors can invoke methods in Java, those methods can unknowingly access fields that have not been initialized. Such fields can even be declared `final`, which causes the type-checker to guarantee that it is initialized by the constructor and exactly once. This is why Java actually initializes *all* (reference) fields to `null` and then mutates them when initialized by the constructor. This can create a back door in the type system. Our examples can be rewritten to have no use of the expression `null` and instead

```
class Nullless<T, U> {
  class Constrain<B extends U> {}
  final Constrain<? super T> constrain;
  final U u;
  Nullless(T t) {
    u = coerce(t);
    constrain = getConstrain();
  }
  <B extends U>
  U upcast(Constrain<B> constrain, B b) {
    return b;
  }
  U coerce(T t) {
    return upcast(constrain, t);
  }
  Constrain<? super T> getConstrain() {
    return constrain;
  }
  public static void main(String[] args) {
    String zero = new Nullless<Integer,String>(0).u;
  }
}
```

Figure 7. Unsound valid Java program without `null` compiled by `javac`, version `1.8.0_25`

access an uninitialized field, as shown in Figure 7. Thus `null` and mutable state are both complexities for which soundness is interesting. A proof of soundness for this system would be much more likely to generalize, since this system has the same values as Java, addresses state, and more accurately reflects object initialization in Java. It would also provide guarantees for the most worrisome cases, recognizing that `null` is an exceptional expression in the system and that object initialization in Java has weak guarantees.

Hindsight It is important to reiterate that in all this discussion we might simply be applying hindsight to claim solutions for a tough problem. The current methodology using minimal calculi makes no such claim, and yet has been very effective for our community. Nonetheless, while much research should stick to such well understood methodologies, we hope some research will explore new methodologies for this problem, and we hope our community will accommodate and encourage this exploration. Over time we might be able to discern which methodologies or combinations thereof, if any, can effectively address this problem. Meanwhile there are issues besides methodology that should also be considered in this discussion.

6.3.2 On Mechanical Verification

More recently, one huge advantage of minimal systems is that they can be mechanically verified with a reasonable, albeit still high, amount of effort. So advancements in mechanical verification might be necessary to reasonably extend this standard and the guarantee it provides to more complex systems. Incorporation of nominal logic [29] might reduce much of the tedium in verifying these systems. Specialization for finite types and types with decidable equality, including especially functions from those types, might enable simpler reasoning about finite contexts and finite-arity operations. First-class treatment of the functors whose fixpoints form inductive and coinductive structures might improve the extensibility of our calculi and their proofs. In general, specialization of mechanical proof systems to common paradigms in our field might allow us to immediately focus our effort on the interesting aspects of the problem at hand, similar to what we do in our informal proofs.

6.3.3 On Values

There are some cultural issues to consider as well. Our community values simplicity. Simplicity makes concepts easier to communicate and comprehend, theorems easier to prove and confirm, and fundamentals easier to separate from accidentals. This value in itself is not problematic. The problem is that, while valuing simplicity, our community must also recognize the need for complexity in many situations. Not all problems are best simplified, especially those tied to complex systems such as industry. And while most of our community's research should not be tied to such complex systems, there is room to encourage more research along this front so that the present and future languages used by millions of people around the world might live up to our ideals. Many other fields have been able to interact with, contribute to, appreciate, and benefit from industry despite its complexities and without being limited by it. Our field should strive for the same balance.

6.3.4 On Process

Lastly, our review and publication process seems biased against anticipating the interaction of features. It simply takes more space to present a larger calculus, or to formalize and discuss the assumptions of an open-ended calculus. And yet our most prestigious publication venues permit the same amount of space for all work, regardless of how much space is appropriate for the work. This biases against thorough considerations of interactions, and it favors minimal closed solutions with no guarantee of composability. It also encourages our community to stick with what is familiar and needs less explanation, rather than branching out both in terms of what tools we can apply and what problems we can apply our own techniques to. Of course there are many good reasons for page limits, so a solution to this bias would need to take many concerns into consideration.

Not everything is simple, and our community must figure out how to encourage complexity when appropriate. Java has been the first- or second-most used programming language for over 15 years, 12 of which has been with generics, and yet we still do not have a formalization of it that has both null pointers and generics, let alone wildcards.

7. Potential Fixes

Here we speculate how to fix the fundamental sources of unsoundness that we have identified for Java and Scala. These languages are widely used, so backwards compatibility is important. Of course, any fix to unsoundness cannot in theory be backwards compatible, so one must assess and account for how these languages are used in practice. Per the discussion in Section 6, we focus on solutions that can tolerate nonsense types.

In Java, the unsoundness is caused by wildcard capture specifically during type-argument inference. Although wildcard capture is used during subtyping, it is only used in situations where a value in question being null automatically implies subtyping holds anyways. Type-argument inference is the only feature that captures a wildcard without the corresponding value being used when the wildcard's evidence is used. Thus, it is both necessary and sufficient to either reject a method invocation or insert null checks when a captured wildcard's evidence introduces a subtyping (not involving the fresh capture variables) that would not otherwise hold.

Determining when evidence introduces a potentially problematic new subtyping is challenging algorithmically because capture can introduce subtypings between fresh capture variables and these still need to be accepted. The fact that these constraints can be recursive, meaning a fresh capture variable is constrained by itself, makes it challenging to separate these subtypings from the subtypings they imply that are not in terms of fresh variables. So one solution is to restrict wildcards so that capturing them never introduces new subtypings. This could be done by not using implicit constraints, but implicit constraints are actually used in practice on occasion. It could be done by not allowing `? super` when the wildcard has an implicit constraint, which Tate et al. determined is extremely rare in practice [35]. Another solution is to only introduce implicit constraints between what Greenman et al. call "materials" [13], which they have determined would be almost certainly be backwards compatible in practice. These constraints are not recursive, which would make it easier to determine if they introduce a new subtyping. Regardless, there is likely to be a backwards-compatible-in-practice solution with no or minimal run-time overhead.

In Scala, the unsoundness is caused by using a path-dependent type where the path contains a null pointer. Unfortunately, path-dependent types give an explicit name to the possibly non-existent type. This enables inconsistencies to be developed and exploited across the program, rather than in a single place like with wildcards. Therefore, the best solution is most likely to check that paths are valid at run time, using compile-time analysis to reduce these checks. The fact that null-pointer checks are already well optimized in the JVM hopefully implies that this might still produce only light run-time overhead despite the frequency of the checks.

Moving forward, we are currently discussing the issues with the teams for both languages. The Java team already had concerns about the types we used [32]. This work has demonstrated that those types can in fact lead to unsoundness. However, the discussion in Section 6 suggests that the solution might be to amend type-argument inference rather than disallow such types. The team plans to adopt one of the aforementioned potential solutions, but at this point needs to investigate more to determine which solution is best. The Scala team has identified yet more sources of unsoundness in Scala closely related to our example but using more advanced features in place of null pointers. Because there currently seems to be no principled solution to all these sources, they have deferred fixing the issues, for now relying on the JVM to catch the complex corner cases at run time.

8. Conclusion

We have shown that Java and Scala are unsound and have been for 12 years. Their unsoundness is due to an unforeseen interaction of features that are each sound separately. The unsoundness was missed repeatedly due to our community's reliance upon minimal calculi; mechanically verified proofs would not have prevented the problem. Any language with types that provide evidence without directly using the corresponding value is susceptible to this issue. Given that this issue was missed even after significant formalization effort, it is likely that many less-thoroughly-examined hypothetical languages and features in our literature share the same problem. This suggests that we as a community need to encourage more research on holistic language design and formalization so that we can prevent such problems in the future.

Acknowledgments

We are extremely grateful to the many people who contributed to this paper. The anonymous reviewers pushed us to present a more thorough and well rounded discussion of the lessons. The Working Group on Language Design made this possible by bringing us together and providing a venue that so successfully encourages the sharing of thoughts, both deep and whimsical. Gavin Bierman and Dan Smith were extremely helpful with understanding and representing Java, its team, and its history. Similarly, Martin Odersky, Dmitry Petrashko, and Tiark Rompf offered valuable insights into Scala, its team, and its history. Last but not least, Jonathan Aldrich, Owen Arden, Kim Bruce, Chris Casinghino, Derek Dreyer, Sophia Drossopoulou, Paul Ebermann, Nate Foster, David Herman, Andrew Hirsch, Chin Isradisaikul, Tom Magrino, Matthew Milano, Greg Morrisett, Andrew Myers, Craig Riecke, Talia Ringer, Sam Tobin-Hochstadt, and Lucas Werkmeister made this paper actually readable and provided significant perspective into the discussion and the history of the relevant events (many of which took place before we started our Ph.D.'s).

References

[1] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In *WadlerFest*, 2016.

[2] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *OOPSLA*, 2014.

[3] Gavin M. Bierman, Matthew J. Parkinson, and Andrew M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, University of Cambridge, April 2003.

[4] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, 1998.

[5] Andrey Breslav. The Kotlin language documentation, February 2016.

[6] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A model for Java with wildcards. In *ECOOP*, 2008.

[7] Peter S. Canning, William R. Cook, Walter L. Hill, Walter G. Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, 1989.

[8] William R. Cook. A proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):305–311, August 1989.

[9] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *TAPOS*, 5(1):3–24, January 1999.

[10] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL*, 1998.

[11] Michael Gordon, Arthur L. Miller, and Christopher P. Wadsworth. Edinburgh LCF: A mechanised logic of computation. *LNCS*, 78, 1979.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley Professional, third edition, 2005.

[13] Ben Greenman, Fabian Muehlboeck, and Ross Tate. Getting F-bounded polymorphism into shape. In *PLDI*, 2014.

[14] Radu Grigore. Java generics are Turing complete. *arXiv*, 2016.

[15] Dan Grossman. Existential types for imperative languages. In *ESOP*, 2002.

[16] Fritz Henglein. Type inference with polymorphic recursion. *TOPLAS*, 15(2):253–289, April 1993.

[17] Tony C. Hoare. Null references: The billion dollar mistake. QCon, March 2009.

[18] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, May 2001.

[19] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *ECOOP*, 2002.

[20] Andrew J. Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. In *FOOL*, 2007.

[21] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Computational consequences and partial solutions of a generalized unification problem. In *LICS*, 1989.

[22] Gavin King. The Ceylon language specification, version 1.0, November 2013.

[23] Lambert Meertens. Incremental polymorphic type checking in B. In *POPL*, 1983.

[24] Bertrand Meyer. Genericity versus inheritance. In *OOPSLA*, 1986.

[25] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, 1984.

[26] Tobias Nipkow and David von Oheimb. Java_{light} is type-safe — definitely. In *POPL*, 1998.

[27] Martin Odersky. The Scala language specification, version 2.9, May 2010.

[28] Benjamin C. Pierce. Bounded quantification is undecidable. In *POPL*, 1992.

[29] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.

[30] Tiark Rompf and Nada Amin. From F to DOT: Type soundness proofs with definitional interpreters. Technical report, Purdue University, October 2015.

[31] The Rust Community. The Rust programming language, May 2015.

[32] Dan Smith. Possibly modify well-formedness restriction for wildcards?, August 2014. `https://bugs.openjdk.java.net/browse/JDK-8054941`.

[33] Christopher Strachey. Fundamental concepts in programming languages. Lecture Notes for the International Summer School in Computer Programming, August 1967.

[34] Alexander J. Summers. Modelling Java requires state. In *FTfJP*, 2009.

[35] Ross Tate, Alan Leung, and Sorin Lerner. Taming wildcards in Java's type system. In *PLDI*, 2011.

[36] Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *ECOOP*, 1999.

[37] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[38] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In *FOOL*, 2005.

[39] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *Symposium on Applied Computing*, 2004.

[40] Geoffrey Alan Washburn. Another type soundness hole, December 2008. `https://issues.scala-lang.org/browse/SI-1557`.