First-Class Genericity for Object-Oriented Languages with Nominal Type Equivalence

Eric Allen
eallen@cs.rice.edu
Rice University
6100 S. Main St.
Houston TX 77005
March 11, 2003

Abstract
We propose to explore the feasibility of adding first-class genericity (including mixins) to a strongly-typed, object-oriented language with nominal (i.e., name-based) type equivalence. In such languages, mixins can be introduced simply as a generic class that extends one of its type parameters, e.g., a class $C<T>$ that extends $T$. Although this formulation of mixins has been used extensively and informally in C++ (via templates), its use has been severely hindered by the fact that C++ does not type check the parametric versions of class definitions, allowing for many insidious runtime errors. We propose developing a formal model of such a language in order to analyze its properties. Our goal is to show that our language design is (i) theoretically sound by formally proving that program execution preserves types (type soundness) and (ii) practical by describing in particular how to efficiently extend Java (the most popular language in this category of languages) to include first-class genericity without modifying the Java Virtual Machine or adversely affecting the compatibility of legacy source and binary code. For the purposes of benchmarking, we will develop a near-production-quality compiler (comparable in robustness to the JSR-14 compiler) that supports Java extended with first-class genericity. We will also develop a suite of benchmarks designed to stress the use of generic types. With these benchmarks, we can measure the performance of our compiler as compared with both Java and the JSR-14 prototype compiler (which supports a form of second-class genericity). Our expectation is that the addition of first-class genericity would not have a significant impact on performance.

1 Introduction
The use of static type systems with nominal type equivalence (as in Java, C#, and C++) has been predominant in popular object-oriented languages for over
a decade. Despite the objections of some language designers, nominal type systems have many advantages that are appealing to object-oriented programming, such as straightforward typing of recursive, and even mutually recursive, datatypes, simplicity, and decipherability of error messages. Nevertheless, the type system employed in most popular OO languages is far too simple to provide robust type checking. Although modern OO languages have thankfully moved away from the lack of type safety exhibited in C++, the simplicity of their type systems still allows for many run-time type errors. Their most significant failing is the lack of support for generic (parameterized) types. This omission restricts the range of abstractions that programs can express and the precision of static type annotation and checking. The incorporation of a comprehensive generic type system can simplify the structure of many programs, eliminate the need for nearly all explicit type casts, and enable programmers to catch far more bugs at compile time through much more precise static type checking.

1.1 Generic Types

With generic types, class definitions may be parameterized by type variables and program text may use generic types in many contexts instead of conventional types. For example, in languages with a Java-like syntax, a *generic class definition* has the form

```java
class Identifier < TypeParameters > extends ...
```

where each entry in the list of `TypeParameters` (separated by commas) is a type variable with an optional *type bound* of the form

```java
{ extends | ClassType }
```
or

```java
{ implements | InterfaceType }
```

A generic vector class in such a language might have the header

```java
class Vector<T>
```

A *generic type* consists of either a type variable or an application of a generic class name to type arguments that may also be generic. In a generic type application, a type parameter may be instantiated as any reference type that satisfies the specified bound. If the bound for a type parameter is omitted, the universal reference type (e.g., `Object`) is assumed.

1.2 Structural vs. Nominal Type Systems

Generic types have existed for many years and have been thoroughly analyzed in the context of languages with structural type equivalence (e.g., ML and Haskell).1 Unfortunately, their adoption into more popular object-oriented

---

1In the parlance of structural type systems, generic typing is usually referred to as “parametric polymorphism”.
languages with nominal type equivalence has been very slow. The difference between languages based on structural and nominal type equivalence is in how these languages determine when two types are equal. In a language with structural type equivalence, two types are equal iff either they are identical primitive types or they have the same structure, i.e., the types of all constituent elements are equal. In languages based on nominal type equivalence, two types are equal iff they have the same name. For the remainder of this paper, we will refer to these two categories of languages as structural languages and nominal languages.

One of the reasons for the slow adoption of generic types in nominal languages is that the analysis and design of type systems in the structural world does not translate well to the nominal world. Although one may naively expect that the analysis in one domain would be easily applicable to the other, the vastly different properties of these two forms of type systems, such as their formulation of mutually recursive datatypes, and their support for inheritance polymorphism (or lack thereof), make cross-application so difficult that it is often easier to simply carry out analysis in the two domains separately. Of course, it would be desirable to find significant correspondences between these two forms of type system and allow for greater cross-application of results, but the discovery of such correspondences has been elusive. Even object-oriented structural languages such as Objective CaML and object-oriented versions of Haskell have vastly different language properties than their nominal counterparts.

Fortunately, we are beginning to see the addition of stronger type systems in the nominal world. In 1999, Sun Microsystems publicly announced its interest in adding generic types to Java by publishing *Java Specification Request 14: Adding Generics to the Java Programming Language* [18]. This specified extension of the Java language is referred to as Generic Java. In that same year, Igarashi, Pierce, and Wadler developed Featherweight GJ, a small formal model of Java extended with generic types, and proved this generic type system to be sound. Two years later, Sun released an “early access” compiler for Generic Java (called JSR-14) based on a GJ compiler written by Martin Odersky [9]. Sun Microsystems has indicated that the next major release of the Java Platform (J2SDK 1.5) will include support for “second class” generic types based on GJ. Other extensions of Java such as NEXTGEN have extended Generic Java with support for generic “type-dependent” operations [10]. There is also a proposal to add generic types to C# [14]. However, all proposals for type-sound inclusion of generic types in these languages include only “second-class” generic types, in the sense that the use of generic types is restricted in many significant ways. Generic type systems based on type erasure, such as GJ, include the most significant restrictions. In these systems, generic types are present only during static type checking. After type checking, every generic type in the program is “erased”, i.e., it is replaced with a non-generic upper bound. For example, type annotations such as `List<T>` are erased to `List`, and (more significantly) naked type variables such as `T` are erased to their declared bound (typically `Object`). In a system based on type erasure, generic types cannot be used safely in any *type-dependent* operations, i.e., operations that explicitly refer to run-time types, such as casts, `instanceof` operations, and `new` expressions.
Other existing formulations of generic types in nominal languages are less restrictive but still prohibit the unrestricted use of generic types.

1.3 Second Class vs. First Class Generic Types

Although the addition of second-class generic types to mainstream object-oriented languages represents a major step forward in the evolution of these languages, the restrictions imposed often prevent programmers from applying generic typing to some important object-oriented coding patterns. For example, the `Cloneable` interface from the core Java API cannot be used in generic classes in GJ because the output of the `clone()` method cannot be cast the appropriate generic type [2]. Even the JSR-14 compiler, which is written in GJ, must breach the GJ type system because the compiler source code requires a cast to type `T` where `T` is a type parameter [6]. To “work around” this restriction, the JSR-14 compiler accommodates breaches in the type system by generating code for programs that use expressions of erased type in contexts requiring a specific generic type. Of course, sound static type checking is lost in the process. Furthermore, there are cases where the compiler generates incorrect code for untypable programs.²

![Figure 1: NextGen Representation of A Simple Generic Class Hierarchy](image)

Of the working formulations of generic types in nominal object-oriented languages, one of the most expressive is the NextGen formulation of Generic Java. NextGen is an upward-compatible implementation of Generic Java that...

²E.g., `new T[]` compiles to `new E[]` where `E` is the erasure (bounding interface) for `T`. 
removes the restriction GJ imposes on the use of type-dependent operations on naked type variables.

In NextGen, the only restriction on the use of generic types is the prohibition against using naked type variables as superclasses in generic class definitions [10]. The NextGen implementation strategy uses the Template Method design pattern to encode the relationships between generic types and their instantiations in a non-generic class hierarchy [12]. Each instantiation of a generic class is a separate class that is visible during program execution as shown in the example in Figure 1.

The lone restriction on the use of generic types in NextGen is significant because it prevents NextGen from supporting mixins—an important form of object-oriented abstraction that has been supported in various object systems for Lisp and some dynamically typed research languages, but not in a mainstream programming language other than a crude macro-based implementation in C++. We introduce the term “first-class genericity” to refer to a generic type system that allows the unrestricted use of generic types in all sensible contexts, including extends clauses of class definitions.

A simple example of a mixin class definition forbidden in NextGen (as well as all other sound type systems for nominal OO languages developed so far) is shown in Figure 2. This class definition is obviously illegal in NextGen because the class extends its own type parameter T. However, the intended meaning of this class definition is clear: each distinct instantiation of the class, such as TimeStamped<Hashtable>, should extend a distinct superclass. In essence, each instantiation defines a new version of the superclass that supports the functionality embodied in class TimeStamp.

class TimeStamped<T> extends T {
    public long time;

    TimeStamped() {
        super();
        time = new java.util.Date().getTime();
    }
}

Figure 2: A Simple Mixin Class

To simulate the behavior of this class in GJ, NextGen, or generic C#, we

---

3There is one other context where naked type variables cannot appear in NextGen, namely the list of interface types implemented by a class. A naked type variable does not have a sensible interpretation in this context because each superinterface specifies a lower bound on the member methods defined in the class. If a superinterface were a type variable, the corresponding lower bound would depend on the particular binding of the type variable, forcing the class to define a method for every possible method signature.

4Because the set of potential generic type instantiations during execution is unbounded, a NextGen implementation must use a modified class loader to construct new instantiation classes on demand.
would either have to copy this class definition once for each class we want to extend, or we would have to use composition (e.g., the Decorator Pattern [12]) to encode the subclassing relation. The former solution involves undesirable code replication. The latter solution forces programs to include a multitude of forwarding methods with less precise types. Moreover, the decorated classes must be designed with decoration in mind. Clearly, there is strong motivation to relax the language definition to allow for class definitions such as TimeStamped. However, eliminating this apparently small restriction on the syntax of Generic Java raises a surprising number of subtle and interesting language design problems. Thus, it is all the more surprising that no formal analysis of first-class genericity in nominal languages exists. Although mixins have been formally analyzed in isolation, previous formal analyses of mixins do not reveal many of their important properties when viewed as parametric classes in a first-class generic type system.

The remainder of this paper is organized as follows. First, we explain the history of mixins and elaborate on the motivation for them. Second, we define a research program to thoroughly explore the addition of first-class genericity to nominal languages, both at the theoretical and practical levels. Finally, we discuss related work and compare it with the goals of this research program.

Although we will focus on determining a practical implementation strategy for Java, we expect that a strategy for Java could be easily applied to other object-oriented languages such as C# that support incremental compilation, class loading, and a static type system with nominal type equivalence.

2 Mixins: History and Motivation

Nearly 20 years ago, the Lisp object-oriented community invented the term mixin [16] to describe a class with a parametric parent such as the TimeStamped class above. The name was inspired by the fact that such classes can be mixed together (via subclassing) in various ways, like nuts and cookie crumbs in ice cream. Denotationally, mixins have been modeled by Bracha and by Ancona and Zucca as functions mapping classes to new subclasses [7, 4]. For example, class TimeStamped can be viewed as a function that takes a class such as Hashtable and returns a new subclass of Hashtable that contains a timestamp. Mixins constitute a powerful abstraction mechanism with many important applications [3, 8, 11]. We briefly cite three of them:

- First, mixins can define uniform class extensions that add the same behavior to a variety of classes meeting a specified interface. The preceding TimeStamped class is an example of this form of mixin.

- Second, mixins provide a simple, disciplined alternative to multiple implementation inheritance. A class constructed as the result of multiple mixin applications contains implementation code from multiple independent sources. Mixins provide essentially all of the expressive power but none of the pathologies of multiple inheritance [11, 11].
Finally, mixins provide the critical machinery required to partition applications into logically independent “modules” or “components” in which all of a module’s contextual requirements are decoupled and captured in its visible interface. Existing package systems for mainstream languages like Java and C# are severely limited by the fact that packages contain embedded references to specific external class names, akin to hard coded filename paths in Unix scripts. These embedded references inhibit the reuse of a package in new contexts, and often prevent programmers from testing a package in isolation.

For example, it is difficult to test programs that make use of the `java.io` and `java.net` packages without actually accessing disk or establishing network connections. With first-class genericity, each module could be formulated as a class containing a collection of generic classes where top-level type parameters are used to designate references to external classes. All references to external classes such as `java.net.Socket` would instead be references to type parameters that could be instantiated during testing with mock objects. Because external class references may be subclassed in a package as well as used, first-class genericity is necessary to implement such a module system.

Formulating mixins as generic classes is particularly appealing because it provides precise parametric type signatures for mixins and enforces them through static type checking. In addition, this approach to defining mixins accommodates the precise typing of non-mixin classes provided by genericity. Hence, code containing mixins can be subjected to the same level of precise parametric type checking applied to other generic types, implying that the parametric types declared in mixins in Generic Java are respected during program execution. Previous formalizations of language-level mixins [16, 17, 8, 7, 11, 3] have not incorporated them into a generic typing discipline, and have sacrificed either robustness or expressiveness as a result.

### 2.1 Accidental Overriding and Hygiene

Because a mixin

```java
class M<T implements I> extends T
```

can have many different superclasses—depending on how client classes instantiate the mixin’s type parameters—the mixin is written with respect to a common interface `I` for the superclass. Unfortunately, this common interface is not sufficient to prevent unintended interference between a mixin instantiation `M<A>` and its superclass `A`. `M<A>` may *accidentally override* a method of `A` that is not a member of the interface `I`—breaking the superclass.

Mixins formulated as generic types make local static type checking of classes particularly difficult because the superclass of a mixin is not known when the mixin is compiled and locally type checked. When a generic or mixin class is instantiated somewhere in a program, each type argument can potentially flow
interface I {
    Object f();
}

class C<T with T()> extends T implements I {
    C() { ... }
    Object f() { ... }
    Integer m() { ... }
}

class D<T extends I with T()> extends T {
    D() { ... }
    Object f() { ... }
    String m() { ... }
}

class DFactory<T extends I with T()> {
    D<T> create() { return new D<T> }
}

class E<T with T()> extends T {
    Integer typeBreaker(C<Object> x) {
        return x.m();
    }
}
...

new E<Object>().typeBreaker(new DFactory<C<Object>>.create())

Figure 3: An example of a mixin with accidental overriding

anywhere in the program through type application. In each mixin application, the signature of the superclass argument must be checked against the signature of the mixin for consistency. The example in Figure 3 illustrates this problem.

In this example, each mixin definition is type correct in isolation. Moreover, the method invocation expression `new DFactory<C<Object>>.create()` at bottom is type correct given the headers of generic classes C, D, DFactory, and E. However, `new DFactory<C<Object>>.create()`, results in the mixin instantiation `D<C<Object>>`, in which the method `m()` in D accidentally overrides the method `m()` in C raising an error because the method return types are inconsistent. Notice that in a language allowing for separate class compilation, such as Java and C#, we can’t detect such accidental overrides during compilation. Class DFactory may be compiled separately from the method call `new DFactory<C<Object>>.create()`). Because a generic class can instantiate other other generic classes with its own type variables, only a whole program analysis could detect all accidental overrides. Even if we resorted to performing an “incremental” whole-program analysis as each class was loaded, the error wouldn’t be detected until load time. Furthermore, since the error messages could involve
code in classes for which the user does not maintain the source code, developers would be unable even to anticipate when such errors would occur.\(^5\)

In the literature on mixins, two different broad categories of semantics have been proposed for mixins: syntactic and hygienic. A syntactic semantics treats mixins as syntactic abbreviations (macros) and defines the meaning of each mixin instantiation as its syntactic expansion—much like the semantics of C++ templates. Such a semantics does not prevent accidental overriding. Moreover, it does not support local static type checking of classes. Since Java supports dynamic class loading, adding syntactic mixins to NextGen would make static type checking impossible; many type errors would not be discovered until load time.

In contrast, hygienic formulations of mixin semantics prevent accidental method overriding, via various mechanisms. A hygienic semantics does not just signal an error when accidental overriding occurs; it actually transforms the program to an equivalent one in which there are no accidental overrides (or evaluates it in such a way that accidental overrides disappear).

One issue that must be addressed by a first-class generic type system is whether mixins are to be treated hygienically or syntactically. We anticipate that, while a hygienic formulation of mixins would guarantee stronger safety properties, a hygienic implementation strategy would be less compatible with existing run-time systems such as the JVM and the .NET virtual machine, as well as with existing compiled binaries. Therefore, we plan to focus our theoretical analysis on a hygienic formulation of mixins, while focusing our implementation on a non-hygienic formulation that could be easily adopted by existing languages. Ideally, a non-hygienic implementation would be extendable in the long-run to a hygienic system.

3 Strategy of Research

In order to explore the addition of first-class genericity to nominal languages, we propose to carry out the following plan of research:

1. Formalize a “core” calculus of first-class generic types. The purpose of this calculus is to allow us to analyze formal properties of a first-class generic type system that would be intractable to prove for a production programming language. This calculus will play the same role for a nominal, object-oriented language with first-class genericity that the lambda calculus plays for functional languages such as Scheme and ML. Such a calculus should capture the relevant aspects of a full language design to a sufficient degree that key analyses over the calculus (such as the determination of type soundness) can be reasonably interpreted as providing evidence of analogous properties over a complete language. Ideally, all

\(^5\)In [7], we have argued that the Principle of “Safe Instantiation”, i.e., allowing the user to reason about what instantiations of a generic class are safe whether he maintains the source or not, is an essential property of any language with generic types.
programs expressed in the core language should be executable programs in a full language with first-class genericity (modulo trivial syntactic modifications). It would also be desirable if this calculus were a proper extension of an existing formalism of second-class generic types, such as Featherweight GJ, so that the formal differences between first-class and second-class generic types could be elucidated.

2. Prove type soundness over this core calculus. By establishing a type soundness proof, we will show that the notion of extending nominal languages with support for first-class genericity is not intrinsically flawed. Furthermore, a type soundness proof for the core calculus will provide corroboration that a full language design based on this calculus is also sound. Finally, because a first-class generic type system would allow the use of generic types in all contexts allowed by NextGen, a proof of type soundness for this calculus would corroborate the soundness of the NextGen type system, which, until now, has not been formally analyzed.

3. Extend the design of an existing object-oriented language to include support for first-class genericity. A natural choice of language to extend is Java, since it is very popular, the underlying type system is sound, and there are already language extensions of Java that support second-class generic types (such as NextGen).

4. Refactor the GJ compiler into a near-production quality software system conducive for the experimental research of new language extensions. This component of our research program is by far the most substantial in terms of the sheer programming effort involved.

5. Extend the refactored codebase to compile valid NextGen programs in addition to GJ programs. This compiler will be the first compiler for the NextGen programming language.

6. Release the NextGen compiler to the research community as a prototype for experimental use. The release of a compiler for external use is important because it will help to ensure that the compiler is reliable in multiple environments over a variety of coding styles.

7. Develop a benchmark suite that makes heavy use of generic types, and measure the performance of this suite in comparison to both a simple type system and a generic type system based on type erasure. In order to ensure objective measurement, this benchmark suite should use generic types in a manner compatible with all tested formulations of generic types. We anticipate that our benchmark results will indicate that run-time support for generic types incurs no significant overhead when compared with simple type systems and generic type systems based on type erasure.

8. Develop a design strategy for compatibly extending NextGen with support for hygienic first-class genericity.
9. Further extend the NEXTGEN compiler to support (non-hygienic) first-class genericity. Although the safety properties of non-hygienic genericity are not as strong as those of a hygienic type system, a non-hygienic system provides greater interoperability with existing run-time systems such as the JVM, and therefore is more likely to be adopted in existing languages. We believe that the adoption of non-hygienic first-class generic types will increase demand in the long run for a hygienic version of first-class generic types.

As mentioned, the single largest task in this plan is step 4. Under a special licensing agreement with Sun Microsystems, our research group has acquired the source code for the GJ compiler. This compiler is complete for the GJ extension to generic Java and consists of approximately 35,000 lines of code. However, the state of the source code is not conducive to extension. There are many design problems, misleading naming conventions, virtually no documentation, and absolutely no test harness at all to aid in refactoring. Furthermore, because the code is written in GJ and not Java, many development tools for Java such as code analysis and refactoring tools cannot be used with it. In order to harness this codebase and to convert it into a production system, we will take the following steps:

1. Retrofit a comprehensive unit test suite over the code.

2. Using our unit test suite as a safety harness, efficiently refactor the package structure, class structure, method structure, and naming conventions of the compiler.

3. Write a comprehensive developer guide describing the new architecture of the code and its key invariants.

4. Develop the necessary infrastructure to place the code under version control, with accompanying build scripts and tools to enforce project invariants such as the running of all unit tests on every commit to our cvs repository.

In our experience, the development of reliable software systems is crucially dependent on the development of a unit test suite over the code. The importance of unit testing to production software development is well-documented in modern literature on software engineering [13]. Unit tests serve several critical functions in a production system.

First, they serve as a form of “executable documentation”, providing programmers with concrete examples of how each unit of functionality in the program should behave. These examples can and should be checked against the state of system on a regular basis. In all other production software projects in our research group, we require, as part of our cvs commit process, that a new version of a project passes all tests before it can be committed to the repository.

Second, unit tests help to vastly improve the efficiency with which a software project can be debugged. Because unit tests enforce many key invariants of
a project, they help the programmer to quickly eliminate many possibilities of error when tracking down a bug. Furthermore, they help to ensure that an eliminated bug never resurfaces. As an integral part of any bug fix, we require that an example manifestation of the bug be added as a unit test in the repository, preventing new code from being checked in if the bug ever comes back.

Finally, unit tests provide the critical framework necessary to effectively refactor a codebase. Because the unit tests check key invariants, a programmer in the midst of refactoring the code can run the unit tests regularly to ensure that these invariants aren’t broken by new refactoring. In the case, of the GJ compiler, our ability to refactor the codebase is critical because of the poor state of the source code.

In order for unit tests to provide us these advantages, they must cover the source code extensively. In the DrJava project, a 50,000 line open-source project by our research group to develop a Java IDE, the percentage of source code devoted to unit tests is approximately one-third.

Unit tests are best developed concurrently with the code they test. Because concurrent development of unit tests helps to catch bugs in the source code more quickly, it is far more efficient than writing unit tests after development. Additionally, the discipline of adding tests over code as one writes it has positive ramifications on how one designs the source code, generally making it simpler and more extensible. Of course, in the case of the GJ compiler, which has already been written, concurrent development of unit tests is no longer possible. So, our goal is to retrofit this compiler with unit tests that cover the key invariants of the source code. In doing so, we will have to discover these invariants through interactive exploration, as they are undocumented. Additionally, because the codebase was not written concurrently with unit tests, many aspects of it are (not surprisingly) very difficult to test. We intend to develop the necessary framework for quickly writing more unit tests, so that we will be able to efficiently extend it in new directions in other research projects.

Despite the difficulties with retrofitting a large codebase with unit tests, we believe that doing so is critical to the success of both this application of the GJ compiler, and future applications. There is simply no other way for us to develop the expertise and control of the source code necessary to maintain it.

Because the existence of development tools can greatly augment programming productivity, we will utilize all tools available for assisting in the refactoring of generic Java code, including our own development tool, DrJava. By utilizing DrJava, we will become an onsite customer of that tool, and therefore help to further its evolution.

4 Related Work

To our knowledge, the first reference to mixins in Java occurs in a paper by Agesen, Freund, and Mitchell [1] describing an extension to Java to support genericity via syntactic expansion in the class loader. While the paper mentions
that this approach can support mixin constructions, no type system supporting mixins is given and the critical language design issues involved in such a language extension—such as mixin hygiene, the status of abstract methods in mixins, and the definition of mixin class constructors—are not discussed. Since their model for supporting generics is syntactic expansion (as in C++ templates), they presumably were proposing non-hygienic mixins. In this case, we do not believe that the static type checking of mixins is compatible with the separate class compilation provided by modern compilers for Java and C# because type correctness requires a whole-program analysis to confirm that overridden methods in mixin instantiations have the proper return types.

The only other practical proposals for adding mixins to Java, namely Jam [3] and Jiazzi [15], do not accommodate generic types. JAM is an extension of Java 1.0 developed by Ancona and Zucca that supports mixin definitions as a new form of top-level definition supplementing classes and interfaces. Each mixin instantiation is explicitly defined by a special form of class definition that includes the constructors for the new class. Jam is based on the theoretical framework for “mixin modules” as presented in [5]. That theoretical framework is not hygienic because it provides no mechanism for altering the “view” of a mixin based on context. Multiple operators are provided for mixin composition with various rules for shadowing colliding methods, but the set of visible methods resulting from each of these rules is not dependent on the context of the reference to a mixin instantiation. Also, it does not account for the many complexities that arise when mixins are formulated as generic types.

Since the JAM type system lacks the expressiveness of generics, it must restrict the use of this within the body of a mixin. In particular, this cannot be passed as argument to a method, which is a severe restriction on the design of object-oriented programs. JAM mixins are not hygienic, but programs that perform accidental method overriding with incompatible type signatures are rejected by the type checker. JAM is implemented by a preprocessor that maps Jam to conventional Java. Since Jam does not support genericity, types cannot flow across a whole program. As a result, Jam can locally type check programs within mixins.

Jiazzi [15] is a component system for Java developed by McDirmid, Flatt, and Hsieh that supports component level mixins. Jiazzi is implemented by a linker that processes class files to produce new class files. Using Jiazzi, a programmer can partition a program into components with unresolved references (wires) and define compositions that wire components together. Since Jiazzi is a component system that is not part of the program source language, it is not directly comparable to an object-oriented language with mixins. But mixins can be created by the the component linking process because the superclass of a class may be an unresolved reference within a component. Since mixins can only be instantiated in the meta-language used to wire components together, Jiazzi does not have to address the same language design issues as those of a

---

6 To appease the Java compiler, which will not compile a class with an undefined superclass, the programmer must define a stub class for each such unresolved reference.
first-class generic type system. Additionally, the Jiazzi type system does not include generic types and thus the complications with mixins formulated as generic types do not arise. Nevertheless, Jiazzi mixins must be hygienic because components only expose selected public methods. In the absence of hygiene, component composition would break component encapsulation. Jiazzi enforces mixin hygiene and the static typing of mixins by performing a whole-program analysis on a program composition.

Hygienic mixins were originally developed by Flatt, Krishnamurthi, and Felleisen in a toy language loosely based on Java called MixedJava [11]. MixedJava does not include generic types; mixins are formulated as a separate language construct. Because all mixin instantiations in that system could be determined statically, their employment of a hygienic semantics was not critical to type soundness, but its inclusion was motivated by software engineering considerations. In MixedJava, all classes are constructed by mixins. To specify both the static types of expressions and the dynamic types of program values, MixedJava uses special type expressions called views that consist of sequences of mixin names. Every value in MixedJava is a pair consisting of an explicit type and an object reference. An object \( o \) has type \( T \) iff the type is a segment of the chain of mixins used to form the class of \( o \). Programs can cast a value to a compatible type, creating a new value with the explicit type specified in the cast.

The hygienic mixin semantics described in [11] is based on the intuition that mixins are functions that map classes to subclasses. Given a mixin definition

```java
class M<T implements I> extends T { ... }
```

and a class \( A \) implementing \( I \), the only members of \( A \) that can be overridden in the mixin instantiation \( M<A> \) are those declared in \( I \). All other methods of \( A \) are systematically renamed to avoid any possible accidental collision. The methods in the type \( I \) bounding the mixins’s superclass are not renamed. Since all of the references to the methods of \( M \) are renamed also, the methods introduced in \( M \) effectively shadow any methods in a superclass with matching signatures. In order to allow a client class to refer to shadowed methods in a mixin, each value is associated with a view that can be modified based on the needs of the client class. It is important to recognize that, unlike casts in Java or C#, views are associated not with expressions but with values; they are carried with values when, e.g., they are passed across method boundaries. As a result, there is a fairly high penalty for hygiene in MixedJava: the size of every reference value is doubled.

The type system of MixedJava is strictly less expressive than a type system incorporating first-class genericity. In particular, MixedJava does not provide a type for the superclass of a mixin. For example, in the body of a mixin

```java
M<T implements I> extends T { ... }
```

7We use a “NextGen-style” syntax to represent this definition, but it would obviously be represented differently in the language defined in [11].
MixedJava cannot name the type $T$ or $M<T>$. As a result, MixedJava does not support the precise typing of polymorphic recursion or other programming patterns that introduce cycles in the mixin type application graph.

MixedJava is an interesting design study but it does not provide a practical basis for extending existing production run-time systems such as C# or the Java Programming Language. The fact that values are pairs containing a view and an object reference means that every value occupies two machine addresses instead of one, nearly doubling the memory footprint of many applications and significantly slowing computation. In addition, it is not clear how to map MixedJava onto existing run-time systems in a way that preserves compatibility with legacy binary code.

5 Conclusion

Adding first-class genericity to a nominal object-oriented language would produce a surprisingly powerful language that supports precisely typed mixins as well as conventional generic classes. But there are many analysis and design issues that must worked out before such an extension can be performed safely. We have outlined a strategy for carrying out just such an analysis and design, as well as a compatible implementation on an existing run-time system. We believe that the results of this research would provide a compelling argument for adding first class genericity to the next generation of object-oriented languages.

References


[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


