

A First-Class Approach to Genericity^{*}

Eric Allen
Rice University
6100 South Main St.
Houston TX 77005
eallen@cs.rice.edu

Jonathan Bannet
Rice University
6100 South Main St.
Houston TX 77005
jbbannet@rice.edu

Robert Cartwright
Rice University
6100 South Main St.
Houston TX 77005
cork@cs.rice.edu

ABSTRACT

This paper describes how to add first-class generic types—including mixins—to strongly-typed OO languages with nominal subtyping such as Java and C#. A generic type system is “first-class” if generic types can appear in any context where conventional types can appear. In this context, a mixin is simply a generic class that extends one of its type parameters, *e.g.*, a class $C<T>$ that extends T . Although mixins of this form are widely used in C++ (via templates), they are clumsy and error-prone because C++ treats mixins as macros, forcing each mixin instantiation to be separately compiled and type-checked. The abstraction embodied in a mixin is never separately analyzed.

Our formulation of mixins using first-class genericity accommodates sound local (class-by-class) type checking. A mixin can be fully type-checked given symbol tables for each of the classes that it directly references—the same context in which Java performs incremental class compilation. To our knowledge, no previous formal analysis of first-class genericity in languages with nominal type systems has been conducted, which is surprising because nominal subtyping has become predominant in mainstream object-oriented programming languages.

What makes our treatment of first-class genericity particularly interesting and important is the fact that it can be added to the existing Java language without any change to the underlying Java Virtual Machine. Moreover, the extension is backward compatible with legacy Java source and class files. Although our discussion of a practical implementation strategy focuses on Java, the same implementation techniques could be applied to other object-oriented languages such as C# or Eiffel that support incremental compilation, dynamic class loading, and nominal subtyping.

^{*}This research has been partially supported by the Texas Advanced Technology Program, the National Science Foundation, and Sun Microsystems, Inc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA '03, October 26–30, 2003, Anaheim, California, USA.
Copyright 2003 ACM 1-58113-712-5/03/0010 ...\$5.00.

Categories and Subject Descriptors

D.1.5 [Programming Techniques]: Object-Oriented Programming; D.3.1 [Programming Languages]: Formal Definitions and Theory; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms

Design, Languages

1. INTRODUCTION

Static type systems with nominal subtyping (as in C++, Eiffel, Java, and C#) have been predominant in mainstream object-oriented languages for over a decade. Yet they have received comparatively little attention from language researchers [16, 24, 18]. Nominal type systems are popular among object-oriented language designers and programmers because they offer some significant advantages over static type systems with structural subtyping. They include:

- simple, intuitive type-checking rules;
- manifest type hierarchies;
- transparent support for mutually recursive data types; and
- easily understood run-time error diagnostics.

Unfortunately, the nominal type systems currently employed in mainstream OO languages are too weak to provide precise type checking. Although Java and C# are type-safe, the imprecision in their type systems often forces programs to perform run-time type checks (casts) which may abort program execution. The biggest weakness in their type systems 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 generic typing 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.

Both Java and C# will soon be extended to support generic types [19, 26], but neither of these extensions is fully first-class. The forthcoming extension for Java is second-class; it provides no support for operations that require generic type information at run-time. In contrast, C# will support operations that depend on run-time generic type information,

but it will not allow the superclass of a generic class to be a type parameter—precluding the definition of mixins.

In this paper, we show how to extend languages like Java and C# to support full first-class genericity including mixins. In the process, we prove that the resulting type system is sound for a core subset of Generic Java [12]. Moreover, in the case of Java, we show how this extension can be efficiently implemented on top of the existing Java Virtual Machine while retaining compatibility with legacy binaries.

1.1 Background

Before we discuss how to add first-class genericity to object-oriented languages with nominal subtyping, we need to briefly explain some of the key concepts underlying genericity and subtyping.

1.1.1 Nominal versus Structural Subtyping

In object-oriented languages with structural subtyping, objects are formally modeled as untagged records where a record type \mathbf{t}_1 is a subtype of record type \mathbf{t}_2 iff \mathbf{t}_1 includes all of the member names of \mathbf{t}_2 and the type of each member in \mathbf{t}_2 is a supertype of the corresponding member of \mathbf{t}_1 . There is no relationship between subtyping and code inheritance because the subtyping relationship between two record types depends only on the signatures of the members of the two types.

In object-oriented languages with nominal subtyping, objects are modeled as tagged records where the tag is typically the name of the class to which the object belongs. In such type systems, a class \mathbf{A} is a subtype of a class \mathbf{B} iff \mathbf{A} inherits members from \mathbf{B} . A class \mathbf{C} with exactly the same members as \mathbf{B} that does not inherit from \mathbf{B} is not a subtype of \mathbf{B} . Since inheritance relationships are explicitly declared by the programmer, the programmer specifically determines whether a given type is a subtype of another.

To illustrate the difference between nominal and structural subtyping, consider the following Java program fragment.

```
abstract class MultiSet {
    abstract public void insert(Object o);
    abstract public void remove(Object o);
    abstract public boolean isMember(Object o);
}

abstract class Set {
    abstract public void insert(Object o);
    abstract public void remove(Object o);
    abstract public boolean isMember(Object o);
}
```

Since Java is based on nominal subtyping, the two abstract classes `Set` and `MultiSet` have no subtyping relationship: no value of type `Set` (other than the degenerate value `null`) is a value of the type `MultiSet` and vice-versa. This distinction is appropriate for the abstract types `Set` and `MultiSet` since the contracts for their methods are incompatible.

In contrast, suppose that Java were structurally subtyped. Then `Set` would be a subtype of `MultiSet` and vice versa because the two classes have identical public member signatures. Structural subtyping forces subtyping relationships

on the basis of matching method signatures even when the method contracts conflict.

1.1.2 Generic Types

In a nominally-typed OO language with generic types, a class definition may be parameterized by type variables and program text may use generic (parameterized) types in many contexts instead of conventional types. For example, in Generic Java [12], 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`. If the bound for a type parameter is omitted, the universal reference type (e.g., `Object`) is assumed. A generic vector class in such a language might have the header `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.

Generic types have been an active subject of research for nearly thirty years and have been extensively analyzed in the context of languages with structural subtyping.¹ But, there has been comparatively little research on generic types in nominal type systems [18, 24].

Nominal subtyping has a major impact on the design of OO programming languages. Since class types are embodied as run-time type tags, it is natural to treat types as predicates that can be queried at run-time and to support operations that explicitly depend on these tags such as casts and class membership tests. In this regard, nominally-typed OO languages are closer to dynamically-typed languages than they are to statically-typed languages with structural subtyping.

OO languages with structural subtyping typically erase type information from run-time data representations, so two different classes with the same members have exactly the same run-time representation. Moreover, there are no tags to distinguish the representations of classes with vastly different members, making the language implementation completely reliant on static type checking for type safety. As a result, type-dependent operations such as casts and type predicates are not supported.

1.1.3 Nominal Type Systems Supporting Genericity

Fortunately, stronger type systems for nominally-typed languages have recently begun to emerge. 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* [26]—building on research by Odersky, Wadler, and others on adding genericity to Java [23, 1, 13, 15, 22] using nominal type systems loosely based on F-bounded polymorphism, a structural subtyping discipline that allows bounded type quantification [14]. This extension of the Java language is

¹In the parlance of structural type systems, generic typing is usually referred to as “parametric polymorphism”.

referred to as *Generic Java*.² Martin Odersky supported this effort by developing a well-engineered compiler for a formulation of Generic Java called GJ that supports genericity using *type erasure* (see Section 1.1.4) [13]. In the same year, Igarashi, Pierce, and Wadler developed Featherweight GJ, a small formal model of Generic Java and proved this generic type system to be sound [18]. Two years later, Sun released an “early access” compiler for Generic Java (called JSR-14) based on the GJ compiler. 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.

During the past year, Microsoft announced that the next major release of the .NET platform will include support for generic types based on work by Kennedy and Syme at Microsoft Research [19]. In contrast to Java, generic type information in .NET will be available at run-time.

1.1.4 Type Erasure

Implementation schemes for generic types based on type erasure, such as GJ, treat generic types as “second-class” types that cannot be used in some important contexts. 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 generic type system based on type erasure, generic types cannot be used safely in *type-dependent* operations, i.e., operations that explicitly refer to run-time types, such as casts, `instanceof` operations, and `new` expressions.

1.2 First-Class Generic Types

Although the forthcoming addition of second-class generic types to Java represents a major step forward in its evolution, the restriction of generic types to “second-class” contexts prevents 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 [3]. Even some early access versions of the JSR-14 compiler, which is written in Generic Java, *breached* the GJ type system; the compiler source code used a cast to type `T` where `T` is a type parameter [9, 27].³ 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—provided that the erased types are compatible. Of course, sound generic type checking is lost in the process. Furthermore, there are cases where the compiler generates incorrect code for untypable programs.⁴

²The original specification for Generic Java including syntactic restrictions to support an implementation based on erasure is given in [12]. See [5] for a detailed discussion of the various dialects of Generic Java.

³See the polymorphic method `get` in class `Context` in package `com.sun.tools.javac.v8.util` of version 1.3 of the JSR14 compiler [27]. In version 2.0 of the compiler, Neal Gafter eliminated the breach by changing data representations.

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

An alternate formulation for Generic Java called NEXTGEN [15] eliminates these pathologies by retaining parametric type information at run-time and customizing the code in generic classes where necessary to support (parametric) “type-dependent” operations. NEXTGEN is upward compatible with the forthcoming JSR-14 implementation of Java. A prototype release of the NEXTGEN compiler is available for download at

<http://www.cs.rice.edu/~javaplt/nextgen/doc>

The generic type system proposed for C# in .NET has essentially the same semantics and functionality.

In NEXTGEN, the relationships between generic classes and their instantiations are encoded in a non-generic class hierarchy. A separate interface is generated on demand (via a specialized class loader) for each instantiation of a generic class. An “instantiation class” is also generated to hold all code specific to a particular instantiation. Code common to all instantiations is factored out into a common “base class”. For example, Figure 1 illustrates the encoding of a generic class `Stack`, its parent class `Vector`, and the instantiation `Stack<Integer>`. Recent benchmark results for a prototype compiler for NEXTGEN demonstrate that NEXTGEN-style treatment of generic types does not have any significant performance overhead when compared with either conventional Java or GJ/JSR-14 [5].

The only significant restriction on the use of generic types in NEXTGEN is the prohibition against using naked type variables as superclasses in generic class definitions [15],⁵ i.e., class definitions of the form

```
class C<T> extends T { ... }
```

are prohibited. This restriction 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 research languages, but not in a mainstream programming language other than the crude macro-based implementation in C++.

A simple example of a mixin class definition forbidden in NEXTGEN (and other nominally-typed OO languages supporting genericity) 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`.

To simulate the behavior of this class in Generic Java or Generic C#, we would either have to copy this class definition

⁵The proposed generic extension of C# imposes the same restriction. Both NEXTGEN and Generic C# also exclude naked type variables from appearing in the list of interface types implemented by a class. Such constructions are not semantically sensible because each superinterface for a class specifies a lower bound on the set of 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*.

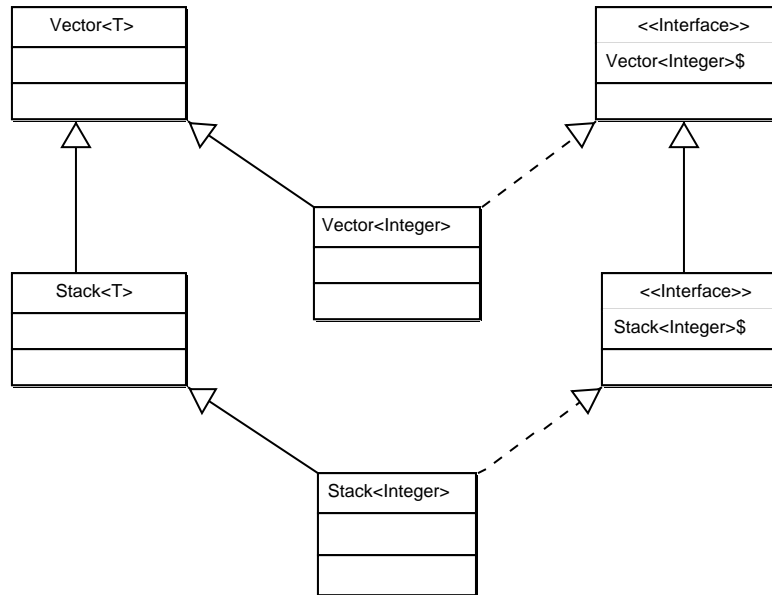


Figure 1: NEXTGEN Representation of A Simple Generic Class

```

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

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

```

Figure 2: A Simple Mixin Class

once for each class we want to extend, or we would have to use *composition* (e.g., the Decorator Pattern [17]) to encode the subclassing relation. The former solution involves undesirable code replication. The latter solution forces source 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 problems.

The remainder of this paper is organized as follows. First, we explain the history of mixins and explain why they should be supported in OO languages. Second, we define the MIXGEN language, an extension of NEXTGEN that supports mixins. Third, we show how to map this language onto an existing and widely used run-time system, the Java Virtual Machine, while maintaining compatibility (including the capacity for subclassing) with existing compiled binaries. Fourth, to demonstrate the soundness of the MIXGEN type system, we present an operational semantics, a type system, and a type soundness theorem for CORE MIXGEN, a subset of MIXGEN that encapsulates the important aspects of the MIXGEN language. Finally, we discuss related work and directions for future research.

2. MIXINS: HISTORY AND MOTIVATION

Nearly 20 years ago, the Lisp object-oriented community invented the term *mixin* [21] 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 [10] and by Ancona and Zucca [7] as functions mapping classes to new subclasses. 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, 11, 16]. 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 [16].
- 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.

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 mixins accommodates the precise typing of non-mixin classes provided by genericity. Hence, code containing mixins can be subjected to the same precise parametric type checking as other generic types, implying that the parametric types declared in mixins are respected during program execution. Previous formalizations of language-level mixins [21, 25, 11, 10, 16, 6] have not incorporated them into a generic typing discipline, and have sacrificed either precise type checking or expressiveness as a result.

3. THE DESIGN OF MIXGEN

Although we are focusing our attention on adding first-class generic types to Java, essentially the same design issues arise in supporting first-class genericity in any strongly-typed object-oriented language with nominal subtyping. We have designed the MIXGEN programming language as an extension of the NEXTGEN formulation of Generic Java. Although the NEXTGEN language defined by Cartwright and Steele explicitly excludes mixins,⁶ its implementation architecture can be gracefully extended to support them. But before we describe the syntax and semantics of MIXGEN, we need to discuss a minor extension to NEXTGEN that addresses a weakness in the design of Generic Java.

3.1 Preliminaries

In Generic Java [15], 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.

One simple solution is to only allow zero-ary constructors to be invoked by `new` operations on a naked type variable and force classes bound to type parameters on which naked `new` operations are performed to include zero-ary constructors. Generic C# follows this approach. But this solution is flawed in two respects. First, it severely limits the utility of supporting `new` operations on naked type parameters. Second, it breaks class encapsulation, since a programmer instantiating a generic class must know which type parameters are used in `new` operations in the class body.

In NEXTGEN, we have adopted a more general solution that accommodates “naked” `new` operations of varying arity. 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 Container<T with {init(); init(Integer);}> {...}
```

requires an instantiation of `T` to support a zero-ary constructor, and a constructor that takes an `Integer`. A `new` operation can only be performed on a naked type variable if the

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

declaration of the type variable includes a `with` clause with a matching constructor.

Several forms of syntactic sugar could be added to the notation for `with` clauses to make it more readable. First, we could replace the constructor method name `init` with the associated bound variable name (e.g., `T` in the example above). Second, enclosing braces, and terminating semicolons, could be omitted from unary lists of signatures. Finally, to keep the size of class headers small, MIXGEN could support an augmented 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 employ these forms of syntactic sugar (except for type bounds) in examples when it is convenient. But we will not discuss them further since they are merely syntactic conventions and they make discussion of the semantic properties of the language more cumbersome.

3.2 MixGen Extensions

MIXGEN is a proper extension 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.

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.

3.3 Cyclic and Infinite Class Hierarchies

Both ordinary Java and Generic Java force the type hierarchy for a program to form a DAG (directed acyclic graph) under the subtyping relation. In Generic Java, all parameterization is erased from class definitions in forming this hierarchy—reducing generic types to the corresponding base (erased) types. This simplification is justified by the following observation: if a collection of generic types has a cycle,

then the collection of corresponding base classes also has a cycle.⁷

In the context of first-class genericity, we must enforce the same constraint. But the analysis of the type hierarchy is more complex. The type hierarchy induced by an arbitrary collection of generic classes and mixins may be cyclic or even infinite. The following examples show some of the complications that can arise.

Example 1.

A class extending a mixin application can extend itself:

```
class C<X with ... > extends X { ... }
class D extends C<D> { ... }
```

Example 2.

A class extending an application of a generic subclass of a mixin can extend itself:

```
class C<X with ... > extends X { ... }
class D<X> extends C<X> { ... }
class E extends D<E> { ... }
```

Example 3.

Classes constructed from mixin applications can form arbitrarily long cycles as the following example demonstrates:

```
class C<X with ... > extends X {...}
class D1 extends C<D0> {...}
...
class Dk-1 extends C<Dk-2> {...}
class D0 extends C<Dk-1> {...}
```

This program creates the cycle:

```
D0 <: C<Dk-1> <: Dk-1 <: C<Dk-2> <: Dk-2 <:
... <: D1 <: C<D0> <: D0
```

Example 4.

Recursion in class definitions involving mixins raises the possibility of infinite class hierarchies. The following program creates an infinite class hierarchy:

```
class C<X with ... > extends D<C<C<X>>> { ... }
class D<X with ... > extends X { ... }
```

consisting of the types

```
D<C<Object>>> <: C<Object> <: D<C<C<Object>>>> <:
C<C<Object>>> <: D<C<C<C<Object>>>>> <: ...
```

To enforce the DAG constraint on type hierarchies, we must analyze the class definitions during compilation and prove that the class hierarchy formed by any program execution forms a DAG. Catching cycles as classes are loaded is unattractive because syntactic malformations should be caught during compilation rather than execution.

One simple way to enforce the DAG constraint in the context of mixins is to prohibit generic classes from parametrically extending mixin classes, *i.e.*, prohibit class definitions such as the definition of `D<X>` in the second example above. In our proof of type soundness, we show that this restriction guarantees that the DAG property holds.

⁷This property is implicitly assumed in [18].

But this restriction is rather severe. Fortunately, we can explicitly check a MIXGEN program to determine if the class definitions can be used to form a non-DAG hierarchy. Before we explain the algorithm for performing explicit checking, we must introduce the concept of a *secondary* mixin. A *secondary* mixin is a generic class that parametrically extends a mixin. More precisely, a generic class `C<T1, ..., TN>` is a secondary mixin if one of the arguments `Ti` is a superclass ancestor of the class. For example, class `D<X>` defined in the second example above is a secondary mixin because the type argument `X` is the superclass of `D<X>`. We can easily identify all of the secondary mixins in a program by first identifying the primitive mixins, then all of the secondary mixins extending a sequence of mixin applications such as

```
class D<X with X()> extends C1<C2<...<Ck<X>...>>
```

where `C1`, `C2`, ..., `Ck` are generic classes that have already been identified as (primitive or secondary) mixins,⁸ and iterating this process until no new secondary mixins are identified.

The DAG checking algorithm simply inspects all of the class headers

```
class C < TypeParameters > extends Type
```

in the program⁹ and applies three reductions to this collection of headers.

- First, it erases all mixin applications and all secondary mixin applications to their superclass arguments.
- Second, it eliminates primary and secondary mixin definitions from the list of class headers.
- Third it erases all type arguments from the remaining headers.

After performing these three reductions, the algorithm inspects the simple types that remain in the headers of the class definitions for cycles, just as conventional Java looks for cycles in class definitions.

The first reduction is based on the observation that mixins simply provide a mechanism for extending a superclass by a subclass (primitive mixin) or a finite chain of subclasses (secondary mixin). These subclass chains can safely be collapsed into their superclasses without changing the DAG status of a class hierarchy graph. From an intuitive perspective, this reduction simply collapses some finite chains in the class hierarchy.

The second reduction is motivated by the fact that mixin headers *in isolation* cannot create a cycle. The remaining headers do not contain any references to mixins because the first reduction eliminated all such references.

The final reduction is justified by the observation that type arguments other than superclass arguments cannot affect the DAG status of the type hierarchy of a program. This is

⁸This notation ignores extra type arguments that are not part of the of superclass chain for `D<X...>`

⁹We are ignoring the implemented interfaces, because they must meet exactly the same constraints as they do for Generic Java.

the same observation used to justify erasing all type parameters from Generic Java programs before checking for cycles in the type hierarchy.

In (first-class) Generic Java, the preceding DAG checking algorithm can be performed incrementally for each Java compilation unit because it is simply an elaboration of the algorithm used to detect cycles in Java class definitions.

To illustrate how this explicit DAG checking algorithm works, consider the four examples at the beginning of this subsection. In the first example, the reduction process leaves the solitary class header `class D extends D`, which is rejected as cyclic. In the second example, the same reduction process leaves the solitary class header `class E extends E`, which is rejected as cyclic. In the third example, the algorithm produces the reduced collection of class headers `class D0 extends Dk-1, ..., class D2 extends D1, class D1 extends D0`, which is rejected as cyclic. In the last example, the algorithm produces the solitary class header `class C extends C`, which is rejected as cyclic.

3.4 Accidental Overriding

Since a mixin

```
class M<T extends B with ... > extends T
```

can be applied to many different superclasses, the mixin code is written with respect to a common type bound `B` which must be satisfied by any superclass argument. Unfortunately, this convention does not prevent unintended interference between a mixin instantiation `M<A>` and its superclass `A`. In particular, `M<A>` may *accidentally override* a method of `A` that is not a member of the bound `B`—breaking the superclass.

Mixins are difficult to type check locally because the set of run-time instantiations of a mixin is not known in general when the mixin is compiled. 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.

Consider the example in Figure 3, which involves accidental overriding. Each class definition is type correct in isolation. Moreover, the method invocation expression

```
new DFactory<C<Object>>().create()
```

has type `C<Object>` given the signatures of generic classes `C`, `D`, `DFactory`, and `E`. Similarly, the expression

```
new E<Object>().typeBreaker(d)
```

has type `Integer`. Nevertheless, the expression

```
new DFactory<C<Object>>().create()
```

generates the mixin instantiation `D<C<Object>>` that accidentally overrides method `m()` of `C` with an inconsistent return type in `D`. When the program is executed, it generates the mixin instantiation `D<C<Object>>`, which is ill-formed because the method return type for `m()` is inconsistent with its definition in the superclass `C`.

```
interface I {
    Object f();
}

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

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

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

class E<T with T()> extends T {
    Integer typeBreaker(C<Object> x) { return x.m(); }
}
...
Object d = new DFactory<C<Object>>().create();
Integer e = new E<Object>().typeBreaker(d);
```

Figure 3: A mixin with accidental overriding

This failure is not detectable from the signatures of the classes involved in the example. In fact, if we change the body of the method `create` in class `DFactory` to `new T()`, the program does not generate an ill-formed mixin instantiation class because accidental overriding does not occur.¹⁰

In languages supporting separate class compilation, such as Java and `C#`, we cannot detect all such accidental overrides during compilation. Since a class can pass its type arguments to other generic classes, a generic class instantiation in the class being compiled can produce an accidental overriding involving a mixin class and an argument class that have already been compiled. The offending mixin class may not even be visible to the compiled class. As a result, only a whole program analysis can detect all accidental overrides. In the context of separate class compilation, the best that we can hope to do is detect inconsistent accidental method overrides at load time. Furthermore, since the error messages could involve package private library classes for which the programmer has neither source code nor documentation, developers could be confronted with syntactic errors involving code that they cannot inspect.¹¹

¹⁰In a Java Virtual Machine, this example would behave differently because the JVM, in contrast to the Java source language, includes the return type of a method in the signature used to match method names. As a result the definitions of `m` would be treated as different methods rather than two definitions of the same method. But programs involving non-hygienic mixins *still cannot be locally type-checked* because methods with matching signatures can have inconsistent attributes such as `final` modifiers, visibility restrictions (*e.g.*, `public`, and `throws` clauses). The crux of the problem is that a whole-program analysis is required to determine what method overrides actually occur and whether they are performed consistently.

¹¹In [4], we argue that the Principle of “Safe Instantiation”, i.e., allow-

3.5 Hygienic Mixins

From the preceding discussion, it is clear that we cannot support the local type checking of mixins (formulated as generic classes) if accidental method overriding occurs. But accidental method overriding is unavoidable if the semantics of mixins is defined by simple syntactic expansion. Fortunately, there is alternative semantics for mixins, developed by Flatt, Krishnamurthi, and Felleisen, that prevents accidental method overriding [16]. In essence, this semantics changes the protocol for method lookup so that accidental method overrides are ignored.

This semantics has not been previously been explicated in the context of mixins as generic classes, so we must develop some new machinery to explain how it works. As a prelude, let us step back and consider how static type checking meshes with dynamic dispatch in nominally-typed object-oriented languages. When a method invocation is statically type-checked, the invoked method's signature is resolved based on the static type T of the receiver. In Java, the byte code generated for the method call refers to this signature and the type T . The actual method code invoked at run-time conforms to this type signature because nominally-typed OO languages generally require overriding methods to have exactly the same signature. Hence, we can model dynamic dispatch as a lookup process that starts in the class hierarchy at the static type of the receiver and walks *down* the path toward the receiver's class to find the method of identical signature that is closest to the receiver's class. Alternatively, we could start the search in the receiver's class and walk *up* the superclass chain in the type hierarchy to find the first method definition with a signature matching the signature specified in the method call. Because overriding methods have matching signatures, these two lookup procedures are semantically equivalent.

In MIXGEN, the class hierarchy is not resolved until run time, so a more sophisticated semantics must be used. To find the method matching signature S in type T , we start with the definition provided by the static type T and search down the hierarchy toward the run-time type for valid overridings of S . If a mixin class on this path does not include S in its superclass bound, then the search stops *because S is hidden in the mixin instantiation and all of its instantiated subclasses*. The resolved method is the last method matching S encountered before reaching a mixin class that hides S . Of course, if no hiding mixin class is encountered in this traversal, the resolved method is the same as it is in conventional Java. For example, in Figure 3, in method

```
Integer typeBreaker(C<Object> x) { return x.m(); }
```

the receiver x has static type $C<Object>$. But in the call

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

the argument to `typeBreaker` (*i.e.*, the return value from `create`) will be an instance of $D<C<Object>>$. So resolution of the method call `m()` will proceed from static type $C<Object>$ toward the run-time type $D<C<Object>>$. In class $C<Object>$, we find method

ing 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.

```
Integer m() { ... }
```

We then proceed to class $D<C<Object>>$. The declared bound on the superclass of mixin D is interface I . Since I does not include method m , we do not look for an overriding method m in $D<C<Object>>$. Method invocation proceeds on the method found in class $C<Object>$.

This approach to method resolution provides the programmer with significant control over which method is invoked in a class with several methods of the same signature (constructed in different mixin applications). By excluding methods from the bound 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 up-casting 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 Java.

It is not obvious that this generalization of method resolution is compatible with the semantics of method resolution embedded in existing run-time systems such as the Java Virtual Machine and the CLR (the virtual machine for C#). In the next section, we will explain how MIXGEN method resolution can be implemented efficiently and compatibly on the JVM, as an extension to NEXTGEN.

4. IMPLEMENTING MIXGEN

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` operations, and `instanceof` tests using as *homogeneous* [23] a representation as possible. Only code for type dependent operations in a generic class is located in 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. In NEXTGEN, all generic method invocations in compiled code use erased type signatures.

In the base class for a generic class, each primitive type-dependent operation is represented by an abstract *snippet* method that is overridden by the appropriate concrete method code in each instantiation class. In generic source code, we refer to each type-dependent primitive operation (such as a cast to a generic type) as a *snippet*.

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 [5]. The only methods in a template class file are snippet methods and constructors. The code for each snippet method is invariant across all possible instantiations except for the references in the constant pool. The 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. For an explanation of this encoding see [15, 5]. Template class files have exactly the same form as conventional class files except

that the constant pools may contain unresolved references to generic type parameters.

During program execution, NEXTGEN relies on a special class loader to recognize references to generic class instantiations and to 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. Unresolved references are represented as deBruijn indices, and characters such as angle brackets and dot delimiters are replaced with special character sequences. For example, a reference to type `List<T>` in class `C<T>` would be mangled to `List$$$L{0}$$$R`.

When a class with a mangled name is first referenced, the class loader reads the template class file (cached in memory if previously accessed) for the specified generic class and patches the unresolved references in the constant pool with the matching mangled names of the type arguments. For example, if class `C<T>` were instantiated as `C<Integer>` then the class loader would patch the type reference `List$$$L{0}$$$R` to `List$$$LInteger$$$R`.

4.1 Basic Implementation Strategy

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 for *mixin* instantiation classes. In 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 4.2 below.

4.2 Enforcing Hygienic Method Invocation

Our hygienic semantics for MIXGEN 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 standard dynamic dispatching mechanism employed in the runtime systems for mainstream OO languages (Java, C#, and C++) which effectively searches up the class hierarchy from the class type of the receiver.¹² However, we can efficiently

¹²The method table in each class object reduces this search to a table lookup.

implement our “downward search” semantics using conventional dynamic dispatch—provided that we use a customized class loader similar to the one already present in NEXTGEN. We will show how we can use such a class loader to systematically rename methods to prevent accidental overriding.

The renaming of methods in mixin classes in the JVM is a subtle issue. The class loader obviously must rename the *new* methods introduced in mixins to avoid accidental overriding. In fact, all *new* methods introduced in classes must be renamed to avoid accidental overriding because a subclass of a mixin instantiation can reintroduce a method that was hidden by the mixin.¹³ Of course, all references to a renamed method must be changed to 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, which can change on each execution of the call site.

We can solve the problem of interface-based method dispatch by adding a forwarding method for each method introduced in a class that implements an interface method.¹⁴ The forwarding method maps `invokeinterface` calls on the (renamed) interface method to the corresponding method introduced in the class.

But there is an additional subtlety here¹⁵ because a class constructed using a series of mixins can implement multiple instantiations of the same generic interface. Each mixin application can only implement one instantiation of a generic interface,¹⁶ but a sequence of mixins can collectively implement several such instantiations (see the program fragment in Figure 4). Both NEXTGEN and MIXGEN use erased method signatures at run-time to maximize code sharing. Essentially all of the code for a generic class is located in a shared based class that is extended by each instantiation of the generic class. If a method in the generic interface has the same internal name (after renaming) in different instantiations supported by the same class, then the forwarding methods for different instantiations of the same generic interface will collide.¹⁷

To eliminate this problem, we can view generic classes as introducing new methods in each instantiation class and distinctly rename the methods in these instantiation classes.

¹³Even 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.

¹⁴If a method introduced in a class implements more than one interface method (which can happen when the same method signature appears in different interfaces), then a forwarding method must be generated for each such interface method.

¹⁵Detected by Martin Odersky in an earlier draft of this paper.

¹⁶This is a restriction of Generic Java that could be eliminated by a more heterogeneous implementation strategy, but it does not appear to be an issue in practice.

¹⁷Collisions can still happen in more heterogeneous implementations without erasure because the signature of a method may not mention the type parameters.

```

interface I<T> { T m(); }

class C<S with S()> extends S implements I<S> {
  S m() { ... }
}

class D implements I<String> {
  String m() { ... }
}

```

Figure 4: Implementing two interface instantiations

This elaboration of our renaming scheme requires a modest revision to the NEXTGEN implementation architecture. The instantiation interfaces previously used only for subtyping purposes (to support generic type casts and `instanceof` tests) must be augmented by the methods introduced in the generic instantiation. The instantiation classes corresponding to the instantiation interfaces must implement these methods by forwarding them to the corresponding methods in the base class.

To invoke the method in these instantiation interfaces, the MIXGEN compiler must generate different byte code for generic method calls than NEXTGEN does. Each method invocation on a receiver of generic type must be compiled to an `invokeinterface` instruction citing the corresponding instantiation interface as the receiver’s static type. If the generic type of the receiver is not ground (because it is within a generic class), then the method call must be implemented as a snippet.

The simplest renaming scheme that satisfies all of above constraints is to prefix the name of every new method introduced in any class (or interface) `C` by a qualifier consisting of the full name of the class `C` followed by a `$` sign. If `C` is generic, the name of `C` in the prefix must be mangled with names introduced in instantiation classes and interfaces.¹⁸ If the mangled name includes free type parameters, then the complete mangled name will not be resolved until an instantiation of the class is loaded. 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 also prefixed by the fully qualified name of the interface. 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.

Notice that the prefixes introduced by the class loader are fully instantiated names so that multiple instantiations of the same interface are distinguished. The code fragment in Figure 4 shows how hygienic mixins can be used to construct a class that implements two different instantiations of the same generic interface.

The class `new C<D>()` implements both `I<D>` and `I<String>`. Our method prefixing scheme generates a forwarding method

¹⁸The new method names in the base class of a generic class (in addition to the instantiation classes) must also be prefixed by the class name.

for `m` in `D` with the prefixed name `I<String>m`.¹⁹ For the generic class `C`, the renaming scheme generates a forwarding method for `m` in the template class (and template interface) for `C` of the form `I<S>m`, where `S` is replaced by the type argument bound to `S` in the instantiation of `C`. The forwarding method for `m` in the template class forwards method calls to the renamed method `m` in the base class for `C`. The NEXTGEN class loader already performs precisely this form of substitution (into strings in the template class constant pool) when it builds generic instantiation classes from template classes because the code in snippet methods refers to the values of type arguments.

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.

4.3 Compiling with clauses

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.

5. CORE MIXGEN

To produce a sound formulation of first-class genericity, we had to identify and resolve many subtle complications in the associated type system. To provide some assurance that our type system is sound, we have developed CORE MIXGEN, a small formal model of the language suitable for proving type soundness. The construction and analysis of this system has been extremely fruitful. Indeed, many of the complications we described in Section 3 were discovered during a formal analysis of CORE MIXGEN. We believe that this analysis is an excellent case study of the value of judicious application of formal methods.

The design of CORE MIXGEN is based on the Featherweight GJ core language for GJ [18]. In the remainder of this pa-

¹⁹Recall that in NEXTGEN and MIXGEN, left and right angle brackets within names are converted to the special character sequences `$$L` and `$$R`, respectively.

CL	:=	class C< \bar{X} extends \bar{N} with $\{\bar{T}\}$ > extends T { \bar{T} \bar{f} ; \bar{K} \bar{M} }
I	:=	init(\bar{T} \bar{x});
K	:=	C(\bar{T} \bar{x}) {super(\bar{e});this. \bar{f} = \bar{e}' };
M	:=	< \bar{X} extends \bar{N} with $\{\bar{T}\}$ > T m(\bar{T} \bar{x}) {return e;}
e	:=	x e.f e.m< \bar{T} >(\bar{e}) new T(\bar{e}) (T)e
T	:=	X N
N	:=	C< \bar{T} >

Table 1: Core Syntax

per, we will refer to these two languages as CMG and FGJ respectively. In developing CMG, we augmented FGJ with the essential features required to support first-class genericity, but nothing more. These features included the following:

- **with** clauses in type parameter declarations. Since FGJ and CMG do not include abstract classes or interfaces, **with** clauses contain only constructor signatures. A **with** clause consists of a sequence of constructor signatures terminated by semicolons and enclosed in braces. For example, `{init(); init(Object x);}` specifies that a type variable contains two constructors: one zero-ary constructor and one constructor that takes a single argument of type `Object`.
- Relaxed restrictions on the use of naked type variables. In CMG (as in MIXGEN), all generic types including type variables are first-class and can appear in casts, **new** operations, and **extends** clauses of class definitions.
- Multiple constructors in a class definition. In FGJ, each class has a standard constructor that takes an initial value for each field as an argument. In CMG, we relax this restriction and permit multiple constructors with arbitrary signatures. This feature allows a class to implement multiple **with** clauses. Without this feature, all classes matching a given **with** clause would have to contain exactly the same collection of fields, crippling the language’s expressiveness.

All CMG programs are valid MIXGEN programs.²⁰ In addition, all FGJ programs are valid CMG programs, modulo two trivial modifications: (1) all type parameter declarations must be annotated with empty **with** clauses, and (2) the arguments in a constructor call must include casts so that they match the parameter types exactly. The former modification is required for the sake of syntactic simplicity; all CMG type parameter declarations must contain **with** clauses. The latter modification is required because CMG

²⁰Some invalid casts that cause errors at run-time in CMG would be detected statically in MIXGEN. The relationship between CMG and MIXGEN parallels the relationship between FGJ and GJ.

allows multiple constructors. In order to keep the resolution of constructor calls simple, an exact match of the static types of constructor arguments to a constructor signature is required. Like FGJ, CMG is a functional language. The body of each method consists of a single **return** statement.

5.1 Syntax

The syntax of CMG is given in Table 1. Throughout all formal rules of the language, the following meta-variables are used over the following domains:

- **d**, **e** range over expressions.
- **I** ranges over constructor signatures.
- **K** ranges over constructors.
- **m**, **M** range over methods.
- **N**, **O**, **P** range over types other than naked type variables.
- **X**, **Y**, **Z** range over naked type variables.
- **R**, **S**, **T**, **U**, **V** range over all types.
- **x** ranges over method parameter names.
- **f** ranges over field names.
- **C**, **D** range over class names.

Following the notation of FGJ, a variable with a horizontal bar above it represents a (possibly empty) sequence of elements in the domain of that variable, with a separator character dependent on context. For example, \bar{T} represents a sequence of types T_0, \dots, T_N , and $\{\bar{T}\}$ represents a sequence of construct signatures in a **with** clause $\{I_0 \dots I_N\}$. As in FGJ, we abuse this notation in select contexts so that, for example, $\bar{T} \bar{f}$ represents a sequence of the structure $T_0 f_0, \dots, T_N f_N$, and $\bar{X} \text{ extends } \bar{S} \text{ with } \{\bar{T}\}$ represents a sequence of type parameter declarations

$$X_0 \text{ extends } S_0 \text{ with } \{\bar{T}\}_0, \dots, X_N \text{ extends } S_N \text{ with } \{\bar{T}\}_N$$

$\frac{\Delta \vdash T <: T \text{ [S-REFLEX]} \quad \frac{\Delta \vdash S <: T \quad \Delta \vdash T <: U}{\Delta \vdash S <: U} \text{ [S-TRANS]}}{\Delta \vdash X <: \Delta(X) \text{ [S-BOUND]}}$
$\frac{CT(C) = \text{class } C < \bar{X} \text{ extends } \bar{N} \text{ with } \{\bar{T}\} > \text{ extends } T \{ \dots \}}{\Delta \vdash C < \bar{S} > <: [\bar{X} \mapsto \bar{S}]T \text{ [S-CLASS]}}$
<hr/> $\text{bound}_{\Delta}(X) = \Delta(X) \quad \text{bound}_{\Delta}(N) = N$

Table 2: Subtyping and Type Bounds

As in FGJ, sequences of field names, method names, and type variables are required to contain no duplicates. Additionally, `this` should not appear as the name of a field or as a method or constructor parameter. The bounds on type variables may contain type parameters declared in the same scope, and they may be mutually recursive.

5.2 Subtyping and Valid Class Tables

Rules for subtyping appear in Table 2. The subtyping relation is represented with the symbol $<: \cdot$. Subtyping is reflexive and transitive, and mixin instantiations are subtypes of the instantiations of their parent types.

A class table CT is a mapping from class names to definitions. A program is a fixed class table with a single expression e . Executing a program consists of evaluating e . As in FGJ, a valid class table must satisfy several constraints: (i) for every C in $\text{dom}(CT)$, $CT(C) = \text{class } C \dots$, (ii) $\text{Object} \notin \text{dom}(CT)$, (iii) every class name appearing in CT is in $\text{dom}(CT)$, (iv) the subtype relation induced by CT is antisymmetric, and (v) the sequence of ancestors of every instantiation type is finite. These last two properties, which are trivial to check in FGJ and Java, are actually quite subtle in CMG, as they are in MIXGEN. CMG avoids both of these complications (cycles and infinite class hierarchies) by placing the following two constraints on class tables:

1. The set of non-mixin class definitions must form a tree rooted at `Object`.
2. No class may be defined to extend a mixin instantiation.

In our proof of type soundness, we show that this restriction is sufficient to prevent both cycles and infinite class hierarchies.

Like FGJ, CMG models class `Object` simply as a tag without a corresponding class definition included in the class table. Class `Object` contains no fields or methods, but it acts as if it contains a single, zero-ary, constructor.

5.3 Type Checking

The typing rules of CMG include three environments:

- A type environment Γ mapping program variables to their static types. Syntactically, these mappings have the form $\bar{x} : \bar{T}$.
- A bounds environment Δ mapping type variables to their upper bounds. Syntactically, these mappings

have the form $\bar{X} < \bar{N}$. The bound of a type variable is always a non-variable type. The bound of a non-variable type N is N .

- A `with` environment Φ mapping type variables to the lower bounds on the set of constructors that they provide. This information is given in `with` clauses and complements the information in the bounds environment. Since CMG does not include abstract classes, constraints on the set of allowed abstract methods do not appear in `with` clauses. Syntactically, `with` environments have the form $\bar{X} \bowtie \{\bar{T}\}$, where $\{\bar{T}\}$ denotes the set of constructor signatures specified in a `with` clause.

When multiple environments are relevant to a typing judgment, they appear together, separated by semicolons. Empty environments are denoted with the symbol \emptyset . In the interest of brevity, we often omit empty environments from typing judgments. For example, the judgment $\emptyset; \emptyset; \emptyset \vdash e \in \text{Object}$ is abbreviated as $\vdash e \in \text{Object}$. 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 safe substitution of all free occurrences of \bar{X} for \bar{Y} in e .

5.4 Well-formed Types and Class Definitions

The rules for well-formed constructs appear in Table 3. A type instantiation is well-formed in environments $\Phi; \Delta$ if all instantiations of type parameters (1) are subtypes of their formal types in Δ , and (2) contain all constructors specified in Φ .²¹ Method definitions are checked for well-formedness in the context of the class definition in which they appear. A method m appearing in class C is well-formed in the body of C if the constituent types are well-formed, the type of the body in Δ is a subtype of the declared type, and m is a valid override of any method of the same name in the static type of the parent of C .

CMG allows multiple constructors in a class. As in FGJ, there is no `null` value in the language, so all constructors are required to assign values to all fields. To avoid pathologies such as the assignment of a field to the (yet to be initialized) value of another field, all expressions in a constructor are typed in an environment binding only the constructor parameters (not the enclosing class fields or `this`).

Class definitions are well-formed if the constituent elements

²¹If a type variable is instantiated with another type variable, Φ is checked to ensure that the sets of specified constructor signatures are compatible.

$\Phi; \Delta \vdash \text{Object ok [WF-OBJECT]} \quad \frac{\mathbf{x} \in \text{dom}(\Delta) \text{ [WF-VAR]}}{\Phi; \Delta \vdash \mathbf{x} \text{ ok}}$
$\frac{CT(\mathbf{C}) = \text{class } \mathbf{C} < \bar{\mathbf{x}} \text{ extends } \bar{\mathbf{N}} \text{ with } \{\bar{\mathbf{T}}\} > \text{ extends } \mathbf{S} \{ \dots \} \\ \Delta \vdash \bar{\mathbf{T}} < : [\bar{\mathbf{x}} \mapsto \bar{\mathbf{T}}] \bar{\mathbf{N}} \quad \Phi \vdash \bar{\mathbf{T}} \text{ includes } [\bar{\mathbf{x}} \mapsto \bar{\mathbf{T}}] \{\bar{\mathbf{T}}\} \quad \Phi; \Delta \vdash \bar{\mathbf{T}} \text{ ok}}{\Phi; \Delta \vdash \mathbf{C} < \bar{\mathbf{T}} > \text{ ok}} \text{ [WF-CLASS]}$
<hr/> $\frac{\begin{array}{l} CT(\mathbf{C}) = \text{class } \mathbf{C} < \bar{\mathbf{x}} \text{ extends } \bar{\mathbf{R}} \text{ with } \{\bar{\mathbf{T}}\} > \text{ extends } \mathbf{S} \{ \bar{\mathbf{T}} \bar{\mathbf{f}}; \bar{\mathbf{K}} \bar{\mathbf{M}} \} \\ \bar{\mathbf{x}} \cap \text{this} = \emptyset \quad \bar{\mathbf{x}} \triangleleft \bar{\mathbf{R}} \vdash \text{override}(\mathbf{S}, < \bar{\mathbf{x}}' \text{ extends } \bar{\mathbf{R}}' \text{ with } \{\bar{\mathbf{T}}'\} > \mathbf{V} \mathbf{m}(\bar{\mathbf{T}}' \bar{\mathbf{x}})) \\ \Phi = \bar{\mathbf{x}} \triangleleft \{\bar{\mathbf{T}}\} + \bar{\mathbf{x}}' \triangleleft \{\bar{\mathbf{T}}'\} \quad \Delta = \bar{\mathbf{x}} \triangleleft \bar{\mathbf{R}} + \bar{\mathbf{x}}' \triangleleft \bar{\mathbf{R}}' \quad \Gamma = \bar{\mathbf{x}} : \bar{\mathbf{T}}' + \text{this} : \mathbf{C} < \bar{\mathbf{x}} > \\ \Phi; \Delta \vdash \bar{\mathbf{R}}' \text{ ok} \quad \Phi; \Delta \vdash \{\bar{\mathbf{T}}'\} \text{ ok} \quad \Phi; \Delta \vdash \mathbf{V} \text{ ok} \quad \Phi; \Delta \vdash \bar{\mathbf{T}}' \text{ ok} \quad \Phi; \Delta; \Gamma \vdash \mathbf{e} \in \mathbf{U} \quad \Delta \vdash \mathbf{U} < : \mathbf{V} \end{array}}{\langle \bar{\mathbf{x}}' \text{ extends } \bar{\mathbf{R}}' \text{ with } \{\bar{\mathbf{T}}'\} > \mathbf{V} \mathbf{m}(\bar{\mathbf{T}}' \bar{\mathbf{x}}) \{ \text{return } \mathbf{e}; \} \text{ ok in } \mathbf{C} < \bar{\mathbf{x}} \text{ extends } \bar{\mathbf{R}} \text{ with } \{\bar{\mathbf{T}}\} >} \text{ [GT-METHOD]}$
$\frac{\begin{array}{l} CT(\mathbf{C}) = \text{class } \mathbf{C} < \bar{\mathbf{x}} \text{ extends } \bar{\mathbf{R}} \text{ with } \{\bar{\mathbf{T}}\} > \text{ extends } \mathbf{S} \{ \bar{\mathbf{T}} \bar{\mathbf{f}}; \bar{\mathbf{K}} \bar{\mathbf{M}} \} \\ \Phi = \bar{\mathbf{x}} \triangleleft \{\bar{\mathbf{T}}\} \quad \Delta = \bar{\mathbf{x}} \triangleleft \bar{\mathbf{R}} \quad \Gamma = \bar{\mathbf{x}} : \bar{\mathbf{V}} \quad \bar{\mathbf{x}} \cap \text{this} = \emptyset \\ \Phi; \Delta \vdash \bar{\mathbf{V}} \text{ ok} \quad \Phi; \Delta; \Gamma \vdash \bar{\mathbf{e}}' \in \bar{\mathbf{U}}' \quad \Phi \vdash \mathbf{S} \text{ includes } \text{init}(\bar{\mathbf{U}}') \quad \Phi; \Delta; \Gamma \vdash \bar{\mathbf{e}} \in \bar{\mathbf{U}} \quad \Delta \vdash \bar{\mathbf{U}} < : \bar{\mathbf{T}} \end{array}}{\mathbf{C}(\bar{\mathbf{V}} \bar{\mathbf{x}}) \{ \text{super}(\bar{\mathbf{e}}'); \text{this}.\bar{\mathbf{f}} = \bar{\mathbf{e}}; \} \text{ ok in } \mathbf{C} < \bar{\mathbf{x}} \text{ extends } \bar{\mathbf{R}} \text{ with } \{\bar{\mathbf{T}}\} >} \text{ [GT-CONSTRUCTOR]}$
$\frac{\begin{array}{l} \bar{\mathbf{K}} \text{ ok in } \mathbf{C} < \bar{\mathbf{x}} \text{ extends } \bar{\mathbf{S}} \text{ with } \{\bar{\mathbf{T}}\} > \quad \bar{\mathbf{M}} \text{ ok in } \mathbf{C} < \bar{\mathbf{x}} \text{ extends } \bar{\mathbf{S}} \text{ with } \{\bar{\mathbf{T}}\} > \\ \Phi = \bar{\mathbf{x}} \triangleleft \{\bar{\mathbf{T}}\} \quad \Delta = \bar{\mathbf{x}} \triangleleft \bar{\mathbf{S}} \quad \Phi; \Delta \vdash \bar{\mathbf{S}} \text{ ok} \quad \Phi; \Delta \vdash \{\bar{\mathbf{T}}\} \text{ ok} \quad \Phi; \Delta \vdash \mathbf{U} \text{ ok} \quad \Phi; \Delta \vdash \bar{\mathbf{T}} \text{ ok} \\ \bar{\mathbf{f}} \cap \text{this} = \emptyset \quad \Delta \vdash \mathbf{C} < \bar{\mathbf{x}} > < : \mathbf{V} \text{ and } \text{fields}(\mathbf{V}) = \bar{\mathbf{T}}' \bar{\mathbf{f}}' \text{ implies } \mathbf{f} \cap \mathbf{f}' = \emptyset \\ \mathbf{K}_i = \mathbf{C}(\bar{\mathbf{T}} \bar{\mathbf{x}}) \{ \dots \} \text{ and } \mathbf{K}_j = \mathbf{C}(\bar{\mathbf{T}} \bar{\mathbf{x}}') \{ \dots \} \text{ implies } i = j \end{array}}{\text{class } \mathbf{C} < \bar{\mathbf{x}} \text{ extends } \bar{\mathbf{S}} \text{ with } \{\bar{\mathbf{T}}\} > \text{ extends } \mathbf{U} \{ \bar{\mathbf{T}} \bar{\mathbf{f}}; \bar{\mathbf{K}} \bar{\mathbf{M}} \} \text{ ok}} \text{ [GT-CLASS]}$

Table 3: Well-formed Constructs

are well-formed, none of the fields known statically to occur in ancestors are shadowed,²² and every constructor has a distinct signature.

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

5.5 Constructors and Methods

The rules for method and constructor inclusion, method typing, method lookup, and valid overrides, appear in Table 4. Method types are determined by searching upward from the static type for the first match. The type of a method includes the class name in which the method occurs, as well as the parameter types and the return type. The included class names are used to annotate receiver expressions in the typing rule for method invocations. As explained in section 5.11, the annotated type of a receiver of an application of method \mathbf{m} is reduced to a more specific type when the more specific type includes \mathbf{m} (with a compatible method signature) in the static type of its parent. Once the annotated type of a receiver is reduced to the most specific type possible, lookup of \mathbf{m} starts at the reduced annotated type.

5.6 Expression Typing

The rules for expression typing are given in Table 5. Naked type variables may occur in new expressions and casts. When checking new operations of naked type, the with environment is checked to ensure that it includes an appropriate constructor signature.

The expression typing rules annotate the receiver expressions for method invocations and field lookups with a static type. In the case of a field lookup, this static type is used to disambiguate the field reference in the presence of accidental shadowing. Although classes are statically prevented from shadowing the known fields of their ancestors, a mixin instantiation may accidentally shadow a field contained in its parent.²³ In the case of method invocations, the receiver is annotated with a static type to allow for a "downward" search of a method definition at run-time, as explained in section 3. Notice that receiver expressions of method invocations are annotated not with their static types *per se*, but instead with the closest supertype of the static type in which the called method is defined. The method found in that supertype is the only method of that name that is statically guaranteed to exist. During computation, the annotated type is reduced whenever possible, modeling the downward search semantics of hygienic mixin method overriding.

²²Notice that this constraint alone does not prevent accidental shadowing in mixin instantiations.

²³For example, a mixin instantiation may extend another instantiation of itself. In that case, *all* fields in the parent instantiation will be shadowed.

$\Phi \vdash \text{Object includes init}()$
$\Phi + X \bowtie \{\bar{T}\} \vdash X \text{ includes } I_k$
$\frac{CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{S} \text{ with } \{\bar{T}\} \text{ extends } T \{ \dots C(\bar{T} \bar{x}) \{ \dots \} \dots \}}{\Phi \vdash C\langle\bar{R}\rangle \text{ includes } [\bar{x} \mapsto \bar{R}] \text{init}(\bar{T})}$
$\frac{CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{N} \text{ with } \{\bar{T}\} \text{ extends } T \{ \bar{T} \bar{f}; \bar{R} \bar{M} \} \quad \langle\bar{Y}\rangle \text{ extends } \bar{N}' \text{ with } \{\bar{T}'\} \text{ extends } T' \text{ m}(\bar{R} \bar{x}) \{ \text{return } e; \} \in \bar{M}}{mtype(m, C\langle\bar{U}\rangle) = C\langle\bar{U}\rangle . [\bar{x} \mapsto \bar{U}] (\langle\bar{Y}\rangle \text{ extends } \bar{N}' \text{ with } \{\bar{T}'\} \text{ extends } T' \text{ m}(\bar{R} \bar{x}))}$
$\frac{CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{S} \text{ with } \{\bar{T}\} \text{ extends } T \{ \bar{T} \bar{f}; \bar{R} \bar{M} \} \quad m \text{ is not defined in } \bar{M}}{mtype(m, C\langle\bar{U}\rangle) = mtype(m, [\bar{x} \mapsto \bar{U}]T)}$
$\frac{CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{S} \text{ with } \{\bar{T}\} \text{ extends } T \{ \bar{T} \bar{f}; \bar{R} \bar{M} \} \quad \langle\bar{Y}\rangle \text{ extends } \bar{S}' \text{ with } \{\bar{T}'\} \text{ extends } T' \text{ m}(\bar{R} \bar{x}) \{ \text{return } e; \} \in \bar{M}}{mbody(m\langle\bar{U}\rangle, C\langle\bar{T}\rangle) = (\bar{x}, [\bar{y} \mapsto \bar{U}] [\bar{x} \mapsto \bar{T}]e)}$
$\frac{CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{S} \text{ with } \{\bar{T}\} \text{ extends } T \{ \bar{T} \bar{f}; \bar{R} \bar{M} \} \quad m \text{ is not defined in } \bar{M}}{mbody(m\langle\bar{V}\rangle, C\langle\bar{U}\rangle) = mbody(m\langle\bar{V}\rangle, [\bar{x} \mapsto \bar{U}]T)}$
$\frac{mtype(m, N) = P . \langle\bar{X}\rangle \text{ extends } \bar{T} \text{ with } \{\bar{T}\} \text{ extends } R \text{ m}(\bar{U} \bar{x}) \text{ implies } \bar{T}', \{\bar{T}'\}, \bar{U}' = [\bar{x} \mapsto \bar{Y}] (\bar{T}, \{\bar{T}\}, \bar{U}) \text{ and } \Delta + \bar{Y} \triangleleft \bar{T}' \vdash R' \triangleleft : [\bar{x} \mapsto \bar{Y}]R}{\Delta \vdash \text{override}(N, \langle\bar{Y}\rangle \text{ extends } \bar{T}' \text{ with } \{\bar{T}'\} \text{ extends } R' \text{ m}(\bar{U}' \bar{x}))}$

Table 4: Constructors and Methods

$\frac{\Phi; \Delta; \Gamma \vdash x \in \Gamma(x) \text{ [GT-VAR]} \quad \frac{\Phi; \Delta \vdash T \text{ ok} \quad \Phi; \Delta; \Gamma \vdash e \in S}{\Phi; \Delta; \Gamma \vdash (T)e \in T} \text{ [GT-CAST]}}{\Phi; \Delta; \Gamma \vdash T \text{ includes init}(\bar{S})}$
$\frac{\Phi; \Delta; \Gamma \vdash \bar{e} \in \bar{S} \quad \Phi; \Delta \vdash T \text{ ok}}{\Phi; \Delta; \Gamma \vdash \text{new } T(\bar{e}) \in T \text{ annotate } [\bar{e} :: \bar{S}]} \text{ [GT-NEW]}$
$\frac{\begin{array}{l} fields(N) = \bar{T} \bar{f} \\ \Phi; \Delta; \Gamma \vdash e \in T \quad \Delta \vdash T \triangleleft : N \\ \Delta \vdash P \triangleleft : N \text{ and } f_i \in fields(P) \text{ implies } P = N \end{array}}{\Phi; \Delta; \Gamma \vdash e.f_i \in T_i \text{ annotate } [e :: N]} \text{ [GT-FIELD]}$
$\frac{\begin{array}{l} \Phi; \Delta \vdash \bar{T} \text{ ok} \quad \Phi; \Delta; \Gamma \vdash e_0 \in T_0 \quad \Phi; \Delta; \Gamma \vdash \bar{e} \in \bar{R} \\ mtype(m, bound_{\Delta}(T_0)) = P . \langle\bar{X}\rangle \text{ extends } \bar{N} \text{ with } \{\bar{T}\} \text{ extends } S \text{ m}(\bar{U} \bar{x}) \\ \Delta \vdash \bar{T} \triangleleft : [\bar{x} \mapsto \bar{T}] \bar{N} \quad \Phi \vdash \bar{T} \text{ includes } [\bar{x} \mapsto \bar{T}] \{\bar{T}\} \quad \Delta \vdash \bar{R} \triangleleft : [\bar{x} \mapsto \bar{T}] \bar{U} \end{array}}{\Phi; \Delta; \Gamma \vdash e_0.m\langle\bar{T}\rangle(\bar{e}) \in [\bar{x} \mapsto \bar{T}]S \text{ annotate } [e_0 \in P]} \text{ [GT-INVK]}$
$\frac{\begin{array}{l} \Delta \vdash \bar{R} \triangleleft : \bar{S} \quad \Phi; \Delta \vdash T \text{ ok} \quad \Phi; \Delta \vdash \bar{S} \text{ ok} \\ \Phi \vdash T \text{ includes init}(\bar{S}) \quad \Phi; \Delta; \Gamma \vdash \bar{e} \in \bar{R} \end{array}}{\Phi; \Delta; \Gamma \vdash \text{new } T(\bar{e} :: \bar{S}) \in T} \text{ [GT-ANN-NEW]}$
$\frac{\begin{array}{l} fields(N) = \bar{T} \bar{f} \quad \Phi; \Delta \vdash N \text{ ok} \\ \Phi; \Delta; \Gamma \vdash e \in T \quad \Delta \vdash T \triangleleft : N \end{array}}{\Phi; \Delta; \Gamma \vdash [e :: N].f_i \in T_i} \text{ [GT-ANN-FIELD]}$
$\frac{\begin{array}{l} \Phi; \Delta \vdash \bar{T} \text{ ok} \quad \Phi; \Delta; \Gamma \vdash e_0 \in T_0 \quad \Phi; \Delta; \Gamma \vdash \bar{e} \in \bar{R} \\ mtype(m, 0) = P . \langle\bar{X}\rangle \text{ extends } \bar{N} \text{ with } \{\bar{T}\} \text{ extends } S \text{ m}(\bar{U} \bar{x}) \\ \Delta \vdash \bar{T} \triangleleft : [\bar{x} \mapsto \bar{T}] \bar{N} \quad \Phi \vdash \bar{T} \text{ includes } [\bar{x} \mapsto \bar{T}] \{\bar{T}\} \quad \Delta \vdash \bar{R} \triangleleft : [\bar{x} \mapsto \bar{T}] \bar{U} \end{array}}{\Phi; \Delta; \Gamma \vdash [e_0 \circ 0].m\langle\bar{T}\rangle(\bar{e}) \in [\bar{x} \mapsto \bar{T}]S} \text{ [GT-ANN-INVK]}$

Table 5: Expression Typing

$fields(\text{Object}) = \bullet$ $CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{S} \text{ with } \{\bar{T}\} \langle \bar{T} \bar{f}; \bar{K} \bar{M} \rangle$ $fields(C\langle\bar{R}\rangle) = [\bar{X} \mapsto \bar{R}] \bar{T} \bar{f}$
$field\text{-vals}(\text{new Object}(), \text{Object}) = \bullet$ $CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{S} \text{ with } \{\bar{T}\} \langle \bar{T} \bar{x} \rangle \{ \dots C(\bar{T} \bar{x}) \{ \text{super}(\bar{e}); \text{this}.\bar{f} = \bar{e}'; \dots \} \}$ $field\text{-vals}(\text{new } C\langle\bar{R}\rangle(\bar{e}'' :: \bar{T}), C\langle\bar{R}\rangle) = [\bar{X} \mapsto \bar{R}] [\bar{x} \mapsto \bar{e}''] \bar{e}'$ $\bar{X} \triangleright \{\bar{T}\}; \bar{X} \triangleleft \bar{S}; \bar{x} : \bar{T} \vdash \bar{e} \in \bar{V} \quad C\langle\bar{R}\rangle \neq N \quad field\text{-vals}(\text{new } [\bar{X} \mapsto \bar{R}] U([\bar{x} \mapsto \bar{e}''] \bar{e} :: \bar{V}), N) = e'''$ $CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{S} \text{ with } \{\bar{T}\} \langle \bar{T} \bar{x} \rangle \{ \dots C(\bar{T} \bar{x}) \{ \text{super}(\bar{e}); \text{this}.\bar{f} = \bar{e}'; \dots \} \}$ $field\text{-vals}(\text{new } C\langle\bar{R}\rangle(\bar{e}'' :: \bar{S}), N) = e'''$

Table 6: Fields and Field Values

Like receiver expressions, the arguments in a `new` expression are annotated with static types. These annotations are used at run-time to determine which constructor is referred to by the `new` operation. Notice that if we had simply used the run-time types of the arguments for constructor resolution, there would be cases in which multiple constructors would match the required signature of a `new` expression.

In order to allow for a subject-reduction theorem over the CMG small-step semantics, it is necessary to provide separate typing rules for annotated field lookup and method invocation expressions. Notice that it would not suffice to simply ignore annotations during typing, since accidental shadowing and overriding would cause the method and field types determined by the typing rules to change during computation. Just as the type annotations play a crucial role in preserving information in the computation rules, they must play an analogous role in typing expressions during computation.

5.7 Stupid Casts

In FGJ, “stupid” casts (the casting of an expression to an incompatible type), were identified as a complication with type soundness [18]. 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 CMG, this issue does not arise, because CMG 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 on a mixin. 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 bound of a mixin instantiation is incompatible with the bound of the type being cast to. Therefore, for the sake of brevity, we simply allow all casts to pass type checking.

5.8 Explicit Polymorphism

Like FGJ, CMG requires explicit polymorphism on parametric methods. Since MIXGEN allows explicit polymorphism, this requirement does not negate the property that all CMG programs are valid MIXGEN programs.

5.9 Fields and Field Values

The rules for the retrieval of the field names and values of an object (used by the typing and computation rules on field lookup) are given in Table 6. The presence of multiple constructors for a class, where constructor signatures do not directly match the field types of a class, makes field value lookup more complex than in FGJ. It is important that a `new` expression is matched to the constructor of the appropriate signature. As a result, unlike FGJ, the mapping *fields* only retrieves those fields directly defined in a class definition. Additionally, a mapping *field-vals* is needed to find the field values of a given object. A static type is passed to *field-vals* to allow for field disambiguation in the presence of accidental shadowing.

5.10 Constructor Call Resolution

In order to avoid the complication of matching multiple constructors, CMG 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.

5.11 Computation

The CMG computation rules are defined in Table 7. As in FGJ, computation is specified via a small-step semantics. Because the static type of a receiver is used to resolve method applications and field lookups, static types must be preserved during computation as annotations on receiver expressions. When computing the application of a method, the appropriate method body is found according to the mapping *mbody*. The application is then reduced to the body of the method, substituting all parameters with their instantiations, and `this` with the receiver. Because it is important that a method application is not reduced until the most specific matching type annotation of the receiver is found, two separate forms are used for type annotations. The original type annotation marks the receiver with an annotation of the form $\in T$. This form of annotation is kept until no further reduction of the static type is possible. At that point, the form of the annotation is switched to $:: T$. Because the computation rules dictate that methods can be applied only on receivers whose annotations are of the latter form, we’re ensured that no further reduction is possible when a method is applied. The symbol \circ is used to designate contexts where either form of annotation is applicable.

$\frac{}{mbody(m\langle\bar{V}\rangle, N) = (\bar{x}, e_0)} \quad \text{[GR-INVK]}$	
$\frac{}{[new\ C\langle\bar{S}\rangle(\bar{e} :: P) :: N] . m\langle\bar{V}\rangle(\bar{d}) \rightarrow [\bar{x} \mapsto \bar{d}][this \mapsto new\ C\langle\bar{S}\rangle(\bar{e} :: P)]_{e_0}} \quad \text{[GR-INV-SUB]}$	
$\frac{CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{S} \text{ with } \{\bar{I}\} \text{ extends } T \{...\} \quad \emptyset; \emptyset \vdash e \in N \quad \emptyset \vdash N <: C\langle\bar{U}\rangle \quad mtype(m, C\langle\bar{U}\rangle) = mtype(m, [\bar{X} \mapsto \bar{U}]T)}{[e \in [\bar{X} \mapsto \bar{U}]T] . m\langle\bar{V}\rangle(\bar{d}) \rightarrow [e \in C\langle\bar{U}\rangle] . m\langle\bar{V}\rangle(\bar{d})} \quad \text{[GR-INV-SUB]}$	
$\frac{CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{S} \text{ with } \{\bar{I}\} \text{ extends } T \{...\} \quad \vdash e \in N \quad \vdash N <: C\langle\bar{U}\rangle \quad mtype(m, C\langle\bar{U}\rangle) \text{ is undefined or } mtype(m, C\langle\bar{U}\rangle) \neq mtype(m, [\bar{X} \mapsto \bar{U}]T)}{[e \in [\bar{X} \mapsto \bar{U}]T] . m\langle\bar{V}\rangle(\bar{d}) \rightarrow [e :: [\bar{X} \mapsto \bar{U}]T] . m\langle\bar{V}\rangle(\bar{d})} \quad \text{[GR-INV-STOP]}$	
$\frac{\emptyset \vdash N <: 0}{(0) new\ N(\bar{e} :: \bar{S}) \rightarrow new\ N(\bar{e} :: \bar{S})} \quad \text{[GR-CAST]}$	$\frac{fields(R) = \bar{T}\ \bar{f} \quad field\text{-vals}(new\ N(\bar{e}), R) = \bar{e}'}{[new\ N(\bar{e}) :: R] . f_i \rightarrow e'_i} \quad \text{[GR-FIELD]}$
$\frac{e_i \rightarrow e'_i}{new\ T(\dots, e_i :: S, \dots) \rightarrow new\ T(\dots, e'_i :: S, \dots)} \quad \text{[GRC-NEW-ARG]}$	
$\frac{e \rightarrow e'}{[e \circ N] . m\langle\bar{V}\rangle(\bar{d}) \rightarrow [e' \circ N] . m\langle\bar{V}\rangle(\bar{d})} \quad \text{[GRC-INV-RECV]}$	
$\frac{e_i \rightarrow e'_i}{[e \circ N] . m\langle\bar{V}\rangle(\dots e_i \dots) \rightarrow [e \circ N] . m\langle\bar{V}\rangle(\dots e'_i \dots)} \quad \text{[GRC-INV-ARG]}$	
$\frac{e \rightarrow e'}{(S)e \rightarrow (S)e'} \quad \text{[GRC-CAST]}$	$\frac{e \rightarrow e'}{[e :: R] . f \rightarrow [e' :: R] . f} \quad \text{[GRC-FIELD]}$

Table 7: Computation

5.12 Type Soundness

A full proof of CMG type soundness is available in an accompanying technical report [2]. In this section, we will present some of the key lemmas in this proof.

Our proof of type soundness differs stylistically from that of FGJ in several respects. The most important difference is a simplified proof of subject reduction that exploits a critical property of all CMG programs. In CMG, all type environments in which the trailing expression of a program is typed are empty. We refer to such expressions as *ground*. Because the evaluation of a program consists of reducing this expression, and because ground expressions always reduce to other ground expressions, it suffices to prove subject reduction solely for ground expressions. We formalize the notion of groundedness with the following definitions.

5.13 Ground Expressions

DEFINITION 1 (GROUND TYPES). A type T is *ground* iff $\vdash T$ ok.

DEFINITION 2 (GROUND EXPRESSIONS). An expression e is *ground* iff $\vdash e \in T$.

5.14 Class Hierarchies

One issue that arises in MIXGEN as a result of first-class genericity is the potential for cyclic and infinite class hierarchies. By using the notion of ground types, we can guarantee the sanity of CMG class hierarchies with the following three lemmas:

LEMMA 1 (COMPACTNESS). For a given ground type $C\langle\bar{N}\rangle$, there is a finite chain of ground types P_0, \dots, P_N s.t. for all i s.t. $1 \leq i \leq N$, $\vdash P_{i-1} <: P_i$ and $P_N = \text{Object}$.

LEMMA 2 (ANTISYMMETRY). For ground types $C\langle\bar{N}\rangle, D\langle\bar{P}\rangle$, if $\vdash C\langle\bar{N}\rangle <: D\langle\bar{P}\rangle$ then either $\not\vdash D\langle\bar{P}\rangle <: C\langle\bar{N}\rangle$ or $C\langle\bar{N}\rangle = D\langle\bar{P}\rangle$.

LEMMA 3 (UNIQUENESS). For a given ground type $C\langle\bar{N}\rangle$, there is exactly one type $P \neq C\langle\bar{N}\rangle$ (i.e., the declared parent instantiation) s.t. both of the following conditions hold:

1. $\vdash C\langle\bar{N}\rangle <: P$
2. If $\vdash C\langle\bar{N}\rangle <: 0$, $C\langle\bar{N}\rangle \neq 0$, and $\vdash 0 <: P$ then $0 = P$.

5.15 Preservation Under Subject Reduction

With these lemmas in hand, we are now in a position to establish a subject reduction theorem.

THEOREM 1 (SUBJECT REDUCTION). For ground expression e , and ground type T , if $\vdash e \in T$ and $e \rightarrow e'$ then $\vdash e' \in S$ where $\vdash S <: T$.

5.16 Statement of Type Soundness

From the theorems established above (as well as a progress theorem included in [2]), we conclude with a statement of type soundness for CMG. First, we need the following definitions:

DEFINITION 3 (VALUE). A ground expression e is a *value* iff e is of the form $\text{new } C\langle T \rangle(\bar{e})$ where all \bar{e} are values.

DEFINITION 4 (BAD CAST). A ground expression e is a *bad cast* iff e is of the form $(T)e'$ where $\vdash e' \in S$ and $\not\vdash S <: T$.

Notice that bad casts include both “stupid casts” (in the parlance of FGJ) and invalid upcasts.

Now let $\xrightarrow{*}$ be the reflexive transitive closure of the reduction relation \rightarrow . Then we can state type soundness for CMG as follows:

THEOREM 2 (TYPE SOUNDNESS). For program (CT, e) s.t. $\vdash e \in T$, evaluation of (CT, e) yields one of the following results:

1. $e \xrightarrow{*} v$ where v is a value of type S and $\vdash S <: T$.
2. $e \xrightarrow{*} e'$ where e' contains a bad cast,
3. Evaluation never terminates, i.e., for every e' s.t. $e \xrightarrow{*} e'$ there exists e'' s.t. $e' \rightarrow e''$.

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, they presumably were proposing non-hygienic mixins. In that 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.

More recently, two other practical proposals for adding mixins to Java have been published, namely JAM [6] and Jiazzi [20], but they do not accommodate generic types. JAM is an extension of Java 1.0 developed by Ancona, Lagorio, 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 a theoretical framework for “mixin modules” described in [8]. This framework is not hygienic because it provides no mechanism for altering the “view” of a mixin class based on context.

Since the JAM type system lacks the expressiveness of generic types, it must severely restrict the use of `this` within the body of a mixin. In particular, `this` cannot be passed as argument to a method. 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 with mixins.

Jiazzi[20] is a sophisticated 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 rather than a programming language, it is not directly comparable to MIXGEN. But Jiazzi supports the definition of mixins that can only be applied at component linking time. In a component, the superclass of a class may be unresolved.²⁴ Since mixins can only be instantiated in the meta-language used to wire components together, Jiazzi does not address the same language design issues as MIXGEN. Since the Jiazzi type system does not support generic types, mixins cannot be assigned precise generic signatures, but this is not an issue in a component system designed for conventional (non-generic) Java. Jiazzi mixins must be hygienic because components only expose selected public methods. In the absence of hygiene, component composition would break component encapsulation. Jiazzi supports mixin hygiene and the static typing of mixins by performing a whole-program analysis on a program composition, systematically renaming methods to avoid accidental overriding.

Hygienic mixins were originally developed by Flatt, Krishnamurthi, and Felleisen in a toy language loosely based on Java called MIXEDJAVA [16]. MIXEDJAVA does not include generic types; mixins are formulated as a separate language construct. Because all mixin instantiations in MIXEDJAVA can be determined statically, sound type checking does not require a hygienic semantics. Flatt *et al* developed a hygienic semantics to support class encapsulation. 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. As a result, there is a fairly high penalty for hygiene in MIXEDJAVA: the size of every reference value is doubled. An object o has type T iff T 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 MIXGEN will not generally preserve the meaning of the original program. Consider the following MIXGEN code fragment:

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

²⁴To appease the Java compiler, which will not compile a class with an undefined superclass, Jiazzi produces a stub class for each such unresolved reference.

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

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

has no effect, because m is *overridden* in each application of the mixin C in $C\langle C\langle A \rangle \rangle$. 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 $C\langle C\langle A \rangle \rangle$ implying that the version of n introduced in $C\langle C\langle A \rangle \rangle$ will be invoked.

In contrast, the corresponding MIXEDJAVA program embeds the type tag $C\langle A \rangle$ 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 $C\langle A \rangle$ and invoke the version of n in $C\langle A \rangle$. This difference reflects the 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, consider the following mixin, written in MIXGEN:

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

If we tried to define a corresponding mixin in MIXEDJAVA, we'd be unable to name the type T or $M\langle T \rangle$ in the body of M . 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.

7. FURTHER RESEARCH

Adding first-class genericity to a nominally-typed object-oriented language 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 existing run-time systems such as the JVM. We believe that these results provide a convincing argument for adding first-class genericity to the next generation of nominally-typed object-oriented languages.

Adding support for full first-class genericity to NEXTGEN introduces more forwarding methods and interface-based method

dispatches into the implementation, which could conceivably add some overhead to programs that use generic types. We do not believe that this overhead is likely to be significant in practice—even in programs that make heavy use of generics—because the extra forwarding methods are typically inlined by JIT compilers. To verify this, we are in the process of implementing a production compiler for MIXGEN so that we can demonstrate its efficiency on a collection of representative benchmarks.

First-class genericity appears particularly appropriate as an extension of C# because the CLR (the .NET virtual machine) supports the notion of *new* methods which do not override any methods in the parent class. As a result, the virtual machine has built-in support for the renaming required to prevent accidental overriding in the methods introduced by a mixin.

8. REFERENCES

- [1] O. Agesen, S. Freund, and J. Mitchell. Adding type parameterization to the Java language. In *OOPSLA*, 1997.
- [2] E. Allen, J. Bannet, and R. Cartwright. Mixins in Generic Java are sound. Technical report, Rice University, 2002.
- [3] E. Allen and R. Cartwright. The case for run-time types in Generic Java. In *Principles and Practice of Programming in Java*, 2002.
- [4] E. Allen and R. Cartwright. Safe instantiation in Generic Java. In *Technical Report, Computer Science Department, Rice University*, 2003. Available at <http://www.cs.rice.edu/CS/PLT/Publications>.
- [5] E. Allen, R. Cartwright, and B. Stoler. Efficient implementation of run-time generic types for Java. In *IFIP WG2.1 Working Conference on Generic Programming*, 2002.
- [6] D. Ancona, G. Lagorio, and E. Zucca. JAM—a smooth extension of Java with mixins. In *ECOOP*, 2000.
- [7] D. Ancona and E. Zucca. A theory of mixin modules: Basic and derived operators. In *Mathematical Structures in Computer Science*, pages 8(4):401–446, 1998.
- [8] D. Ancona and E. Zucca. A theory of mixin modules: algebraic laws and reduction semantics. *Mathematical Structures in Computer Science*, 12(6):701–737, 2002.
- [9] J. Bloch and N. Gafter. Personal communication, 2002.
- [10] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity, and Multiple Inheritance*. PhD thesis, University of Utah, 1992.
- [11] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA*, 1990.
- [12] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. *GJ Specification*, 1998.

- [13] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, 1998.
- [14] L. Cardelli, S. Martini, J. Mitchell, and A. Scedrov. An extension of system f with subtyping. *Information and Computation* 109(1-2), pages 4–56, 1994.
- [15] R. Cartwright and G. Steele. Compatible genericity with run-time types for the Java programming language. In *OOPSLA*, 1998.
- [16] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer’s reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*. Springer-Verlag, 2000.
- [17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [18] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, 1999.
- [19] A. Kennedy and D. Syme. Design and implementation of generics for the .net common language runtime. In *PLDI*, 2001.
- [20] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *OOPSLA*, 2001.
- [21] D. Moon. Object-oriented programming with Flavors. In *OOPSLA*, 1986.
- [22] A. Myers, J. Bank, and B. Liskov. Parameterized types for Java. In *POPL*, 1997.
- [23] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *POPL*, 1997.
- [24] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [25] A. Snyder. CommonObjects: An overview. In *SIGPLAN Workshop on Object-oriented Programming*, 1986.
- [26] Sun Microsystems, Inc. JSR 14: Add generic types to the Java Programming Language, 2001.
- [27] Sun Microsystems, Inc. JSR-14 v1.3 prototype compiler source code, 2003. Available at <http://www.jcp.org/en/jsr/detail?id=014>.