

First Class Genericity for Java

Eric E. Allen Jonathan Bannet Robert Cartwright
eallen@cs.rice.edu jbannet@cs.rice.edu cork@cs.rice.edu

Rice University
6100 S. Main St.
Houston TX 77005

November 24, 2002

Abstract

This paper describes how to add first class genericity including mixins to the Java language without modifying the Java Virtual Machine or adversely affecting the compatibility of legacy source and binary code. In Java augmented by generic (parameterized) types, a mixin is simply a generic class $C<T>$ that extends the type parameter T . Hence, mixins are the byproduct of fully supporting first class status for generic types. We show that the our language design is *(i)* theoretically sound by proving that program execution preserves types (type soundness) and *(ii)* practical by describing how to implement it efficiently on top of the existing Java Virtual Machine. To our knowledge, this paper presents the first type soundness result for a precisely typed, object-oriented programming language with mixins.

1 Introduction

Since its debut in 1995, the Java platform has won wide acceptance in both industry and academia as a superior vehicle for developing software applications because it supports object-oriented design, comprehensive static type checking, safe program execution, and an unprecedented degree of portability. Despite its spectacular success, the Java platform is still in the 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 (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 would simplify the structure of many Java 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.

Fortunately, the designers of Java did not attempt to design the ultimate object-oriented language but provided hooks for “growing the language” [22]. Java is the first mainstream language that includes a “strongly typed” binary representation (class files) and a corresponding virtual machine for executing binaries as part of the language standard. This binary representation is extensible by new attributes (and generalizations of existing attributes) which can be interpreted by new class file loaders. As a result, the behavior of the Java platform can be *extended without modifying the underlying virtual machine*, thereby preserving the behavior of existing Java binaries. In addition, the Java source language can be easily extended

by building new source-to-bytecode compilers (replacing `javac`) that implement a superset of the existing language. In its short history, Java has already experienced a major growth spurt—through the addition of inner classes—and is on the verge of growing again—through the addition of “second class” generic types.

1.1 Adding Generic Types to Java

In early 1998, programming language researchers Martin Odersky, Philip Wadler, and Robert Cartwright collaborated with colleagues Gilad Bracha, David Stoutamire, and Guy Steele at Sun Microsystems in exploring possible designs for a genericity facility for Java. The researchers agreed on a common syntax for Generic Java, but produced two different language extensions: a restricted version based on type erasure called GJ by Bracha, Odersky, Stoutamire, and Wadler [11] and a comprehensive version carrying run-time type information called NEXTGEN by Cartwright and Steele [9, 4]. NEXTGEN and GJ were designed in concert so that NEXTGEN is backward compatible with GJ. GJ does not support the operations of Generic Java that require run-time parametric type information.

1.1.1 Generic Classes

In Generic Java, class definitions may be parameterized by type variables and program text may use generic types in place of conventional types (with a few exceptions). A *generic class definition* has the form

```
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

```
{ extends | ClassType }
```

or

```
{ implements | InterfaceType }
```

For example, a vector class might have the header

```
class Vector<T>
```

Interface definitions are similarly generalized. A formal syntax for Generic Java is given in the online GJ Specification [12].

A *generic type* consists of either a type variable or an application of a generic class/interface 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 `Object` is assumed. A generic type can appear anywhere that a class or interface name can appear in ordinary Java—except as the superclass or superinterface of a class or interface definition. This restriction means that a “naked” type variable cannot be used as a superclass.

The scope of the type variables introduced in the header of a class or interface definition is the body of the definition, including the bounding types appearing in the header. For example, a generic ordered list class might have the type signature

```
class List<A, B implements Comparator<A> >
```

where `A` is the element type of the list and `B` is a singleton¹ ordering class for `A`. Static members of a generic class create holes in the scope of all the enclosing type abstractions. A nested static class can be generic but all of the type variables of the class must be explicitly introduced in the definition of the class.

¹A *singleton* class is a class with only one instance. Classes with no fields can generally be implemented as singletons.

1.1.2 Polymorphic Methods

Method definitions can also be parameterized by type. In Generic Java, the syntax for the header of a method definition is generalized to:

```
{Modifiers} {< TypeParameters >} Type Identifier ( { ArgumentList } )
```

where `Type` can be `void` as well as a conventional type. The scope of the type variables introduced in the type parameter list (`TypeParameters` above) is the header and body of the method. When a polymorphic method is invoked, the type instantiation information can be inferred in most cases. Generic Java can use the types of the argument values in the invocation to determine the values of the type arguments.² Generic Java also provides a syntax for explicitly binding the type arguments for a polymorphic method invocation, but none of the current compilers (GJ, JSR-14, and NEXTGEN) support this syntax yet.

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* [23]. Two years later, Sun released an “early access” compiler for Generic Java (called JSR-14) based on a GJ compiler written by Martin Odersky [11]. 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.

1.2 GJ vs. NEXTGEN

GJ restricts the use of Generic Java to programs that can be implemented using type erasure. As a result, GJ prohibits:

- parametric casts,
- parametric `instanceof` tests³,
- parametric catch operations, and
- new operations of “naked” parametric type such as `new T()` and `new T[]`.

In essence, GJ supports the subset of Generic Java in which generic types are not first class. NEXTGEN supports the full Generic Java language.

1.3 Second Class vs. First Class Generic Types

Although the addition of second class generic types to Java represents a major step forward in the evolution of Java, the restrictions imposed by type erasure 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 because the output of the `clone()` method cannot be cast the appropriate generic type [5]. 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 [7]. 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.⁴

²The GJ compiler implements more general inference rules that treat the value `null` as a special case.

³GJ supports parametric casts and `instanceof` tests provided the parametric information in the operation is implied by context. In such cases, the parametric cast or `instanceof` test can be implemented by their type erasures.

⁴E.g., `new T[]` compiles to `new E[]` where `E` is the erasure (bounding interface) for `T`.

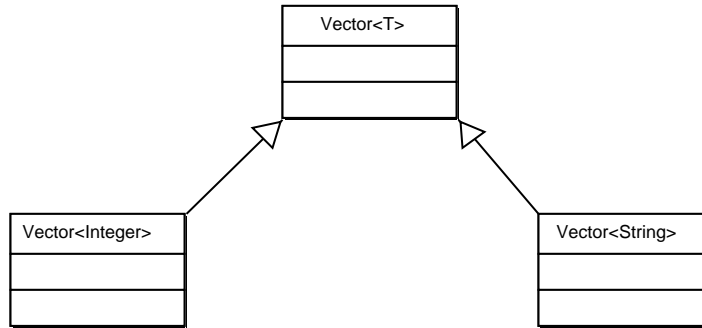


Figure 1: NextGen Representation of A Simple Generic Class

In contrast, the NEXTGEN formulation of Generic Java treats generic types as essentially first class citizens [9]: the only restriction on the use of generic types is the prohibition against using naked type variables as superclasses in generic class definitions.⁵ In NEXTGEN, each instantiation of a generic class is a separate class that is visible during program execution as shown in the example in Figure 1. Recent benchmark results for a prototype compiler for NEXTGEN demonstrate that the “first class” treatment of generic types does not have any significant performance overhead when compared to either Java or JSR-14 [4].

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++.

A simple example of a forbidden mixin class definition in NEXTGEN 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

⁵There 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*.

To simulate the behavior of this class in GJ or NEXTGEN, we 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 [14]) 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 interesting language design issues.

1.4 Mixins

Nearly 20 years ago, the Lisp object-oriented community invented the term *mixin* [20] 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 [10, 2]. 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 [6, 8, 15]. 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 [15, 16].
- Finally, mixins provide the critical machinery required to partition Java applications into logically independent “modules” or “components” in which all of a module’s contextual requirements are decoupled and captured in its visible interface. The existing Java package system is 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 as Generic Java code in GJ and NEXTGEN, implying that the parametric types declared in mixins in Generic Java are respected

during program execution. Previous formalizations of language-level mixins [20, 21, 8, 10, 15, 6] have not incorporated a precise parametric typing discipline.

Although the NEXTGEN language defined by Cartwright and Steele explicitly excludes mixins,⁶ its implementation architecture can be gracefully extended to support them. In fact, the NEXTGEN architecture can be extended to support a *hygienic semantics* [16] for mixins without adding significant overhead to the execution of non-mixin or mixin classes.

The remainder of this paper is organized as follows. First, we define the MIXGEN language, an extension of NEXTGEN that supports mixins. Second, we show how to map this language onto the JVM by extending the NEXTGEN implementation architecture. Third to demonstrate the soundness of the MixGen type system, we present an operational semantics, type inference system, and type soundness theorem for CORE MIXGEN, a subset of MIXGEN that encapsulates the important aspects of MIXGEN. Finally, we discuss related work and directions for future research.

2 The Design of MIXGEN

Before we describe the syntax of semantics of MIXGEN, we need to discuss a minor extension to Generic Java that addresses a weakness in the design of Generic Java.

2.1 Preliminaries

In Generic Java [9], the definition of a generic class implicitly imposes some restrictions on the type arguments that may be used to instantiate the class. In particular, the use of a `new` operation on a naked type variable in a generic class forces the corresponding type argument to be a concrete class and to provide a constructor with a matching signature. These restrictions can and should be made explicit.

The forthcoming stable compiler for NextGen will support an optional `with` clause in the declaration of a type variable that specifies a list of constructor signatures. The `with` clause restricts the binding of the type variable to concrete classes with the specified constructors. For example, the generic class header

```
class Vector<T with T()> { ... }
```

requires that any argument support a zero-ary constructor. In the new version of NextGen, a `new` operation can only be performed on a naked type variable if the declaration of the type variable includes a `with` clause with a matching constructor.

2.2 MixGen Extensions

MIXGEN is a proper extension of the new version of NEXTGEN; all existing NEXTGEN programs are valid MIXGEN programs, with the same semantics. MIXGEN extends NEXTGEN as follows:

1. The superclass specified for a generic class may be a type variable. When the superclass of a generic class is a type variable, then the declaration of the type variable must include a `with` clause.
2. A `with` clause may include `abstract` and `final` declarations for some of the methods in the bounding type `I` for the type variable. A type `T` may be bound to such a type variable only if it is a subtype of `I` and its `abstract/final` methods are a subset of those declared as `abstract/final` in the `with` clause.

⁶NEXTGEN was designed to address the requirements codified in JSR-14 which does not mention mixins.

```

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

```

Figure 3: An example of a mixin with accidental overriding

These two additions are motivated solely by the inclusion of mixins in the language. The first extension allows the superclass of a generic class to be a type variable—defining a mixin—provided that the bound for the type variable includes a `with` clause. The `with` clause is essential because mixin constructors must invoke a superclass constructor before initializing the fields of the mixin. In the absence of a `with` clause, the signatures of the superclass constructors would be unknown.

The second extension allows methods in the superclass of a mixin to be abstract or final, provided that they are explicitly declared as such in the `with` clause for the type variable designated as the superclass. This restriction enables a MIXGEN compiler to determine precisely which methods in a mixin instantiation are abstract and to prevent the mixin from attempting to override inherited final methods.

For notational convenience, MIXGEN could support an extended form of interface, called a *type bound*, consisting of an interface and a `with` clause. Type bounds would only be allowed as the bounds for type variables in the headers of generic classes. We will not discuss this feature further since it is simple syntactic sugar.

2.3 Accidental Overriding and Hygiene

Since a mixin

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

can have many different superclasses—depending on how client classes instantiate the mixins 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 $M\langle A \rangle$ and its superclass A . $M\langle A \rangle$ may *accidentally override* a method of A that is not a member of the interface I —breaking the superclass. Figure 3 presents an example of accidental overriding that causes a type error. In the mixin instantiation $D\langle C\langle Object \rangle \rangle$, the method $m()$ in D accidentally overrides the method $m()$ in C raising an error because the method return types are inconsistent.

In the literature on mixins, two different semantics have been proposed for mixins: syntactic and hygienic. The 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. This semantics does not prevent accidental overriding. Moreover, it does not support local static type checking of classes.

Mixins make local static type checking of classes 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 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 the example, each mixin definition is type correct in isolation. Moreover, the method invocation expression at bottom is type correct given the headers of generic classes C , D , and E , the class member signatures of E . But the syntactic expansion of $D\langle C\langle Object \rangle \rangle$ contains a type error because method $m()$ in D *accidentally overrides* method $m()$ in C and the method return types are inconsistent. 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.

The hygienic semantics for mixins was developed by Flatt, Krishnamurthi, and Felleisen to eliminate accidental method overriding [15]. The semantics is based on the intuition that mixins are functions that map classes to subclasses. Given a mixin definition

```
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 $M\langle A \rangle$ are those declared in I . All other methods of A are unaffected.

One way to explain this semantics is through method renaming. The methods *introduced* by a mixin

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

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.⁷

In the semantic model for MIXGEN, hygiene is enforced by using a more sophisticated form of dynamic dispatch (method lookup) than conventional Java. If MIXGEN method invocations were resolved simply by finding to the method implementation closest to the run-time type of the receiver, as they are in Java, then mixins would not be hygienic and would break the Java static type system. In Figure 3, the invocation on the `typeBreaker` method, which has static type `Integer`, would reduce to a run-time value of type `String`, which is inconsistent with the static return type of the invocation. The problem is that mixin instantiation

⁷This explanation is incomplete because it does not specify how to match method invocations with the the appropriate new method name. On method dispatches with respect to a class type, it is easy to determine which renamed method is meant by searching the class hierarchy as described below. But method dispatches with respect to interface types require more sophisticated treatment. This issue is discussed in Section 3.3.

`D<C<Object>>` *accidentally overrides* method `m`. In the presence of accidental overriding, we do not see any way to detect such inconsistencies using local type checking as performed by Java compilers.

In order to address this problem, we have defined method invocation in MIXGEN to be a generalization of the method invocation mechanism in Java. Notice that when a method invocation is statically checked in Java, the method signature checked is (necessarily) resolved according to the static type of the receiver. This resolution mechanism works in the context of overriding because any overriding method must have the same signature as the method in the static type. Semantically, the process of method invocation in Java can be viewed as starting at the static type of the receiver, walking down the class hierarchy toward the run-time type, and finding the matching method of identical signature that is closest to the run-time type.

In Java, this semantic lookup procedure can also be described as finding the matching method definition closest to the run-time type because Java prevents (via type checking at compile time and at load time) any overriding method from having an incompatible signature at run time. In MIXGEN, the class hierarchy is unresolved until run time, so a more sophisticated semantics must be used. When searching for a method in MIXGEN, the method call must start at the static type of the receiver and search down the hierarchy toward the run-time type. If a mixin class on this path does not include the specified method in its superclass interface, then the search stops *because the method is hidden in the mixin class and all of its subclasses*. The resolved method is the last matching method encountered before reaching the mixin class that hides the method. Of course, if no hiding mixin class is encountered in this traversal, the resolved method is the same as it is in conventional Java.

It is not obvious that this generalization of method resolution is compatible with the semantics of method resolution embedded in the Java Virtual Machine. In the next section, we will explain how mixin method resolution can be implemented efficiently without modifying the JVM.

This approach to method resolution provides the programmer with significant control over which method in a class constructed using multiple mixin instantiations is invoked. By excluding methods from the bounding interface of a mixin parent type, the programmer can place a barrier that prevents those methods from being overridden by subclasses. If a programmer wants to explicitly designate the static type from which the search starts, he may do so by casting the receiver expression. By upcasting the static type of the receiver, a program can invoke methods that are otherwise hidden. This mechanism is analogous the use of upcasting to access shadowed fields of an object in conventional Java.

3 Design

Before we describe how to extend the NEXTGEN implementation architecture to support MIXGEN, we need to describe that architecture in more detail. NEXTGEN supports type-dependent operations on generic types, such as casts, `new` expressions, and `instanceof` tests using a representation that is as *homogeneous* [17] as possible. Only the type dependent operations in a generic class are pushed into the instantiation classes; all of the other code for a generic class is shared among these instantiations in an abstract base class with erased method signatures.

The NEXTGEN compiler translates a generic class to two class files: a class file for an erased based class and a *template class* file for creating instantiation classes that extend the base class [4].⁸ Template class files have exactly the same form as conventional class files except for the fact that the constant pools contain unresolved references to generic type parameters.

⁸The compiler also generates a *template interface* file for creating an instantiation interface corresponding to each instantiation class. These interfaces are used to encode subclassing relationships among generic class instantiations. Since they are not used in the implementation of mixins, they are not discussed in this paper. See [9, 4] for more details.

During program execution, NEXTGEN relies on a special class loader to recognize references to generic class instantiations and dynamically construct each such class the first time that it is referenced. Generic types are represented by mangled class names that encode the generic class name and all of the type arguments. When a class with a mangled name is first referenced, the class loader reads the template class file (cached in memory if it has been accessed previously) for the specified generic class and patches the unresolved references in the constant pool with the matching mangled names of the type arguments.

3.1 Overview of How to Implement MIXGEN

The NEXTGEN implementation architecture does not directly support mixins because the instantiations of a particular mixin all have different superclasses. They cannot be subclasses of a common base class.

The simplest way to extend NEXTGEN to support mixins is to use a fully *heterogeneous* representation of mixin instantiation classes. In this such a representation, each instantiation of a mixin is a separate class containing all of the code for the methods immediately defined in the mixin. In principle, the common code across these instantiations could be factored out into a common code object that is embedded in each instantiation. But this approach is more complex than a purely heterogeneous implementation and presumably less efficient because of the overhead incurred in forwarding method calls to the common code object.

To add mixins to NEXTGEN, we must extend the NEXTGEN compiler to translate mixins to template class files and modify the class loader to enforce mixin hygiene. If hygiene were not an issue, the existing NEXTGEN class loader would suffice for this purpose. But this naive approach does not prevent accidental overriding. The class loader must systematically rename class methods to enforce hygiene. We describe how renaming can be done in Section 3.3 below.

3.2 Compiling MIXGEN

Recall that the declaration of bounding type I in a mixin declaration

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

must include a supplementary `with` clause specifying constructor signatures and method constraints specifying which superclass methods may be abstract and which may be final. The compiler uses this information to determine whether the superclass argument used in a mixin instantiation is compatible with the methods introduced by the mixin. The superclass must support the specified constructors and not include any final or abstract methods in the bounding type other than those declared in the `with` clause. Similarly, a mixin is not well-formed unless it is compatible with any superclass that has the specified abstract and final methods.

The information in the `with` clause for a type parameter is only needed during program compilation. Programs that violate the specified constraints are rejected by the compiler as ill-formed. Since the compiler must be able to compile the client classes of a mixin separately from the mixin, the information in the `with` clause must be stored as an “optional” attribute in the template class file for the mixin. The class loader ignores such optional attributes at load time.

3.3 Enforcing Hygienic Method Invocation

The semantics for MIXGEN described above formulates method invocation as a downward search from the static type of the receiver, allowing method overriding in a mixin instantiation only when the method is included in the bounding type of the superclass. This search is more elaborate than the method invocation

mechanism employed in the JVM, which effectively searches up the class hierarchy from the class type of the receiver.⁹ However, our “downward search” semantics can be implemented on the existing JVM provided that we systematically rename method names in the MIXGEN class loader to make accidental overriding impossible.

The renaming of methods in mixin classes is a subtle issue. The class loader obviously must rename the *new* methods introduced in a mixin to avoid accidental overriding. In fact, all *new* methods must be renamed to avoid accidental overriding because a subclass of a mixin instantiation can reintroduce a method that was hidden by the mixin.¹⁰ Similarly, any overriding definition of a renamed method in a subclass must be renamed and each reference to a renamed method must be updated to reflect the new name. For each method invocation based on a class type, the class loader can determine the class on the ancestor hierarchy where that method was introduced and perform the appropriate renaming. But this transformation does not work for method invocations based on interface types. Such a method invocation may resolve to several different renamed methods depending on the class of the receiver!

We can solve the problem of interface-based method dispatch by adding forwarding method for each method introduced in a class that maps `invokeinterface` calls on the external name (appearing in the interface) to the generated internal name.

The simplest renaming scheme that satisfies all of above constraints is to prefix the name of every new method introduced in a any class `C` (mixin instantiation or otherwise) by a qualifier consisting of name of the class `C` and a `$` sign.¹¹ The class loader processes every method descriptor for a class based dispatch (`invokevirtual`) in the constant pool mapping the method name to its prefixed form. Method descriptors for interface based dispatches are left unchanged. If the class is a mixin instantiation, then a forwarding method must be generated for each *new* method (introduced in the mixin) mapping its external name to its prefixed name.

In this manner, method invocation *behaves* as if it were performed by a downward search from the static type of the receiver. But it is implemented using the standard JVM protocol.

3.4 Optimized Renaming

The preceding renaming scheme is conceptually simple, but it can significantly increase the size of constant pools if packages have long names. If we add a table to the constant pool relating internal method names with the corresponding external names (and other information such as where the method is introduced), then we can generate short names using a “gensym” mechanism, reducing the size of constant pools. We do not know if the space saved by such an approach is worth the extra complexity.

4 CORE MIXGEN

The semantics for MIXGEN method lookup is by far the most intricate aspect of the MIXGEN design. For this reason, we have constructed CORE MIXGEN, a small formal model of the language in order to study the properties of method lookup and to prove the soundness of the associated type system. In this regard, we were strongly influenced by the design of Featherweight GJ [18]. There are many similarities between Featherweight GJ and CORE MIXGEN. Most notably, both languages are purely functional, single-threaded, and devoid of `null` references.

⁹The method table in each class object reduces this search to a table lookup.

¹⁰Even the methods in classes that are not subclasses of a mixin instantiation must be renamed to avoid colliding with method dispatches based on interface types.

¹¹If `C` is a generic instantiation, the name of `C` in the prefix must be mangled.

On the other hand, there are several important differences. CORE MIXGEN requires all user-defined classes to be constructed by mixins and supports (single-inheritance) interfaces while Featherweight GJ focuses on conventional classes with no interfaces. By restricting the language to mixins, we focus our attention on the technical issues that we believe warrant the most careful treatment. Because interfaces (enhanced with constructor signatures) play a crucial role in method application resolution in MIXGEN, it was important to include at least a simplified version of them in CORE MIXGEN. In CORE MIXGEN, interface extension is limited to single inheritance to avoid the pathology of multiple interfaces containing the same method name but different signatures. A class in CORE MIXGEN cannot implement such a collection of interfaces because method overloading is not available. The declared bound of every type parameter is an interface. Similarly, every mixin implements exactly one interface. The root of the interface hierarchy is the universal `Object` interface. For brevity, constructor signatures are included in the bounding interfaces of type variables instead in of distinct `with` clauses.

4.1 Syntax

A program in CORE MIXGEN consists of a sequence of mixin definitions and generic interface definitions followed by a single expression e . The program is executed by evaluating e . The body of each method consists of a single `return` statement.

$PROG$	\mapsto	$\overline{DEF} e$
DEF	\mapsto	$\text{class } C\langle\overline{X}\rangle \text{ extends } \overline{J}\rangle \text{ extends } X_i \text{ implements } J' \{ \overline{T} \overline{f}; \overline{K} \overline{M} \}$ $\quad $ $\quad \text{interface } I\langle\overline{X}\rangle \text{ extends } \overline{J}\rangle \text{ extends } J' \{ \overline{INIT} \overline{AM} \}$ $\quad $ $\quad \text{interface } I\langle\overline{X}\rangle \text{ extends } \overline{J}\rangle \text{ extends Object } \{ \overline{INIT} \overline{AM} \}$
$INIT$	\mapsto	$C\langle\overline{T} \overline{x}\rangle;$
K	\mapsto	$C\langle\overline{T} \overline{x}\rangle \{ \text{super}(\overline{e}); \text{this}.\overline{f} = \overline{e}'; \}$
AM	\mapsto	$\langle\overline{X}\rangle \text{ extends } \overline{T}\rangle T m(\overline{T} \overline{x});$
M	\mapsto	$\langle\overline{X}\rangle \text{ extends } \overline{T}\rangle T m(\overline{T} \overline{x}) \{ \text{return } e; \}$
e	\mapsto	x $\quad $ $\quad \text{this}.\overline{f}$ $\quad $ $\quad e.m\langle\overline{T}\rangle(\overline{e})$ $\quad $ $\quad \text{new } T(\overline{e})$ $\quad $ $\quad (T)e$ $\quad $ $\quad e \text{ instanceof } T$
T	\mapsto	X $\quad $ N
N	\mapsto	$C\langle\overline{T}\rangle$
J	\mapsto	$I\langle\overline{T}\rangle$

Figure 4: Syntax

The syntax of CORE MIXGEN is given in Figure 4. In the sequel, the formal rules of the language use the following meta-variables:

- C, D range over class names.
- I, J range over interface names.
- X, Y, Z range over type variables.
- N, O, P range over non-variable types.
- A, Q, R, S, T, U, V range over all types.
- f, g range over field names.
- m, n range over method names.
- x, y, z range over method parameter names.

Following the notation of Featherweight GJ, a variable with a horizontal bar above it represents a (possibly empty) sequence of elements in the domain of that variable, with elements separated by commas. For example, \bar{T} represents a sequence of types T_0, \dots, T_N . As in Featherweight GJ, we abuse this notation slightly in select contexts so that, for example, $\bar{T} \bar{f}$ represents a sequence of the structure $T_0 f_0, \dots, T_N f_N$. To simplify complications with variable substitution, all class names, interface names, method names, field names, type parameters, and method parameters in a CORE MIXGEN program are required to be distinct.

In CORE MIXGEN, all generic types including type variables are first-class and can appear in casts, `new` expressions, `instanceof` tests, and `extends` clauses of class definitions. In fact, because all class definitions are mixin definitions, *only* type variables are allowed in `extends` clauses. All classes are declared to extend exactly one type variable.

To avoid complications with implementing multiple (possibly incompatible) interfaces, all classes are required to implement exactly one interface, and the implemented interface must be a subinterface of the bound on the parent type. Conceptually, one can think of the implemented interface as the analog of the union of all the interfaces a mixin would implement in the full language. All bounds on type parameters are required to be interfaces. As in F-bounded polymorphism, these bounds may contain type parameters declared in the same scope [13]. An interface can extend either an interface instantiation or `Object`.

When mixins and interfaces are instantiated, all type parameters must be instantiated only with mixin instantiations. This restriction allows us to avoid some checks required in the full language such as enforcing the fact that the superclass of a mixin instantiation is not an interface. Mixin instantiations and interface instantiations are represented by distinct meta-variables in the grammar.

The function CT (short for “class table”) takes the name of any class (except `Object`) and returns the corresponding class definition. Like Featherweight GJ, CORE MIXGEN models the class `Object` simply as a tag without a corresponding class definition included in the class table. In CORE MIXGEN this design decision has additional motivation: in the absence of a non-mixin root class `Object`, there would be no base case in the inductive definition of a mixin instantiation. The CORE MIXGEN class `Object` contains no methods or type variables,

The `Object` tag may also be used as a static type declaration and a bounding interface. When used in these contexts, `Object` acts as if it were an interface with a single zero-ary constructor signature:

```
Object();
```

$$\Delta(C\langle\bar{X}\rangle) = C\langle\bar{X}\rangle$$

$$\Delta(X) = N \text{ where } \{X \triangleleft N\} \subseteq \Delta$$

Figure 5: Type Bounds Environments

Every subtype of `Object` must implement this zero-ary constructor.

The following class definitions for the `Boolean` type hierarchy are included in the class table of every program:

```
interface Boolean<> extends Object {}
class True<T extends Object> extends T implements Boolean { True() {super();} }
class False<T extends Object> extends T implements Boolean { False() {super();} }
```

The shorthand notations `true` and `false` respectively represent

```
new True<Object>()
new False<Object>()
```

In `CORE MIXGEN`, all fields are private. This restriction is enforced syntactically: only field accesses on `this` are grammatical. Unlike hygienic mixin method invocation, mixin field access in `MIXGEN` is straightforward (as in Java, it is based solely on the static type of the receiver), so we chose to simplify the core language by leaving out the shadowing issues that arise with public fields.

4.2 Type Environments

As in Featherweight GJ, the typing rules of `CORE MIXGEN` include two environments. First, a type environment Γ , maps program variables to their static types. Syntactically, these mappings are of the form $\bar{x} : \bar{T}$. `CORE MIXGEN` typing judgments also require a bounds environment Δ to map type variables to their upper bounds. Syntactically, these mappings are of the form $\bar{X} \triangleleft \bar{T}$. Notice that the bound of a type variable is always an interface instantiation. The bound of a non-variable type N is N . When both environments are relevant to a typing judgment, they appear together, separated by a semicolon. The extension of an environment E with environment E' is written as $E + E'$.

We use the notation $[\bar{X} \mapsto \bar{Y}]e$ to signify the substitution of all free occurrences of X for Y in e . The superscript of an expression corresponds to the static type of the initial expression before reduction. As an expression is reduced, the original static type is preserved.

4.3 Well-formedness

The rules for well-formed constructs appear in Figure 6. A type instantiation is well-formed in a bounds environment Δ if all instantiations of type parameters are subtypes of their formal types in Δ . Method definitions are well-formed in Δ if the constituent types are well-formed and the type of the body in Δ is a subtype of the declared type. Interface definitions are well-formed if the constituent elements are well-formed.

Unlike Featherweight GJ, `CORE MIXGEN` allows multiple constructors in a class, and the arguments to a constructor need not be directly related to the fields of the class. This feature is important in order to allow a mixin to implement multiple constructor signatures in its bounding interface. Because there is no `null` value, all constructors are required to assign values to all fields. Class definitions are well-formed if:

- the constituent elements are well-formed;
- all field assignments are of the appropriate type;

$$\begin{array}{c}
\Delta \vdash \text{Object ok} \quad \frac{X \in \text{dom}(\Delta)}{\Delta \vdash X \text{ ok}} \quad \frac{\text{PROG} = \overline{\text{DEF}} e \quad \overline{\text{DEF}} \text{ ok} \quad \emptyset; \emptyset \vdash e \in T}{\text{PROG ok}} \\
\\
\frac{\Delta \vdash \bar{T} \text{ ok} \quad \Delta \vdash I' \text{ ok} \quad \Delta \vdash \overline{AM} \text{ ok} \quad \Delta \vdash \overline{INIT} \text{ ok}}{\text{interface } I \langle \bar{X} \text{ extends } \bar{T} \rangle \text{ extends } I' \{ \overline{INIT} \overline{AM} \} \text{ ok} \\ \text{interface } I \langle \bar{X} \text{ extends } \bar{T} \rangle \text{ extends Object } \{ \overline{INIT} \overline{AM} \} \text{ ok}} \\
\\
\frac{\Delta + \bar{Y} \triangleleft \bar{T} \vdash \bar{T} \text{ ok} \quad \Delta + \bar{Y} \triangleleft \bar{T} \vdash T \text{ ok} \quad \Delta + \bar{Y} \triangleleft \bar{T} \vdash \bar{S} \text{ ok} \quad \Delta + \bar{Y} \triangleleft \bar{T} \vdash V \leq T}{\Delta \vdash \langle \bar{Y} \text{ extends } \bar{T} \rangle T \ m(\bar{S} \ \bar{v}) \text{ ok}} \\
\\
\frac{\Delta + \bar{Y} \triangleleft \bar{T} \vdash \bar{T} \text{ ok} \quad \Delta + \bar{Y} \triangleleft \bar{T} \vdash T \text{ ok} \quad \Delta + \bar{Y} \triangleleft \bar{T} \vdash \bar{S} \text{ ok} \quad \Delta + \bar{Y} \triangleleft \bar{T} \vdash V \leq T \quad \Delta + \bar{Y} \triangleleft \bar{T}; \Gamma + \bar{v}: \bar{S} \vdash e \in V}{\Delta; \Gamma \vdash \langle \bar{Y} \text{ extends } \bar{T} \rangle T \ m(\bar{S} \ \bar{v}) \{ \text{return } e; \} \text{ ok}} \\
\\
\frac{\Delta \vdash \bar{A} \text{ ok} \quad \Delta \vdash \bar{A} \leq [\bar{X} \mapsto \bar{A}] \bar{T} \quad \text{CT}(C) = \text{class } C \langle \bar{X} \text{ extends } \bar{T} \rangle \text{ extends } X_i \text{ implements } I' \{ \bar{T} \ \bar{f}; \bar{K} \ \bar{M} \}}{\Delta \vdash C \langle \bar{A} \rangle \text{ ok}} \\
\\
\frac{\Delta \vdash \bar{A} \text{ ok} \quad \Delta \vdash \bar{A} \leq [\bar{X} \mapsto \bar{A}] \bar{T} \quad \text{CT}(C) = \text{interface } I \langle \bar{X} \text{ extends } \bar{T} \rangle \text{ extends } I' \{ \overline{INIT} \overline{AM} \}}{\Delta \vdash I \langle \bar{A} \rangle \text{ ok}} \\
\\
\text{CT}(S) = \text{interface } I' \langle \bar{Y} \text{ extends } \bar{N} \rangle \text{ extends } I'' \{ \overline{INIT} \overline{AM} \} \\
\text{CT}(C) = \text{class } C \langle \bar{X} \text{ extends } \bar{T} \rangle \text{ extends } X_i \text{ implements } I' \{ \bar{T} \ \bar{f}; \bar{K} \ \bar{M} \} \\
\bar{X} \triangleleft \bar{T} \vdash I'' \leq N_i \\
\bar{X} \triangleleft \bar{T} \vdash \text{mtype}(I', m) = \text{mtype}(C \langle \bar{X} \rangle, m) \\
\text{includes}(C \langle \bar{X} \rangle, m) \text{ implies } \bar{X} \triangleleft \bar{T}; \bar{f}: \bar{T} + \text{this}: C \langle \bar{X} \rangle \vdash m \text{ ok} \\
\text{includes-constructor}(I, \bar{A}) \text{ implies includes-constructor}(C \langle \bar{X} \rangle, \bar{A}) \\
K_i = C(\bar{C} \ \bar{x}) \{ \text{super}(\bar{p}); \text{this}.\bar{f} = \bar{e}'; \} \text{ implies includes-constructor}(I_i, \bar{T}_1) \\
\text{and } \bar{X} \triangleleft \bar{T}; \bar{f}: \bar{T} + \bar{x}: \bar{C} \vdash \bar{p} \in \bar{T}_2 \\
\text{and } \bar{X} \triangleleft \bar{T}; \bar{f}: \bar{T} + \bar{x}: \bar{C} \vdash \bar{T}_2 \leq \bar{T}_1 \\
\text{and } \bar{X} \triangleleft \bar{T}; \bar{f}: \bar{T} + \bar{x}: \bar{C} \vdash \bar{e} \in \bar{T}_3 \\
\text{and } \bar{X} \triangleleft \bar{T} \vdash \bar{T}_3 \leq \bar{T} \\
\hline
\text{class } C \langle \bar{X} \text{ extends } \bar{T} \rangle \text{ extends } X_i \text{ implements } I' \{ \bar{T} \ \bar{f}; \bar{K} \ \bar{M} \} \text{ ok}
\end{array}$$

Figure 6: Well-formed Constructs

$$\begin{array}{c}
K_i = C(\overline{U} \overline{x}) \{ \dots \} \\
\hline
CT(C) = \text{class } C\langle \overline{Y} \rangle \text{ extends } \overline{T} \rangle \text{ extends } Y_i \text{ implements } I' \{ \overline{T} \overline{f}; \overline{K} \overline{M} \} \\
\hline
\text{includes-constructor}(C\langle \overline{R} \rangle, [\overline{Y} \mapsto \overline{R}] \overline{U}) \\
\\
INIT_i = I(\overline{U} \overline{x}) \\
\hline
CT(I) = \text{interface } I\langle \overline{X} \rangle \text{ extends } \overline{T} \rangle \text{ extends } I' \{ \overline{INIT} \overline{AM} \} \\
\hline
\text{includes-constructor}(I\langle \overline{R} \rangle, [\overline{X} \mapsto \overline{R}] \overline{U}) \\
\\
\text{includes-constructor}([\overline{X} \mapsto \overline{Q}] I', \overline{R}) \\
\hline
CT(I) = \text{interface } I\langle \overline{X} \rangle \text{ extends } \overline{T} \rangle \text{ extends } I' \{ \overline{INIT} \overline{AM} \} \\
\hline
\text{includes-constructor}(I\langle \overline{Q} \rangle, \overline{R})
\end{array}$$

Figure 7: Constructor Signature Checking

- all method definitions are well-formed in the scope of the class definition;
- all methods in the implemented interface are implemented with methods of the correct signatures;
- the implemented interface is a subtype of the bound of the parent type; and
- all constructor signatures in super-interfaces are implemented with well-formed constructors.

The rules for checking constructors appear in Figure 7. When checking class well-formedness, `this` is added to the type environment before checking each method for well-formedness.

A program is well-formed if all class and interface definitions are well-formed, and the tailing expression can be typed with the empty type and bounds environments.

$$\begin{array}{c}
\text{reflex-sub: } \Delta \vdash T \leq T \quad \text{trans-sub: } \frac{\Delta \vdash S \leq T \quad \Delta \vdash T \leq U}{\Delta \vdash S \leq U} \\
\\
\text{object-sub: } \Delta \vdash T \leq \text{Object} \quad \text{bound-sub: } \Delta \vdash X \leq \Delta(X) \\
\\
\text{class-sub: } \frac{CT(C) = \text{class } C\langle \overline{X} \rangle \text{ extends } \overline{T} \rangle \text{ extends } X_i \text{ implements } I' \{ \overline{T} \overline{f}; \overline{K} \overline{M} \}}{\Delta \vdash C\langle \overline{T} \rangle \leq T_i \quad \Delta \vdash C\langle \overline{T} \rangle \leq [\overline{X} \mapsto \overline{T}] I'} \\
\\
\text{interface-sub: } \frac{CT(S) = \text{interface } I\langle \overline{X} \rangle \text{ extends } \overline{T} \rangle \text{ extends } I' \{ \overline{INIT} \overline{AM} \}}{\Delta \vdash I\langle \overline{T} \rangle \leq [\overline{X} \mapsto \overline{T}] I'}
\end{array}$$

Figure 8: Subtyping

4.4 Subtyping

Rules for subtyping appear in Figure 8. The subtyping relation is represented with the symbol \leq . Subtyping is reflexive and transitive, and mixin instantiations are subtypes of the instantiations of their parent types.

4.5 Expression Typing

$$\begin{array}{c}
\Delta \vdash \bar{Q} \text{ ok} \\
\hline
CT(C) = \text{class } C \langle \bar{X} \text{ extends } \bar{I} \rangle \text{ extends } X_i \text{ implements } I' \{ \dots \langle \bar{Y} \text{ extends } \bar{T} \rangle R m(\bar{R} \bar{x}) \{ \text{return } e; \} \dots \} \\
\hline
\Delta \vdash \text{mtype}(m, C \langle \bar{Q} \rangle) = [\bar{X} \mapsto \bar{Q}] (\langle \bar{Y} \text{ extends } \bar{T} \rangle R m(\bar{R} \bar{x})) \\
\\
\Delta \vdash \bar{Q} \text{ ok} \\
\hline
CT(I) = \text{interface } I \langle \bar{X} \text{ extends } \bar{N} \rangle \text{ extends } I' \{ \dots \langle \bar{Y} \text{ extends } \bar{N} \rangle R m(\bar{R} \bar{x}) \dots \} \\
\hline
\Delta \vdash \text{mtype}(m, I \langle \bar{Q} \rangle) = [\bar{X} \mapsto \bar{Q}] (\langle \bar{Y} \text{ extends } \bar{N} \rangle R m(\bar{R} \bar{x})) \\
\\
\Delta \vdash \bar{Q} \text{ ok} \quad \text{-includes}(m, C) \\
\hline
CT(C) = \text{class } C \langle \bar{X} \text{ extends } \bar{I} \rangle \text{ extends } X_i \text{ implements } I' \{ \bar{T} \bar{f}; \bar{K} \bar{M} \} \\
\hline
\Delta \vdash \text{mtype}(m, C \langle \bar{Q} \rangle) = \text{mtype}(m, [\bar{X} \mapsto \bar{Q}] I_i) \\
\\
\Delta \vdash \bar{Q} \text{ ok} \quad \text{-includes}(m, C) \\
\hline
CT(I) = \text{interface } I \langle \bar{X} \text{ extends } \bar{I} \rangle \text{ extends } I' \{ \overline{INIT} \overline{AM} \} \\
\hline
\Delta \vdash \text{mtype}(m, I \langle \bar{Q} \rangle) = \text{mtype}(m, [\bar{X} \mapsto \bar{Q}] I')
\end{array}$$

Figure 9: Method Type Resolution

The function **mtype**, defined in Figure 9, takes a type and a method name and returns a meta-level “method signature” type consisting of the method’s bound type parameters, its parameter types, and its return type. Method types are determined simply by searching upward from the static type for the first match. Because there is no multiple interface inheritance, and because there is a least upper bound on the chain of interfaces implemented by a mixin, there is always a unique supertype to resort to during an upward search.

$$\begin{array}{c}
\text{var-type: } \Delta; \Gamma \vdash x \in \Gamma(x) \qquad \text{instanceof-type: } \frac{\Delta; \Gamma \vdash e \in S \quad \Delta \vdash T \text{ ok}}{\Delta; \Gamma \vdash e \text{ instanceof } T \in \text{Boolean}} \\
\\
\text{field-type: } \frac{\text{fields}(C \langle \bar{X} \rangle) = \bar{T} \bar{f} \quad \Delta; \Gamma \vdash \text{this} \in C \langle \bar{X} \rangle}{\Delta; \Gamma \vdash \text{this} \cdot f_i \in T_i} \\
\\
\text{cast-type: } \frac{\Delta \vdash T \text{ ok} \quad \Delta; \Gamma \vdash e \in S}{\Delta; \Gamma \vdash (T)e \in T} \qquad \text{new-type: } \frac{\text{includes-constructor}(\Delta(T), \bar{T}) \quad \Delta; \Gamma \vdash \bar{e} \in \bar{T} \quad \Delta \vdash T \text{ ok}}{\Delta; \Gamma \vdash \text{new } T(\bar{e}) \in T} \\
\\
\text{app-type: } \frac{\Delta \vdash \bar{T} \text{ ok} \quad \Delta; \Gamma \vdash e_0 \in T_0 \quad \Delta; \Gamma \vdash \bar{e} \in \bar{R} \quad \Delta \vdash \bar{R} \leq [\bar{X} \mapsto \bar{T}] \bar{S} \quad \Delta \vdash \bar{T} \leq [\bar{X} \mapsto \bar{T}] \bar{Y} \quad \Delta \vdash \text{mtype}(m, T_0) = \langle \bar{X} \text{ extends } \bar{Y} \rangle S m(\bar{S} \bar{x})}{\Delta; \Gamma \vdash e_0 \cdot m \langle \bar{T} \rangle(\bar{e}) \in [\bar{X} \mapsto \bar{T}] S}
\end{array}$$

Figure 10: Expression Typing

The rules for expression typing are listed in Figure 10. Because all fields are private, we need only specify a typing rule for field access on `this`. Recall that `this` is bound in the type environment when checking method bodies. Naked type variables may occur in `new` expressions and casts. When checking `new` expressions of naked type, the bound of the type variable is checked to ensure that it includes an appropriate constructor signature.

In Featherweight GJ, stupid casts (the casting of an expression to an incompatible type), were identified as a complication with type soundness. In that language, a special rule was added to the type system that allowed expressions with subexpressions that reduced to stupid casts to continue to be typed during evaluation, so as not to violate subject reduction. Stupid casts were untypable only when they occurred in the original program text, before reduction. In CORE MIXGEN, this issue does not arise, because CORE MIXGEN does not check for stupid casts in a program. Because mixin instantiations are not resolved until run time, rarely is it possible to statically detect a stupid cast. They can be detected either when a ground type (i.e., a type containing no type variables) is cast to an incompatible ground type, or when the bounding interface of a mixin instantiation is incompatible with the bounding interface of the target type of the cast. Therefore, for the sake of simplicity, we simply allow all casts to pass type checking.

Like Featherweight GJ, CORE MIXGEN requires explicit polymorphism on parametric methods.

$$\begin{array}{c}
\text{mbody}(m\langle\bar{V}\rangle, T', C\langle\bar{S}\rangle) = (\bar{x}, e_0) \\
\text{fields}(C\langle\bar{S}\rangle) = \bar{U} \bar{f} \quad \text{field-vals}(\text{new } C\langle\bar{S}\rangle(\bar{e})) = \bar{e}' \\
\text{app-comp: } \frac{}{(\text{new } C\langle\bar{S}\rangle(\bar{e})).m\langle\bar{V}\rangle(\bar{d})^T \rightarrow [\bar{x} \mapsto \bar{d}][\bar{f} \mapsto \bar{e}'][\text{this} \mapsto \text{new } C\langle\bar{S}\rangle(\bar{e})]e_0^T} \\
\text{cast-comp: } \frac{\emptyset \vdash N \leq O}{(O) (\text{new } N(\bar{e})^T)^O \rightarrow \text{new } N(\bar{e})^O} \quad \text{field-comp: } \frac{\text{fields}(C\langle\bar{S}\rangle) = \bar{T} \bar{f} \quad \text{field-vals}(\text{new } C\langle\bar{S}\rangle(\bar{e})) = \bar{e}'}{\text{new } C\langle\bar{S}\rangle(\bar{e}).f_i^T \rightarrow e_i'^T} \\
\text{instanceof-true-comp: } \frac{\emptyset \vdash C\langle\bar{S}\rangle \leq D\langle\bar{T}\rangle}{\text{new } C\langle\bar{S}\rangle(\bar{e}) \text{ instanceof } D\langle\bar{T}\rangle^T \rightarrow \text{true}^T} \\
\text{instanceof-false-comp: } \frac{\emptyset \vdash C\langle\bar{S}\rangle \not\leq D\langle\bar{T}\rangle}{\text{new } C\langle\bar{S}\rangle(\bar{e}) \text{ instanceof } D\langle\bar{T}\rangle^T \rightarrow \text{false}^T} \\
\text{app-cong: } \frac{e^T \rightarrow e'^T}{e^T.m\langle\bar{U}\rangle(\bar{a})^R \rightarrow e'^T.m\langle\bar{U}\rangle(\bar{a})^R} \\
\text{cast-cong: } \frac{e^T \rightarrow e'^T}{((O)e^T)^O \rightarrow ((O)e'^T)^O} \quad \text{field-cong: } \frac{e^T \rightarrow e'^T}{e^T.f_i \rightarrow e'^T.f_i} \\
\text{instanceof-cong: } \frac{e^T \rightarrow e'^T}{e^T \text{ instanceof } C\langle\bar{A}\rangle^R \rightarrow e'^T \text{ instanceof } C\langle\bar{A}\rangle^R} \\
\text{arg-cong: } \frac{e^T \rightarrow e'^T}{e^N.m\langle\bar{R}\rangle(e_1^{T_1} \dots e_n^{T_n})^R \rightarrow e'^N.m\langle\bar{R}\rangle(e_1^{T_1} \dots e_n^{T_n})^R}
\end{array}$$

Figure 11: Computation

$$\begin{array}{c}
\frac{CT(C) = \text{class } C \langle \bar{X} \text{ extends } \bar{T} \rangle \text{ extends } X_i \text{ implements } I' \{ \dots \langle \bar{Y} \text{ extends } \bar{M} \rangle S m(\bar{S} \bar{x}) \{ \text{return } e; \} \dots \}}{\text{includes}(C, m)} \\
\\
\frac{CT(I) = \text{interface } I \langle \bar{X} \text{ extends } \bar{N} \rangle \text{ extends } I' \{ \dots \langle \bar{Y} \text{ extends } \bar{M} \rangle S m(\bar{S} \bar{x}) \dots \}}{\text{includes}(I, m)} \\
\\
\text{includes}(I', m) \\
\frac{CT(I) = \text{interface } I \langle \bar{X} \text{ extends } \bar{T} \rangle \text{ extends } I' \{ \overline{INIT} \overline{AM} \}}{\text{includes}(I, m)}
\end{array}$$

Figure 12: Direct Method Inclusion

$$\begin{array}{c}
\frac{CT(C) = \text{class } C \langle \bar{X} \text{ extends } \bar{T} \rangle \text{ extends } X_i \text{ implements } I' \{ \dots \langle \bar{Y} \text{ extends } \bar{M} \rangle T m(\bar{T} \bar{x}) \{ \text{return } e; \} \dots \}}{\text{mbody-simple}(m \langle \bar{U} \rangle, C \langle \bar{Q} \rangle) = (\bar{x}, [\bar{Y} \mapsto \bar{U}][\bar{X} \mapsto \bar{Q}])e} \\
\\
\neg \text{includes}(m, C) \\
\frac{CT(C) = \text{class } C \langle \bar{X} \text{ extends } \bar{T} \rangle \text{ extends } X_i \text{ implements } I' \{ \bar{T} \bar{f}; \bar{K} \bar{M} \}}{\text{mbody-simple}(m \langle \bar{U} \rangle, C \langle \bar{Q} \rangle) = \text{mbody-simple}(m \langle \bar{U} \rangle, Q_i)} \\
\\
\text{mbody}(m \langle \bar{U} \rangle, C \langle \bar{T} \rangle, C \langle \bar{T} \rangle) = \text{mbody-simple}(m \langle \bar{U} \rangle, C \langle \bar{T} \rangle) \\
\\
\neg \text{includes}(m, [\bar{X}_E \mapsto \bar{X}]I) \\
\text{chain}(C \langle \bar{R} \rangle, D \langle \bar{Q} \rangle) = C \langle \bar{R} \rangle, \dots, E \langle \bar{Z} \rangle, D \langle \bar{Q} \rangle \quad C \neq D \\
\frac{CT(E) = \text{class } E \langle \bar{X}_E \text{ extends } \bar{N}_E \rangle \text{ extends } X_{E_i} \text{ implements } I \{ \dots \}}{\text{mbody}(m \langle \bar{U} \rangle, D \langle \bar{Q} \rangle, C \langle \bar{R} \rangle) = \text{mbody-simple}(m \langle \bar{U} \rangle, D \langle \bar{Q} \rangle)} \\
\\
\text{includes}(m, [\bar{X}_E \mapsto \bar{X}]I) \\
\text{chain}(C \langle \bar{R} \rangle, D \langle \bar{Q} \rangle) = C \langle \bar{R} \rangle, \dots, E \langle \bar{T} \rangle, D \langle \bar{Q} \rangle \quad C \neq D \\
\frac{CT(E) = \text{class } E \langle \bar{X}_E \text{ extends } \bar{N}_E \rangle \text{ extends } X_{E_i} \text{ implements } I \{ \dots \}}{CT(D) = \text{class } D \langle \bar{X}_D \text{ extends } \bar{N}_D \rangle \text{ extends } X_{D_i} \text{ implements } I' \{ \dots \}} \\
\text{mbody}(m \langle \bar{U} \rangle, D \langle \bar{Q} \rangle, C \langle \bar{R} \rangle) = \text{mbody}(m \langle \bar{U} \rangle, E \langle \bar{T} \rangle, C \langle \bar{R} \rangle)
\end{array}$$

Figure 13: Method Lookup

4.6 Computation

The CORE MIXGEN computation rules are defined in Figure 10. Because the static type of a receiver is used to resolve method applications, static types must be preserved during computation as superscripts on expressions. When computing the application of a method, the appropriate method body is found using **mbody**. The application is then reduced to the body of the method, substituting all parameters with their instantiations, and **this** with the receiver.

The function **mbody**, defined in Figure 13, takes three arguments: a method name with instantiated

$$\begin{array}{l}
\mathbf{fields}(\mathbf{Boolean}) = \emptyset \quad \mathbf{fields}(\mathbf{true}) = \emptyset \quad \mathbf{fields}(\mathbf{false}) = \emptyset \quad \mathbf{fields}(\mathbf{Object}) = \emptyset \\
\\
\frac{CT(C) = \mathbf{class} \ C \langle \bar{X} \rangle \ \mathbf{extends} \ \bar{I} \rangle \ \mathbf{extends} \ X_i \ \mathbf{implements} \ I' \ \{\bar{I} \ \bar{f} \dots\}}{\mathbf{fields}(C \langle \bar{R} \rangle) = [\bar{X} \mapsto \bar{R}] \bar{I} \ \bar{f}}
\end{array}$$

Figure 14: Class Fields

$$\frac{\Delta \vdash \bar{S} = [\bar{X} \mapsto \bar{R}] \bar{T} \quad CT(C) = \mathbf{class} \ C \langle \bar{X} \rangle \ \mathbf{extends} \ \bar{I} \rangle \ \mathbf{extends} \ X_i \ \mathbf{implements} \ I' \ \{\dots C \langle \bar{T} \ \bar{g} \rangle \ \{\mathbf{this}.\bar{f} = \bar{e}'\}; \dots\}}{\mathbf{field-vals}(\mathbf{new} \ C \langle \bar{R} \rangle (\bar{e})^{\bar{S}}) = [\bar{X} \mapsto \bar{R}] [\bar{g} \mapsto \bar{e}] \bar{e}'}$$

Figure 15: Field Values

type parameters, a static type (preserved as a superscript on the receiver) and run-time type of the receiver. The function **chain** defines the sequence of class instantiations occurring from the run-time type to the static type of the receiver (a formal definition of **chain** is emitted for the sake of brevity, but the chain of parent instantiations can be obtained directly from the the run-time type, as all superclasses occur as instantiations of type parameters). **mbody** searches down this chain from the static type until it finds a mixin instantiation that does not include the method name in its implemented interface. At that point, it calls the helper function **mbody-simple**, which takes the same method name with instantiated type parameters and a type as input, and searches upward until it finds a mixin that includes the method directly in its body. The rules for direct method inclusion are provided in Figure 12.

4.7 Fields and Field Values

The rules for the retrieval of the field names and types of a class (used by the typing and computation rules on field lookup) are provided in Figure 14.

The rules for the retrieval of field values from an object are provided in Figure 15. Notice that inclusion of multiple constructors for a class, where constructor signatures do not directly match the field types of a class, complicates field value lookup in comparison to Featherweight GJ. It is important that a **new** expression is matched to the constructor of the appropriate signature. In order to avoid complications with resolving multiple matching constructors, CORE MIXGEN requires that the static types of the arguments to a **new** expression *exactly* match a constructor of the corresponding class. Casts can always be used to ensure that the static types of the arguments satisfy this requirement.

5 Type Soundness

We sketch the proof of a type soundness theorem for CORE MIXGEN. The full proof will be available in a forthcoming technical report [3]. We start by stating several supporting lemmas. In particular, we must establish some lemmas concerning the preservation of properties under variable and type variable substitution. Because computation in CORE MIXGEN, as in Featherweight GJ, consists almost entirely of method application, and because method application consists of substituting variables into the body of a

method and reducing it, the preservation of properties under substitution plays a central role in our type soundness theorem.

One complication that arises is the relationship between field types in the lexical scope of a class to the return types of method applications on instances of the class. In CORE MIXGEN, the constructor can assign arguments to fields which are not directly related to constructor arguments. As a result, the type bounds environment in which a field is assigned, and the environment enclosing the reduction of a getter, have no relationship with one another in general. At first glance, this appears to pose a serious problem for a type preservation lemma: how do we relate the type of a field returned from a getter to the static type of the method invocation?

Featherweight GJ avoids this problem entirely because the values assigned to all fields must be provided directly in the arguments to a constructor call (which also entails that there can be only one constructor per class). Since Featherweight GJ requires receivers to be reduced to `new` expressions before method invocation, all fields of a receiver are defined in the same environment as any method invocation, including getters. In the case of CORE MIXGEN, we can't require all field values to be provided as arguments in constructor calls without losing an important property of the modeled language, namely that distinct parent instantiations of a mixin can have distinct sets of fields. Instead, we would have to require all instantiations of a mixin's parent to have an identical set of fields and a single identical constructor signature.

Fortunately, this seeming difficulty goes away when one considers that the only expressions that are ever reduced during a program evaluation are ground expressions (i.e., expressions with no free type variables).

Lemma 1 (Program Expression Groundedness) *If a program computation includes the reduction $e \rightarrow e'$ then e and e' must be ground expressions.*

Proof Sketch. The initial body in a well-typed program is required to be ground, as it occurs in an empty type environment. Also, reduction of a ground expression will always yield a ground expression. Thus, all expressions that a program reduces to are ground. This implies that all reduced expressions are ground and that $\Delta; \Gamma$ are empty during evaluation.

Now establishing the following two substitution lemmas is straightforward:

Lemma 2 (Preservation of Typing Under Type Variable Substitution) *For ground types \bar{S} , if $\bar{X} \triangleleft \bar{N}; \Gamma \vdash e \in T$ and $\bar{X} \triangleleft \bar{N} \vdash \bar{S} \leq \bar{N}$ then $\bar{X} \triangleleft \bar{N}; [\bar{X} \mapsto \bar{S}]\Gamma \vdash [\bar{X} \mapsto \bar{S}]e \in [\bar{X} \mapsto \bar{S}]T$.*

Lemma 3 (Preservation of Typing Under Variable Substitution) *Given $\bar{X} \triangleleft \bar{N}; \bar{v} : \bar{B} \vdash e \in T$ and $\bar{X} \triangleleft \bar{N}, \bar{v} : \bar{B} \vdash \bar{y} \in \bar{B}'$ and $\bar{X} \triangleleft \bar{N} \vdash \bar{B}' \leq \bar{B}$ then $\bar{X} \triangleleft \bar{N}; \bar{v} : \bar{B} \vdash [\bar{v} \mapsto \bar{y}]e \in T_1$ where $\bar{X} \triangleleft \bar{N} \vdash T_1 \leq T$.*

Proof Sketch By structural induction over the typing rules.

With these theorems in hand, we can establish the following subject reduction theorem:

Theorem 1 (Preservation of Types Under Subject Reduction) *For type and bounds environments Δ, Γ , ground expression e , and type T , if $\Delta; \Gamma \vdash e \in T$ and $e \rightarrow e'$ then $\Delta; \Gamma \vdash e' \in S$ where $\Delta; \Gamma \vdash S \leq T$.*

Proof Sketch By case analysis over the computation rules. The most significant case is method application, where we apply the substitution lemmas above to establish that the resulting expression is of the appropriate type.

We also state the following progress theorem:

Theorem 2 (Progress) *For a well-typed ground expression e :*

1. *If e is of the form $e'.f$ and $\vdash e' \in T$ then $\mathbf{fields}(T)$ includes f .*
2. *If e is of the form $e'^S.m\langle\bar{T}\rangle(e'')$ and $\vdash e' \in R$ then $\mathbf{includes}(R, m)$ and $\mathbf{mbody}(m\langle\bar{T}\rangle, S, R)$ is well-defined.*
3. *If e is of the form $\mathbf{new} T(\bar{e}')^S$ and $\vdash \bar{e}' \in \bar{T}'$ then $\mathbf{includes-constructor}(T, \bar{T}')$.*

Proof Sketch The first condition is greatly simplified by the fact that all fields are private, and follows immediately from **field-type**. The second condition is proved by structural induction over **includes** and **mbody**. Because e is well-typed, m must exist in the static type. Because mixins in CORE MIXGEN may not include abstract methods, any well-formed subtype of the static type must include a definition for m . Therefore, it suffices to show that **includes** and **mbody** will find this implementation. The third condition is shown by structural induction on **includes-constructor**. Again, because e is well-typed, the called constructor must exist in the static type, and therefore must be included in any well-formed subtype.

6 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 Java compilers (*e.g.*, `javac`) 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 [6] and Jiazzi [19], 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. 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[19] 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 program 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 Java

language, it is not directly comparable to MixGen. 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.¹² Since mixins can only be instantiated in the meta-language used to wire components together, Jiazzi does not have address the same language design issues as MIXGEN. 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.

The language design that is most closely related to MIXGEN is MIXEDJAVA, a toy language loosely based on Java, developed by Flatt, Krishnamurthi, and Felleisen. MIXEDJAVA is a language similar to CORE MIXGEN but it does not support genericity. In MIXEDJAVA, all classes are constructed by mixins. To specify both the static types of expressions and the dynamic types of program values, MIXED JAVA uses special type expressions called consisting of sequences of mixin names. Every value in MIXED JAVA 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 tagging of values with types significantly affects the semantics of the language. A naive translation of a MIXEDJAVA program into NEXTGEN will not generally preserve the meaning of the original program. Consider the following MIXGEN code fragment:

```
class C<T extends J> extends T implements J {
    ...
    ... m(...) {... n(...) ...}
    ... n(...) {...}
}
```

where m is in J but n is not. Now consider the class $C<C<A>>$ where A implements I . Let y be an object of class $C<C<A>>$. In MIXGEN, casting y to static type $C<A>$ before invoking m

```
((C<A>)y).m(...)
```

has no effect, because m is *overridden* in each application of the mixin C in $C<C<A>>$. The invoked method is the code for m in the second mixin application. Hence, if m subsequently invokes the method n , the static type of `this` is $C<C<A>>$ implying that the version of n in introduced in $C<C<A>>$ will be invoked.

In contrast, the corresponding MIXEDJAVA program embeds the type tag $<C<A>$ as part of the value of y . Hence, the invocation of n on `this` in the body of m will dispatch with respect to the type $<C<A>$ and invoke the version of n of $C<A>$. This difference reflects that fact that object views are dynamically attached to objects in MIXEDJAVA while they are statically attached to program expressions in MIXGEN.

In MIXGEN we can simulate the MixedJava semantics when needed by leveraging genericity to associate a dynamic view with an object of mixin type. In particular, we can parameterize any method that needs a dynamic view of an object by type T and cast the object within the method to type T .

The type system of MIXEDJAVA is less expressive than the type system of MIXGEN because it does not support genericity. In particular, MIXEDJAVA does not provide a type for the superclass of a mixin. For example, in the body of a mixin

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

¹²To 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.

MIXEDJAVA cannot name the type τ or $\langle T \rangle$. 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 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 the JVM in a way that preserves compatibility with legacy binary code.

7 Conclusion

Adding first class genericity to Java produces a surprisingly powerful language that supports precisely typed mixins as well as conventional generic classes. We have shown how the resulting language can be safely type checked and how it can be efficiently implemented on top of the existing Java Virtual Machine. We believe that these results provide a compelling argument for adding first class genericity to Java.

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. Mixins in Generic Java are Sound. Technical Report, Computer Science Department, Rice University, December 2002.
- [4] 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.
- [5] E. Allen, R. Cartwright. The Case for Run-time Types in Generic Java. *Principles and Practice of Programming in Java*, June 2002.
- [6] D. Ancona, G. Lagorio, E. Zucca. JAM-A Smooth Extension of Java with Mixins. ECOOP 00, LNCS, Springer Verlag, 2000.
- [7] J. Bloch, N. Gafter. Personal communication.
- [8] G. Bracha W. Cook. Mixin-based Inheritance. OOPSLA '90, October 1990 .
- [9] R. Cartwright, G. Steele. Compatible Genericity with Run-time Types for the Java Programming Language. In *OOPSLA '98*, October 1998.
- [10] G. Bracha. The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. Ph.D. dissertation, Dept. of Computer Science, University of Utah 1992.
- [11] 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.

- [12] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. GJ Specification. Online manuscript available at <http://www.cis.unisa.edu.au/pizza/gj/Documents/>, May 1998.
- [13] P. Cunning, W. Cook, W. Hill, W. Olthoff, J. Mitchell. F-bounded Quantification for Object-oriented Programming. In *Proc. of the ACM FPCA*. pp. 273-280. September 1989.
- [14] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass. 1995.
- [15] M. Flatt, S. Krishnamurthi, M. Felleisen. Classes and Mixins. In *POPL 1998*, January 1998.
- [16] 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.
- [17] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. In *POPL 1997*, January 1997, 146–159.
- [18] A. Igarashi, B. Pierce, P. Wadler Featherweight Java: A Minimal Core Calculus for Java and GJ. In *OOPSLA '99*, November 1999.
- [19] S. McDirmid, M. Flatt, W. Hsieh. Jiazzi: New Age Components for Old Fashioned Java. In *OOPSLA '01*, 2001.
- [20] D. Moon. Object-oriented Programming with Flavors. In *OOPSLA '86*, 1986.
- [21] A. Snyder. CommonObjects: An Overview. In *Proceedings of the 1986 SIGPLAN Workshop on Object-oriented Programming, Sigplan Notices* **21(10)**, 19-28, 1986.
- [22] G. Steele. Growing a Language. In *Journal of Higher-Order and Symbolic Computation* (Kluwer) **12** (3), October 1999, 221–236.
- [23] Sun Microsystems, Inc. JSR 14: Add Generic Types To The Java Programming Language. Available at <http://www.jcp.org/jsr/detail/14.jsp>.