

Safe Instantiation in Generic Java

Eric E. Allen Robert Cartwright
Sun Microsystems, Inc. Rice University
`eallen@cs.rice.edu` `cork@cs.rice.edu`

March 19, 2004

Abstract

This paper presents the “Safe-Instantiation Principle,” a new design principle for evaluating extensions of Java with support for generic types. We discuss the GJ and NextGen formulations of Generic Java and the implications of safe instantiation on both approaches. We then consider the implications of safe-instantiation for the addition of mixins to Java via generic types. Finally, we defend the formulation of mixins as *hygienic* program constructs, arguing that a hygienic formulation is the only way to maintain safe instantiation and type soundness in Java with mixins, and to prevent the introduction of insidious bugs with no clearly defined point of blame.

1 Introduction

Despite its clear advantages over competing mainstream languages such as C++, the Java platform is still in its early stages of evolution. From the perspective of both software engineering and programming pedagogy, Java has a crude type system. Its most significant failing is the lack of support for generic types. As has been shown in [5, 9, 10, 12], this omission restricts the range of abstractions that programs can express and the precision of static type annotation and checking. Furthermore, as shown in [6], excluding run-time type information in Generic Java (via GJ-style “type-erasure”) would severely limit the advantages of adding generic types, as compared to approaches such as the NextGen formulation of Generic Java, in which generic type information is maintained at run-time. We have extended the NextGen type system to support the inclusion of naked type parameters in any context where ordinary types may appear, including the `extends` clauses of class definitions, in [3]. The extended language has been formally modeled and proven sound, and a strategy for compatibly implementing the language on the JVM has been presented in [3]. This new language has been dubbed MixGen because it effectively allows class definitions with parameterized parent type, language constructors traditionally referred to as “mixins”. In this paper, we present the *Safe Instantiation Principle*, a new criterion for evaluating various potential extensions of Java with generic types, and we evaluate the GJ, NextGen, and MixGen languages with respect to this criterion.

2 The Safe Instantiation Principle

In order to motivate the Safe Instantiation Principle, consider a situation in which an applications programmer is attempting to make use of a generic class $C<T>$ that he does not maintain. Because Java supports separate class compilation, the only constraints on instantiations of T that are accessible by the client programmer are the declared constraints, *e.g.*, an upper type bound on instantiations such as T `extends Comparable`. Also because of separate compilation, it is impossible when compiling a generic class to determine all of the instantiations of C that will occur at run-time. If a client programmer instantiates $C<T>$ with a type argument that satisfies the declared constraints but still causes an error such as a `ClassCastException`, the client programmer may be left with no recourse but to report a “bug” to the maintainer of C . But if the maintainer of C is unable to specify constraints that prohibit such instantiations, he is powerless to prevent C ’s misuse. Informal documentation may help to some extent, but if the error caused by an instantiation has no immediate symptoms, diagnosis of the problem once it is noticed can be exceedingly difficult. Similar difficulties with error diagnosis have provided some of the most powerful arguments for type safety and against unsafe operations in languages such as pointer arithmetic and manual storage management. For this reason, we propose that any extension of Java that supports generic types should adhere to the following principle:

***The Safe Instantiation Principle:** Instantiating a parametric class with types that meet the declared constraints on the parameters should not cause an error.*

Fundamentally, the Safe Instantiation Principle is a prerequisite for type soundness in Generic Java. If a generic type system does not enforce it, then the value of static type checking is substantially reduced, just as it is reduced by unchecked template expansion in C++. But although this principle may appear hard to disagree with, it is actually missing in many generic extensions of the Java type system [7, 9, 10, 12]. In fact, this principle is deceptively difficult to adhere to in Generic Java. In the remainder of this paper, we will examine three formulations of Generic Java and discuss the implications of safe instantiation on each of them.

3 GJ, Type Erasure, and Safe Instantiation

In the GJ formulation of Generic Java, which forms the basis for the JSR-14 prototype compiler from Sun Microsystems and is scheduled for inclusion as part of J2SE v1.5, generic type information is kept only for static type checking. After type checking, all generic types are “erased” and are therefore unavailable at run-time. For this reason, type-dependent operations such as casts, `instanceof` checks, and `new` expressions are not safe in general in GJ [10]. As was demonstrated in [6], the lack of run-time type information in GJ significantly impairs the expressiveness of Generic Java. However, an orthogonal problem with GJ is that it fundamentally violates safe instantiation. Although the compiler issues a warning when a generic class includes unsafe operations, the programmer is not prevented from producing unsafe class files. Thus, a client programmer has no guarantee that the instantiation of a generic class will not result in a type error (such as a `ClassCastException`) when the instantiation types are misapplied in the body of the class. We do not know of any satisfactory way to implement Generic Java via type

erasure that respects safe instantiation, and we believe that this very fact is a strong argument in favor of an approach such as NextGen that includes run-time type information. But it is also easy to violate safe instantiation in the design of NextGen if we are not careful.

4 NextGen and Safe Instantiation

The key issue with NextGen and safe instantiation concerns `new` expressions. NextGen allows `new` operations on type parameters, but if their use is not restricted, a matching constructor for a `new` expression may not exist. Additionally, unless every instantiation of a type parameter is concrete, a `new` operation may be attempted on an abstract class.

To address these problems, we can augment the constraints on type parameters to include `with` clauses, in which we specify the constructors required by each type parameter. The notion of `with` clauses for Generic Java were introduced in [1] and were formally modeled in [3]. By specifying a `with` clause on a type parameter, instantiations of the parameter that do not meet the constraints of the `with` clause are prevented. For example, suppose we wanted to define a generic class `Sequence<T>`, with a factory method `newSequence(int n)` that produced a new sequence containing `n` default elements of type `T`. Naturally, the method `newSequence` will have to construct new elements of type `T`. We can use a `with` clause to specify a legal constructor to call as follows:¹

```
class Sequence<T with T()> {
    public Sequence() {...}
    ...
    public static Sequence<T> newSequence(int n) {
        if (n == 0) return new Sequence<T>();
        else return Sequence<T>.newSequence(n - 1).cons(new T());
    }
}
```

The `with` clause on type parameter `T` requires that every instantiation of `T` include a zeroary constructor, preventing instantiations such as, *e.g.*, `Sequence<Integer>`, but allowing instantiations such as `Sequence<String>` or (because of the zeroary constructor defined in class `Sequence`), `Sequence<Sequence<String>>`. Notice that a separate generic class may instantiate these type parameters with type parameters of its own. For example, we might have:

```
class SequenceBox<R with R()> {
    Sequence<R> value = new Sequence<R>();
    ...
}
```

In this example, the instantiation of type parameter `T` in class `Sequence` with type parameter `R` in `SequenceBox` succeeds because the constructors specified in the `with` clause of `R` contain those

¹Unlike GJ, NextGen defines static members to be within scope of class-level type parameters, preventing us from having to declare a method-level type parameter in the definition of `newSequence`.

specified in the `with` clause of `T`. Because of separate class compilation, we have no information other than the `with` clause of `R` to ensure that instantiation `Sequence<R>` is safe.²

The other potential difficulty with `new` expressions in `NextGen` is the possibility of instantiating type parameters with abstract classes. Fortunately, `with` clauses allow us to escape this problem easily. Because the only purpose of `with` clauses in `NextGen` is to allow for `new` operations, we can require without loss of expressiveness that every instantiation of a type parameter that includes a `with` clause must be concrete, guaranteeing safe instantiation. Unfortunately, once we extend `NextGen` to `MixGen`, this simple solution is no longer sufficient.

5 MixGen and Safe Instantiation

In the `MixGen` extension to Generic Java, support is included for the occurrence of naked type parameters in any context where an ordinary type can occur, including the `extends` clause of a class definition. As explained in [3], the ramifications of this seemingly benign extension are far-reaching in terms of analysis, design, and compatible implementation on the JVM. Additionally, any extension of Java with support for mixins, either through generic types or through a separate facility such as in `Jam`³ must address several implications for safe instantiation. In particular, the following three threats arise:

1. The instantiation of a superclass might not contain a constructor matching the signature of a superconstructor call in a mixin constructor.
2. A mixin's parent may be instantiated with an abstract class, and an inherited abstract method that is not overridden may be invoked on the mixin.
3. A mixin may “accidentally” override a method inherited from an instantiation of its superclass, and the overridden method may contain an incompatible type signature.

In `MixGen`, the first two problems are handled by `with` clauses, just as the two analogous problems are handled in `NextGen`. However, one complication in the case of `MixGen` is that we want to allow instantiated superclasses of concrete mixins to be abstract. Because we still need to call superconstructors in the constructor definitions of mixins, we need to specify valid constructor signatures in the `with` clauses of parametric parent types, and therefore we cannot require the instantiation of every type parameter including a `with` clause to be concrete. But then the instantiation of a type parameter with an abstract class may match all of the constructors specified in the `with` clause and still cause an error when an inherited (non-overridden) abstract method is invoked on a mixin instantiation. To solve this problem, we must extend `with` clauses so that, in addition to specifying the required constructors of a type parameter, they also include the set of allowed abstract methods in an instantiation.⁴ For example, suppose we defined a generic mixin `ScrollPane<T>` intended to extend any subtype of an abstract class `Pane`. We

²The semantics of `with` clauses are also complicated by the fact that they are within scope of the type parameters of a class definition, requiring substitution of actual for formal parameters before checking for matching parameters. A formal semantics is presented in [3]

³`Jam` is an extension to Java 1.0 that includes a form of mixin. It was first presented in [7].

⁴Note that this extension of `with` clauses maintains upward compatibility with `NextGen`.

may want to allow a superclass to declare an abstract method `scroll` that is overridden in `ScrollPane`. We can allow that as follows:

```
class ScrollPane<T with {T(), abstract scroll()}> extends T {  
    public ScrollPane() { super(); ... }  
    public void scroll() {...}  
    ...  
}
```

Then every instantiation of the parent type of `ScrollPane` must include a zeroary constructor, and may not include any abstract methods other than `scroll()`. The instantiation of a type parameter that includes a `with` clause may not include abstract methods not specified in the `with` clause. If there is no `with` clause, the type parameter cannot occur as the parent type of a class, and so we can safely allow it to be instantiated with an abstract class.

The third problem for mixins is more subtle. Because the programmer defining a mixin does not know all of the instantiations of the mixin’s parent that will exist at runtime, there is no way that he can know exactly what methods will be inherited at runtime. Therefore, there is no way he can prevent the possibility of “accidentally overriding” an inherited method. This problem was first discussed in [11], where accidental overriding was presented as undesirable in the context of component-based software engineering; a programmer should never override the functionality of a class unless he intends to do so. But accidental overriding is also undesirable for a more fundamental reason: it is intrinsically incompatible with both type soundness and safe instantiation. To see why, one only needs to consider that if an accidentally overridden method is invoked on a mixin instantiation, the return type of the method may be incompatible with the return type in the static type of the receiver, breaking preservation of types under subject reduction and violating safe instantiation. Furthermore, in Generic Java, separate class compilation and the possibility of instantiating type parameters with yet other type parameters eliminates any hope of statically detecting all such overrides.

The original solution to accidental overriding, as proposed by Felleisen et. al. and demonstrated in `MixedJava`, is “hygienic mixins” [11]. Instead of allowing accidental overriding, hygienic mixins allow a mixin instantiation to contain multiple methods of the same name, with resolution of method application dependent on context. Hygienic mixins have been criticized by Ancona, Lagorio, and Zucca, who say that it leads to “ambiguity problems typical of multiple inheritance” [7].⁵ But in light of the fact that hygienic mixins are a necessary condition for safe instantiation, we believe they are a necessary feature of any production extension of Java with mixins. In `MixGen`, generic types are leveraged to simplify the semantics of hygienic mixins as compared with their incarnation in `MixedJava`. Additionally, it is explained in [3] how the `MixGen` formulation of hygienic mixins can be implemented compatibly on the JVM, making them an attractive candidate for a production extension of Java.

⁵Consequently, hygienic mixins have been omitted from the Jam language. If an accidental overriding occurs in which the signatures of the methods are incompatible, the program is rejected by the type checker. More seriously, if the type signatures of the methods happen to match, the inherited method will be overridden without signaling an error, causing an accidental altering of the semantics of the program. We believe that both of these problems are incompatible with the construction of reliable large-scale software systems.

6 Conclusion

As the above case studies have shown, adhering to the Safe Instantiation Principle when formulating a generic extension of Java is deceptively difficult. Nevertheless, we believe that adherence to this principle is necessary to ensure that generic types will help to improve the robustness and reliability of Java programs. We hope that the considerations presented here can be of use to other designers when composing new additions to the Java type system and to other languages.

References

- [1] O. Agesen, S. Freund and J. Mitchell. Adding Type Parameterization to the Java Language. In *OOPSLA'97*.
- [2] D. Ancona and E. Zucca. A Theory of Mixin Modules: Basic and Derived Operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.
- [3] E. Allen, J. Bannet, R. Cartwright. First-class Genericity for Java. Submitted to *ECOOP 2003*.
- [4] E. Allen, J. Bannet, R. Cartwright. Mixins in Generic Java are Sound. Technical Report, Computer Science Department, Rice University, December 2002.
- [5] E. Allen, R. Cartwright, B. Stoler. Efficient Implementation of Run-time Generic Types for Java. *IFIP WG2.1 Working Conference on Generic Programming*, July 2002.
- [6] E. Allen, R. Cartwright. The Case for Run-time Types in Generic Java. *Principles and Practice of Programming in Java*, June 2002.
- [7] D. Ancona, G. Lagorio, E. Zucca. *JAM-A Smooth Extension of Java with Mixins*. *ECOOP 00*, LNCS, Springer Verlag, 2000.
- [8] J. Bloch, N. Gafter. Personal communication.
- [9] R. Cartwright, G. Steele. Compatible Genericity with Run-time Types for the Java Programming Language. In *OOPSLA '98*, October 1998.
- [10] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *OOPSLA '98*, October 1998.
- [11] M. Flatt, S. Krishnamurthi, M. Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. *Formal Syntax and Semantics of Java*, volume 1523, June 1999.
- [12] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. In *POPL 1997*, January 1997, 146–159.
- [13] Sun Microsystems, Inc. JSR 14: Add Generic Types To The Java Programming Language. Available at <http://www.jcp.org/jsr/detail/14.jsp>.