

Mixins in Generic Java are Sound

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

Rice University
6100 S. Main St.
Houston TX 77005

January 2, 2003

Abstract

This technical report presents a type soundness proof for CORE MIXGEN, a small formal language designed for studying the addition of first-class genericity to Java. CORE MIXGEN captures the most intricate aspects of the MIXGEN programming language, an efficient extension of Java, proposed by Allen, Bannet, and Cartwright, that adds first class genericity while maintaining full compatibility with the existing JVM [4]. We begin by reviewing the semantics of CORE MIXGEN, and proceed by establishing several key lemmas. Finally, we conclude by establishing preservation and progress theorems. To our knowledge, this proof is the first type soundness result for a precisely typed, object-oriented programming language with mixins.

1 INTRODUCTION

The MIXGEN programming language is an efficient extension of Java that adds first class genericity while maintaining full compatibility with the existing JVM. We have established that the MIXGEN language design constitutes a feasible extension of Java, by describing how to implement it efficiently on top of the JVM, in [4]. Nevertheless, the semantics of MIXGEN includes many subtle aspects. In particular the semantics of method lookup is quite intricate, and deviates from the conventional Java lookup mechanism in important ways. Because of these subtleties, it is not obvious that the MIXGEN language satisfies type soundness. In this technical report, we argue that the MIXGEN design is sound by establishing a type soundness result for CORE MIXGEN, a small formal model of MIXGEN that captures the most subtle properties of the full language. Our presentation of this proof assumes knowledge of the MIXGEN language design, as presented in [4]. The presentation of CORE MIXGEN semantics in the proceeding sections is a review of the semantics presented in that work.

2 Core MixGen

The design of CORE MIXGEN was based on that of Featherweight GJ [18]. In the remainder of this paper, we will refer to these two languages as CMG and FGJ respectively. With CMG, we have tried to extend FGJ in just those ways necessary to support first-class genericity. The necessary extensions were as follows:

1. The introduction of **with** clauses in type parameter declarations. As in FGJ, there are no abstract classes or interfaces in CMG, so **with** clauses contain only constructor signatures (no abstract method declarations). A **with** clause consists of a sequence of constructor signatures terminated by semicolons and enclosed in braces. For example, `with {init(); init(Object x);}` specifies that a type variable contains two constructors: one zeroary constructor and one constructor that takes a single argument of type `Object`.
2. The relaxation of 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** expressions, and **extends** clauses of class definitions.
3. The allowance of multiple constructors in a class definition. The parameters to a constructor need not be directly related to the class fields. This feature is important in order to allow a class to satisfy multiple **with** clauses. If, as in FGJ, constructor parameters were required to match all class fields exactly, then every type satisfying a **with** clause would have to contain identical fields, which would severely cripple the language's expressiveness.

All CMG programs are valid MIXGEN programs.¹ In addition, all Featherweight GJ 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 Core MixGen 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.

3 Syntax

The syntax of CORE MIXGEN is provided in Figure 1. Throughout all formal rules of the language, the following meta-variables are used over the following domains:

- **d, e** range over expressions.

¹As explained in section 5.4, some invalid casts that cause errors at run-time in CMG would be detected statically in MIXGEN. The relation of CMG to MIXGEN is analogous to that of FGJ to GJ, except that FGJ programs are not valid GJ programs because FGJ uses only explicit polymorphism in polymorphic methods, which is not supported in GJ [18].

```

CL ::= class C< $\bar{X}$  extends  $\bar{N}$  with  $\{\bar{I}\}$ > 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  $\{\bar{I}\}$ > 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< $\bar{T}$ >

```

Figure 1: Core MixGen Syntax

- 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 $\{\bar{I}\}$ 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} extends \bar{S} with $\{\bar{I}\}$ represents a sequence of

type parameter declarations X_0 extends S_0 with $\{\bar{I}\}_0, \dots, X_N$ extends S_N with $\{\bar{I}\}_N$. 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. As in F-bounded polymorphism, the bounds on type variables may contain type parameters declared in the same scope [13].

4 Subtyping and Valid Class Tables

Rules for subtyping appear in Figure 2. The subtyping relation is represented with the symbol $<: .$ 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: (1) for every C in $dom(CT)$, $CT(C) = \text{class } C\dots$, (2) $\text{Object} \notin dom(CT)$, (3) every class name appearing in CT is in $dom(CT)$, (4) the subtype relation induced by CT is antisymmetric, and (5) 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. The following example induces a trivial cycle in the hierarchy of class instantiations:

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

Although we could devise rules that would reject this simple example, we can add arbitrary levels of indirection to type applications, making it impossible for local static checking to catch everything. For example, consider the following class definitions:

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

Then we have the following cycle:

```
D <: C<F> <: F <: E<D> <: D
```

In addition to cycles, polymorphic recursion raises the possibility of infinite class hierarchies. The following definitions:

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

Induce an infinite, non-cyclic, class hierarchy:

$$\begin{array}{c}
\Delta \vdash T <: T \text{ [S-REFLEX]} \quad \frac{\Delta \vdash S <: T \quad \Delta \vdash T <: U}{\Delta \vdash S <: U} \text{ [S-TRANS]} \\
\Delta \vdash X <: \Delta(X) \text{ [S-BOUND]} \\
\frac{CT(C) = \text{class } C\langle \bar{X} \text{ extends } \bar{N} \text{ with } \{\bar{I}\} \rangle \text{ extends } T \{ \dots \}}{\Delta \vdash C\langle \bar{S} \rangle <: [\bar{X} \mapsto \bar{S}]T} \text{ [S-CLASS]} \\
\hline
bound_{\Delta}(X) = \Delta(X) \quad bound_{\Delta}(N) = N
\end{array}$$

Figure 2: Subtyping and Type Bounds

$D\langle C\langle \text{Object} \rangle \rangle <: C\langle \text{Object} \rangle <: D\langle C\langle C\langle \text{Object} \rangle \rangle \rangle <: C\langle C\langle \text{Object} \rangle \rangle <: D\langle C\langle C\langle C\langle \text{Object} \rangle \rangle \rangle \rangle <: \dots$

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 classes must form a tree rooted at `Object`.
2. No class may extend a mixin instantiation.

In the section on type soundness, we show that this restriction is sufficient to prevent both cycles and infinite class hierarchies.

4.1 Class Object

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, zeroary, constructor.

5 Type Checking

The typing rules of CMG include three environments. First, a type environment Γ , maps program variables to their static types. Syntactically, these mappings are of the form $\bar{x} : \bar{T}$. Like FGJ, CMG typing judgements also require a bounds environment Δ to map type variables to their upper bounds. Syntactically, these mappings are of the form $\bar{X} \triangleleft \bar{N}$. The bound of a type variable is always a non-variable type. The bound of a non-variable type N is N . Finally, a `with` environment Φ is used to store the constructor signature constraints appearing in `with` clauses on type variables. These `with` environments are used to check that a type includes the appropriate constructor when used in a new expression or in a super-constructor call.

Because CMG does not include abstract classes, constraints on the set of allowed abstract methods are not included in `with` clauses. Syntactically, `with` environments are of 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 judgement, they appear together, separated by semicolons. Empty environments are denoted with the symbol \emptyset , but, for brevity, we often omit empty environments from typing judgements, so, 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 .

5.1 Well-formed Types and Class Definitions

The rules for well-formed constructs appear in Figure 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 .² 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 .

Unlike Featherweight GJ, CORE MIXGEN allows multiple constructors in a class, and the arguments to a constructor need not be directly related to the constructor's arguments. This feature is important in order to allow a mixin to satisfy multiple constructor signatures. 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 are well-formed, none of the fields known statically to occur in ancestors are shadowed,³ 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.

5.2 Constructors and Methods

The rules for method and constructor inclusion, method typing, method lookup, and valid overrides, appear in Figure 4. Method types are determined by searching upward from the static type for the first match. The type of the class in which a method occurs is prepended to method types. These prepended classes are used to annotate receiver expressions in the typing rule for method invocations. As explained in the section on computation, the annotated type of a receiver of an application of method m is reduced to a more specific type when the more specific type includes m (with a compatible method signature) in the static type of its

²If a type variable is instantiated with another type variable, Φ is checked to ensure that the sets of specified constructor signatures are compatible.

³Notice that this constraint alone does not prevent accidental shadowing in mixin instantiations.

$\Phi; \Delta \vdash \text{Object ok}[\text{WF-OBJECT}] \quad \frac{X \in \text{dom}(\Delta)}{\Phi; \Delta \vdash X \text{ ok}}[\text{WF-VAR}]$
$\frac{CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{N} \text{ with } \{\bar{I}\}\rangle \text{ extends } S \{...\} \quad \Delta \vdash \bar{T} \langle: [\bar{X} \mapsto \bar{T}]\bar{N} \quad \Phi \vdash \bar{T} \text{ includes } [\bar{X} \mapsto \bar{T}]\{\bar{I}\} \quad \Phi; \Delta \vdash \bar{T} \text{ ok}}{\Phi; \Delta \vdash C\langle\bar{T}\rangle \text{ ok}}[\text{WF-CLASS}]$
<hr style="border: 1px solid black;"/> $\frac{CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{R} \text{ with } \{\bar{I}\}\rangle \text{ extends } S \{\bar{T} \bar{f}; \bar{K} \bar{M}\} \quad \bar{x} \cap \text{this} = \emptyset \quad \bar{X} \triangleleft \bar{R} \vdash \text{override}(S, \langle\bar{X}' \text{ extends } \bar{R}' \text{ with } \{\bar{I}'\}\rangle \vee m(\bar{T}' \bar{x})) \quad \Phi = \bar{X} \bowtie \{\bar{I}\} + \bar{X}' \bowtie \{\bar{I}'\} \quad \Delta = \bar{X} \triangleleft \bar{R} + \bar{X}' \triangleleft \bar{R}' \quad \Gamma = \bar{x} : \bar{T}' + \text{this} : C\langle\bar{X}\rangle \quad \Phi; \Delta \vdash \bar{R}' \text{ ok} \quad \Phi; \Delta \vdash \{\bar{I}'\} \text{ ok} \quad \Phi; \Delta \vdash \bar{V} \text{ ok} \quad \Phi; \Delta \vdash \bar{T}' \text{ ok} \quad \Phi; \Delta; \Gamma \vdash e \in U \quad \Delta \vdash U \langle: V}{\langle\bar{X}' \text{ extends } \bar{R}' \text{ with } \{\bar{I}'\}\rangle \vee m(\bar{T}' \bar{x}) \{ \text{return } e; \} \text{ ok in } C\langle\bar{X}\rangle \text{ extends } \bar{R} \text{ with } \{\bar{I}\}\rangle}[\text{GT-METHOD}]$
$\frac{CT(C) = \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{R} \text{ with } \{\bar{I}\}\rangle \text{ extends } S \{\bar{T} \bar{f}; \bar{K} \bar{M}\} \quad \Phi = \bar{X} \bowtie \{\bar{I}\} \quad \Delta = \bar{X} \triangleleft \bar{R} \quad \Gamma = \bar{x} : \bar{V} \quad \bar{x} \cap \text{this} = \emptyset \quad \Phi; \Delta \vdash \bar{V} \text{ ok} \quad \Phi; \Delta; \Gamma \vdash e' \in \bar{U}' \quad \Phi \vdash S \text{ includes } \text{init}(\bar{U}') \quad \Phi; \Delta; \Gamma \vdash \bar{e} \in \bar{U} \quad \Delta \vdash \bar{U} \langle: \bar{T}}{C(\bar{V} \bar{x}) \{ \text{super}(\bar{e}'); \text{this}.\bar{f} = \bar{e}; \} \text{ ok in } C\langle\bar{X}\rangle \text{ extends } \bar{R} \text{ with } \{\bar{I}\}\rangle}[\text{GT-CONSTRUCTOR}]$
$\frac{\bar{K} \text{ ok in } C\langle\bar{X}\rangle \text{ extends } \bar{S} \text{ with } \{\bar{I}\}\rangle \quad \bar{M} \text{ ok in } C\langle\bar{X}\rangle \text{ extends } \bar{S} \text{ with } \{\bar{I}\}\rangle \quad \Phi = \bar{X} \bowtie \{\bar{I}\} \quad \Delta = \bar{X} \triangleleft \bar{S} \quad \Phi; \Delta \vdash \bar{S} \text{ ok} \quad \Phi; \Delta \vdash \{\bar{I}\} \text{ ok} \quad \Phi; \Delta \vdash U \text{ ok} \quad \Phi; \Delta \vdash \bar{T} \text{ ok} \quad \bar{f} \cap \text{this} = \emptyset \quad \Delta \vdash C\langle\bar{X}\rangle \langle: V \text{ and } \text{fields}(V) = \bar{T}' \bar{f}' \text{ implies } f \cap f' = \emptyset \quad K_i = C(\bar{T} \bar{x}) \{...\} \text{ and } K_j = C(\bar{T} \bar{x}') \{...\} \text{ implies } i = j}{\text{class } C\langle\bar{X}\rangle \text{ extends } \bar{S} \text{ with } \{\bar{I}\}\rangle \text{ extends } U \{\bar{T} \bar{f}; \bar{K} \bar{M}\} \text{ ok}}[\text{GT-CLASS}]$

Figure 3: Well-formed Constructs

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

Figure 4: Constructors and Methods

parent. Once the annotated type of a receiver is reduced to the most specific type possible, lookup of `m` starts at the reduced annotated type.

5.3 Expression Typing

The rules for expression typing are listed in Figure 5. Naked type variables may occur in `new` expressions and casts. When checking `new` expressions 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 of 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.⁴ 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 dictated by the hygienic approach to mixins [4]. 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.

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` expression. 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.

5.4 Stupid Casts

In Featherweight GJ, "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 CORE MIXGEN, this issue does not arise, because CORE

⁴One pathological case of such accidental shadowing occurs when a mixin instantiation extends another instantiation of itself. Then all fields in the parent class are shadowed with fields of incompatible type.

$$\begin{array}{c}
\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]} \\
\\
\frac{\Phi \vdash T \text{ includes init}(\bar{S}) \quad \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}{c} \text{fields}(N) = \bar{T} \bar{f} \\ \Phi; \Delta; \Gamma \vdash e \in T \quad \Delta \vdash T <: N \\ \Delta \vdash P <: N \text{ and } f_i \in \text{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}{c} \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} \\ \text{mtype}(m, \text{bound}_\Delta(T_0)) = P. <\bar{X} \text{ extends } \bar{N} \text{ with } \{\bar{T}\}> S m(\bar{U} \bar{x}) \\ \Delta \vdash \bar{T} <: [\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} <: [\bar{X} \mapsto \bar{T}]\bar{U} \end{array}}{\Phi; \Delta; \Gamma \vdash e_0.m<\bar{T}>(\bar{e}) \in [\bar{X} \mapsto \bar{T}]S \text{ annotate } [e_0 \in P]} \text{ [GT-INVK]} \\
\\
\hline
\\
\frac{\begin{array}{c} \Delta \vdash \bar{R} <: \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}{c} \text{fields}(N) = \bar{T} \bar{f} \quad \Phi; \Delta \vdash N \text{ ok} \\ \Phi; \Delta; \Gamma \vdash e \in T \quad \Delta \vdash T <: N \end{array}}{\Phi; \Delta; \Gamma \vdash [e :: N].f_i \in T_i} \text{ [GT-ANN-FIELD]} \\
\\
\frac{\begin{array}{c} \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} \\ \text{mtype}(m, 0) = P. <\bar{X} \text{ extends } \bar{N} \text{ with } \{\bar{T}\}> S m(\bar{U} \bar{x}) \\ \Delta \vdash \bar{T} <: [\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} <: [\bar{X} \mapsto \bar{T}]\bar{U} \end{array}}{\Phi; \Delta; \Gamma \vdash [e_0 \circ 0].m<\bar{T}>(\bar{e}) \in [\bar{X} \mapsto \bar{T}]S} \text{ [GT-ANN-INVK]}
\end{array}$$

Figure 5: Expression Typing

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

Figure 6: Fields and Field Values

MIXGEN does not check for stupid casts in a program. Because mixin instantiations are not resolved until run-time, rarely is it possible to statically detect a stupid cast 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 bounding interface of a mixin instantiation is incompatible with the bounding interface of the type being cast to. Therefore, for the sake of brevity, we simply allow all casts to pass type checking.

5.5 Explicit Polymorphism

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

5.6 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 provided in Figure 6. The inclusion of multiple constructors for a class, where constructor signatures do not directly match the field types of a class, complicates field value lookup in comparison to Featherweight GJ. It is important that a `new` expression is matched to the constructor of the appropriate signature. 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.

5.7 Constructor Call Resolution

In order to avoid complications with resolving multiple matching constructors, CORE MIXGEN requires that the static types of the arguments to a `new` expression *exactly* match a constructor of the corresponding class. Casts can always be used to ensure that the static types of the arguments satisfy this requirement.

6 Computation

The CORE MIXGEN computation rules are defined in Figure 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.

7 Type Soundness

We now establish a proof of type soundness for CORE MIXGEN. We start by stating several supporting lemmas. In particular, we must establish some lemmas concerning the preservation of properties under variable and type variable substitution. Because computation in CORE MIXGEN, as in Featherweight GJ, consists almost entirely of method application, and because method application consists of substituting variables into the body of a method and reducing it, the preservation of properties under substitution plays a central role in a CMG type soundness theorem.

Our proof of type soundness is quite different from that of FGJ in several respects. Most significantly, we have simplified the proof of subject reduction by taking advantage of a particular property of all CMG programs. Notice that 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.

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

Figure 7: Computation

7.1 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 $\vdash e \in T$.

Lemma 1 (Contained Elements of Ground Expressions are Ground) If an expression e is ground then

1. If e' is a sub-expression of e then e' is ground.
2. If T appears in e then T is ground.

Proof Trivial induction over the derivation of $\vdash e \in T$. This lemma is employed pervasively in what follows.

Lemma 2 (Type Substitution Preserves Constructor Inclusion) For ground types \bar{T} , if $\bar{X} \bowtie \{\bar{T}\} \vdash R$ includes $\text{init}(\bar{S})$ and $\vdash \bar{T}$ includes $[\bar{X} \mapsto \bar{T}]\{\bar{T}\}$ then $\vdash [\bar{X} \mapsto \bar{T}]R$ includes $[\bar{X} \mapsto \bar{T}]\text{init}(\bar{S})$.

Proof Case analysis over the derivation of $\bar{X} \bowtie \{\bar{T}\} \vdash R$ includes $\text{init}(\bar{S})$.

Case $\bar{X} \bowtie \{\bar{T}\} \vdash \text{Object}$ includes $\text{init}()$: Trivial.

Case $\bar{X} \bowtie \{\bar{T}\} \vdash X_i$ includes I_k : We're given that $\vdash \bar{T}$ includes $[\bar{X} \mapsto \bar{T}]\{\bar{T}\}$. But $[\bar{X} \mapsto \bar{T}]X_i = T_i$, finishing the case.

Case $\bar{X} \bowtie \{\bar{T}\} \vdash C\langle\bar{R}\rangle$ includes $[\bar{Y} \mapsto \bar{R}]\text{init}(\bar{S}')$: We must show that $\vdash [\bar{X} \mapsto \bar{T}]C\langle\bar{R}\rangle$ includes $[\bar{X} \mapsto \bar{T}][\bar{Y} \mapsto \bar{R}]\text{init}(\bar{S}')$. Because $[\bar{X} \mapsto \bar{T}]C\langle\bar{R}\rangle = C\langle[\bar{X} \mapsto \bar{T}]\bar{R}\rangle$, we have $\vdash C\langle[\bar{X} \mapsto \bar{T}]\bar{R}\rangle$ includes $[\bar{Y} \mapsto [\bar{X} \mapsto \bar{T}]\bar{R}]\text{init}(\bar{S}')$. But $[\bar{Y} \mapsto [\bar{X} \mapsto \bar{T}]\bar{R}]\text{init}(\bar{S}')$ = $[\bar{X} \mapsto \bar{T}][\bar{Y} \mapsto \bar{R}]\text{init}(\bar{S}')$, finishing the case. \square

Lemma 3 (Type Substitution Preserves Fields) For ground types \bar{S} and non-variable type N , if $\text{fields}(N) = \bar{T} \bar{f}$ then $\text{fields}([\bar{X} \mapsto \bar{S}]N) = [\bar{X} \mapsto \bar{S}]\bar{T} \bar{f}$.

Proof Case analysis over the derivation of $\text{fields}(N) = \bar{T} \bar{f}$.

Case $\text{fields}(\text{Object}) = \bullet$: Trivial.

Case $\text{fields}(C\langle\bar{R}\rangle) = [\bar{Y} \mapsto \bar{R}]\bar{T} \bar{f}$: We must show that $\text{fields}([\bar{X} \mapsto \bar{S}]C\langle\bar{R}\rangle) = [\bar{X} \mapsto \bar{S}][\bar{Y} \mapsto \bar{R}]\bar{T} \bar{f}$. But $[\bar{X} \mapsto \bar{S}]C\langle\bar{R}\rangle = C\langle[\bar{X} \mapsto \bar{S}]\bar{R}\rangle$. Then we have $\text{fields}(C\langle[\bar{X} \mapsto \bar{S}]\bar{R}\rangle) = [\bar{Y} \mapsto [\bar{X} \mapsto \bar{S}]\bar{R}]\bar{T} \bar{f} = [\bar{X} \mapsto \bar{S}][\bar{Y} \mapsto \bar{R}]\bar{T} \bar{f}$. \square

Lemma 4 (Type Substitution Preserves Method Types) For ground types \bar{S} and non-variable type N ,

$$mtype(m, N) = N. \langle \bar{X} \text{ extends } \bar{O} \text{ with } \{\bar{I}\} \rangle T m(\bar{U} \bar{x})$$

implies

$$mtype(m, [\bar{Y} \mapsto \bar{S}]N) = [\bar{Y} \mapsto \bar{S}](N. \langle \bar{X} \text{ extends } \bar{O} \text{ with } \{\bar{I}\} \rangle T m(\bar{U} \bar{x}))$$

Proof Only one of the two rules defining $mtype$ matches the premise

$$mtype(m, N) = N. \langle \bar{X} \text{ extends } \bar{O} \text{ with } \{\bar{I}\} \rangle T m(\bar{U} \bar{x})$$

and this rule applies equally well to the substituted forms. \square

Lemma 5 (Type Substitution Preserves Subtyping) For ground types \bar{U} , if $\bar{X} \triangleleft \bar{N} \vdash S <: T$ and $\vdash \bar{U} <: [\bar{X} \mapsto \bar{U}]\bar{N}$ then $\vdash [\bar{X} \mapsto \bar{U}]S <: [\bar{X} \mapsto \bar{U}]T$.

Proof By structural induction over the derivation of $\bar{X} \triangleleft \bar{N} \vdash S <: T$.

Case S-Reflex: Trivial.

Case S-Trans: Follows immediately from the induction hypothesis.

Case S-Bound: $S = X_i$, $T = N_i$, $\bar{X} \triangleleft \bar{N} \vdash S <: N_i$. Then $[\bar{X} \mapsto \bar{U}]S = U_i$ and $[\bar{X} \mapsto \bar{U}]T = [\bar{X} \mapsto \bar{U}]N_i$. But we're given that $\vdash \bar{U} <: [\bar{X} \mapsto \bar{U}]\bar{N}$, finishing the case.

Case S-Class: Then $S = C \langle \bar{R} \rangle$ where $CT(C) = \text{class } C \langle \bar{Y} \text{ extends } \bar{V} \text{ with } \{\bar{I}\} \rangle \text{ extends } T' \{ \dots \}$ and $[\bar{Y} \mapsto \bar{R}]T' = T$. We must show that $\vdash [\bar{X} \mapsto \bar{U}]C \langle \bar{R} \rangle <: [\bar{X} \mapsto \bar{U}][\bar{Y} \mapsto \bar{R}]T'$. But notice that

$$[\bar{X} \mapsto \bar{U}]C \langle \bar{R} \rangle = C \langle [\bar{X} \mapsto \bar{U}]\bar{R} \rangle = C \langle [\bar{X} \mapsto \bar{U}][\bar{Y} \mapsto \bar{R}]\bar{Y} \rangle = C \langle [\bar{Y} \mapsto [\bar{X} \mapsto \bar{U}]\bar{R}]\bar{Y} \rangle$$

Also, $[\bar{X} \mapsto \bar{U}][\bar{Y} \mapsto \bar{R}]T' = [\bar{Y} \mapsto [\bar{X} \mapsto \bar{U}]\bar{R}]T'$. Then $\vdash C \langle [\bar{X} \mapsto \bar{U}]\bar{R} \rangle <: [\bar{Y} \mapsto [\bar{X} \mapsto \bar{U}]\bar{R}]T'$ by [S-CLASS], finishing the case. \square

Lemma 6 (Type Substitution Preserves Type Okness) For ground types \bar{U} , if $\bar{X} \bowtie \{\bar{I}\}; \bar{X} \triangleleft \bar{N} \vdash S \text{ ok}$, $\vdash \bar{U} <: [\bar{X} \mapsto \bar{U}]\bar{N}$ and $\vdash \bar{N}$ includes $[\bar{X} \mapsto \bar{N}]\{\bar{I}\}$ then $\vdash [\bar{X} \mapsto \bar{U}]S \text{ ok}$.

Proof By structural induction over the derivation of $\bar{X} \bowtie \{\bar{I}\}; \bar{X} \triangleleft \bar{N} \vdash S \text{ ok}$.

Case WF-Object: Trivial.

Case WF-Var: $\bar{X} \bowtie \{\bar{I}\}; \bar{X} \triangleleft \bar{N} \vdash S$ ok Let $S = X_i$. Then $[\bar{X} \mapsto \bar{U}]S = U_i$. But we are given that $\vdash U_i$ ok.

Case WF-Class: $\bar{X} \bowtie \{\bar{I}\}; \bar{X} \triangleleft \bar{N} \vdash C \langle \bar{T} \rangle$ ok. Immediate from Lemmas 2, 5, and the induction hypothesis. \square

Lemma 7 (Supertypes of Ground Types are Ground) *For ground type N , if $\vdash N \prec: P$ then P is ground.*

Proof Structural induction over $\vdash N \prec: P$.

Case S-Reflex: Trivial.

Case S-Trans: Immediate from the induction hypothesis.

Case S-Bound: Impossible since N is ground.

Case S-Class: Let $N = C \langle \bar{R} \rangle$ where $CT(C) = \text{class } C \langle \bar{Y} \text{ extends } \bar{V} \text{ with } \{\bar{I}\} \rangle \text{ extends } T \{ \dots \}$ and $[\bar{Y} \mapsto \bar{R}]T = P$. By [GT-CLASS], $\bar{Y} \bowtie \{\bar{I}\}; \bar{Y} \triangleleft \bar{V} \vdash T$ ok. Then by Lemma 6, $\vdash [\bar{Y} \mapsto \bar{R}]T$ ok. \square

Lemma 8 (Ground Expressions Have Ground Types) *If an annotated expression e is ground and $\vdash e \in T$ then T is ground.*

Proof By structural induction over the derivation of $\vdash e \in T$.

Case GT-Var: Impossible since variables are typed according to their assignment in a (non-empty) type environment.

Case GT-Cast, GT-Ann-New: Immediate from the antecedents in these two rules requiring that $\Phi; \Delta \vdash T$ ok.

Case GT-Ann-Field: $\vdash e = [e :: N].f_i \in T_i$. By Lemma 1, N is ground. Then by Lemma 3, T_i is ground.

Case GT-Ann-Invk: Let $e = [e_0 :: T_0].m \langle \bar{R} \rangle (\bar{e})$, $mtype(m, T_0) = C \langle \bar{U} \rangle . \langle \bar{Y} \text{ extends } \bar{N} \text{ with } \{\bar{I}\} \rangle T' m(\bar{S} \bar{x})$. Let $CT(C) = \text{class } C \langle \bar{X} \text{ extends } \bar{N}' \text{ with } \{\bar{I}'\} \rangle \text{ extends } V \{ \dots \}$. Then $T' = [\bar{X} \mapsto \bar{U}]T''$ where T'' is the return type of m in the original method definition in C . In that case, the type T of expression e is $[\bar{Y} \mapsto \bar{R}]T'$ by [GT-ANN-INVK], so we must show that $\vdash [\bar{Y} \mapsto \bar{R}]T'$ ok. But, by [GT-METHOD], $\bar{X} \bowtie \{\bar{I}'\} + \bar{Y} \bowtie \{\bar{I}\}; \bar{X} \triangleleft \bar{N}' + \bar{Y} \triangleleft \bar{N} \vdash T'$ ok. Then by Lemma 6, we have $\vdash [\bar{X} \mapsto \bar{U}][\bar{Y} \mapsto \bar{R}]T'$ ok, \square

7.2 Class Hierarchies

Now that we have formalized the notion of ground types and expressions, we concern ourselves with the potential for cyclic and infinite class hierarchies. We must ensure that the constraints placed on class tables prevents these hierarchies from forming. The following lemmas do exactly that.

Lemma 9 (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 $P_N = \text{Object}$.*

Proof This condition is required directly on non-mixin instantiations for all well-formed class tables. In the case of mixin instantiations, suppose for a contradiction that there exists a mixin instantiation for which the required condition does not hold. Then the instantiation of its parent must be a mixin instantiation; otherwise the lemma would obviously be satisfied for the parent instantiation and, by [S-CLASS], for N as well. So let the instantiation of the parent class of N be mixin instantiation N' . By reasoning analogous to that showing that N' is a mixin instantiation, the parent instantiation of N' must also be a mixin instantiation N'' . Similarly, the parent instantiation of N'' must be a mixin instantiation, and so on. But since all mixin instantiations syntactically contain their parent instantiations, all of these mixin ancestors of N would be syntactically contained in N , and so the syntactic representation of N would be infinitely long. \otimes Thus, the condition holds for all mixin instantiations as well as non-mixin class instantiations. \square

Lemma 10 (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 $C\langle\bar{N}\rangle = D\langle\bar{P}\rangle$.*

Proof By structural induction on the derivation of $\vdash C\langle\bar{N}\rangle <: D\langle\bar{P}\rangle$.

Case S-Reflex: Then $C\langle\bar{N}\rangle = D\langle\bar{P}\rangle$.

Case S-Class: Then $D\langle\bar{P}\rangle$ is the parent of $C\langle\bar{N}\rangle$. There are two subcases.

Subcase C is not a mixin: Then the constraints on well-formed class tables dictate that $D\langle\bar{P}\rangle$ must not be a mixin instantiation. Therefore, the constraints on the non-mixin class hierarchy also require that $\not\vdash D\langle\bar{P}\rangle <: C\langle\bar{N}\rangle$.

Subcase C is a mixin: Then $D\langle\bar{P}\rangle = N_i$. If D is not a mixin, then it is part of the non-mixin class hierarchy, and so it can't be a subtype of a mixin instantiation. If D is a mixin, then suppose for a contradiction that $\vdash D\langle\bar{P}\rangle <: C\langle\bar{N}\rangle$. Because non-mixins are prevented from extending mixin instantiations, the chains of parent classes from $C\langle\bar{N}\rangle$ to $D\langle\bar{P}\rangle$ and from $D\langle\bar{P}\rangle$ to $C\langle\bar{N}\rangle$ both contain only mixin instantiations. But then each of $C\langle\bar{N}\rangle$ and $D\langle\bar{P}\rangle$ would syntactically contain the other, which is impossible. \otimes So $\not\vdash D\langle\bar{P}\rangle <: C\langle\bar{N}\rangle$.

Case S-Trans: Then there is some T s.t. $\vdash C\langle\bar{N}\rangle <: T$ and $\vdash T <: D\langle\bar{P}\rangle$. If $C\langle\bar{N}\rangle = D\langle\bar{P}\rangle$ we're finished, so assume $C\langle\bar{N}\rangle \neq D\langle\bar{P}\rangle$. By the induction hypothesis, either $C\langle\bar{N}\rangle = T$ or $\not\vdash T <: C\langle\bar{N}\rangle$. But if $C\langle\bar{N}\rangle = T$ then $\vdash C\langle\bar{N}\rangle <: D\langle\bar{P}\rangle$ was already derived as a premise to [S-TRANS], and then by the induction hypothesis, $\not\vdash D\langle\bar{P}\rangle <: C\langle\bar{N}\rangle$. Finally, consider the case that $C\langle\bar{N}\rangle \neq T$. Then $\vdash D\langle\bar{P}\rangle <: C\langle\bar{N}\rangle$ implies $\vdash T <: C\langle\bar{N}\rangle$ which contradicts the induction hypothesis. So, $\not\vdash D\langle\bar{P}\rangle <: C\langle\bar{N}\rangle$. \square

Lemma 11 (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 $0 = P$.*

Proof Let P be the declared parent instantiation of $C\langle\bar{N}\rangle$. Suppose for a contradiction that there were a type $0 \neq P$ s.t. $\vdash C\langle\bar{N}\rangle <: 0$, $C\langle\bar{N}\rangle \neq 0$, and $\vdash 0 <: P$. Then there is some finite derivation of $\vdash C\langle\bar{N}\rangle <: 0$. Consider a shortest derivation of $\vdash C\langle\bar{N}\rangle <: 0$, i.e., a derivation employing no more rule applications than any other derivation. Such a derivation can't conclude with [S-REFLEX] because $C\langle\bar{N}\rangle \neq 0$. Also, it can't conclude with [S-CLASS] because $P \neq 0$. Thus, it must conclude with [S-TRANS]. Then there is some type $0'$ s.t. $\vdash C\langle\bar{N}\rangle <: 0'$ and $\vdash 0' <: 0$. Similarly, a shortest derivation of $\vdash C\langle\bar{N}\rangle <: 0'$ can't conclude with [S-REFLEX]; otherwise our derivation of $\vdash C\langle\bar{N}\rangle <: 0$ is not a shortest derivation. Also, our derivation of $\vdash C\langle\bar{N}\rangle <: 0'$ can't conclude with [S-CLASS]; otherwise $0' = P$ which is impossible by Lemma 10. Thus, a shortest derivation of $\vdash C\langle\bar{N}\rangle <: 0'$ must conclude with [S-TRANS]. Continuing in this fashion, we can show that at each step in our derivation of $\vdash C\langle\bar{N}\rangle <: 0$, the rule [S-TRANS] must be employed, requiring yet another step in the derivation. Thus, no finite length derivation could conclude with $\vdash C\langle\bar{N}\rangle <: 0$, so $\not\vdash C\langle\bar{N}\rangle <: 0$. \otimes . Therefore, type 0 does not exist. \square

7.3 Preservation

We now turn our attention to a proof of preservation of CMG types under subject reduction. We start by establishing three more lemmas that are essential to establishing type preservation for field lookups and method invocation.

Lemma 12 (Type Substitution Preserves Typing) *For annotated ground types \bar{U} , if $\bar{X} \bowtie \{\bar{I}\}$; $\bar{X} \triangleleft \bar{N}$; $\Gamma \vdash e \in S$, $\vdash \bar{U} \triangleleft [\bar{X} \mapsto \bar{U}]\bar{N}$ and $\vdash \bar{U}$ includes $[\bar{X} \mapsto \bar{U}]\{\bar{I}\}$ then $[\bar{X} \mapsto \bar{U}]\Gamma \vdash [\bar{X} \mapsto \bar{U}]e \in [\bar{X} \mapsto \bar{U}]S$.*

Proof By structural induction over $\bar{X} \bowtie \{\bar{I}\}$; $\bar{X} \triangleleft \bar{N}$; $\Gamma \vdash e \in S$.

Case GT-Var: $e = x$, $\Phi; \Delta; \Gamma \vdash e \in \Gamma(x)$. Then $[\bar{X} \mapsto \bar{U}]\Gamma \vdash x \in [\bar{X} \mapsto \bar{U}]\Gamma(x)$.

Case GT-Cast: $e = (T)e'$. By Lemma 6, $\vdash [\bar{X} \mapsto \bar{U}]T$ ok. By the induction hypothesis $[\bar{X} \mapsto \bar{U}]\Gamma \vdash [\bar{X} \mapsto \bar{U}]e'$ ok. Thus, $\Gamma \vdash [\bar{X} \mapsto \bar{U}]e \in [\bar{X} \mapsto \bar{U}]T$ by [GT-CAST].

Case GT-Ann-New, GT-Ann-Field: Trivial. The antecedents of these rules apply to the substituted forms by straightforward application of the induction hypothesis, and supporting substitution lemmas.

Case GT-Ann-Invk: $e = [e_0 \circ 0].m\langle\bar{T}\rangle(\bar{e}) \in [\bar{X} \mapsto \bar{T}]S$ All antecedents in [GT-ANN-INVK] but the substituted method type apply by straightforward application of the induction hypothesis, and supporting substitution lemmas. But it remains to show that

$$mtype(m, [\bar{X} \mapsto \bar{U}]0) = [\bar{X} \mapsto \bar{U}]P.\langle\bar{X} \text{ extends } \bar{N} \text{ with } \{\bar{T}\}\rangle S m(\bar{U} \bar{x})$$

There are two cases:

Subcase $0 = P$: Then by Lemma 4, $mtype(m, [\bar{X} \mapsto \bar{U}]0) = [\bar{X} \mapsto \bar{U}](0.\langle\bar{X} \text{ extends } \bar{N} \text{ with } \{\bar{T}\}\rangle S m(\bar{U} \bar{x}))$.

Subcase $0 \neq P$: Because the annotated type of the receiver in the original invocation expression from which e was reduced was determined by [GT-INVK], it must have matched P . The only reduction of the original expression which could have modified the annotated type is [GR-INV-SUB]. But the antecedents of [GR-INV-SUB] ensure that an annotated type S is reduced to T only if $mtype(m, S) = mtype(m, T)$. Therefore, $mtype(m, 0) = mtype(m, P)$ and the case is finished by Lemma 4. \square

Lemma 13 (Term Substitution Preserves Typing) *For annotated expression e annotated ground expressions \bar{e} , and ground types \bar{T} , if $\bar{x} : \bar{T} \vdash e \in S$ and $\vdash \bar{e} \in \bar{R}$ where $\vdash \bar{R} <: \bar{T}$ then $\vdash [\bar{x} \mapsto \bar{e}]e \in S'$ where $\vdash S' <: S$.*

Proof By structural induction over the derivation of $\bar{x} : \bar{T} \vdash e \in S$.

Case GT-Var: $e = x_i$. Then $\vdash [\bar{x} \mapsto \bar{e}]e = e_i$ so letting $S' = R_i$ finishes the case.

Case GT-Cast: $e = (S)e'$. By the induction hypothesis, $\vdash [\bar{x} \mapsto \bar{e}]e'$ is well-typed, so $\vdash [\bar{x} \mapsto \bar{e}]e \in S$.

Case GT-Ann-New: $e = \text{new } S(\bar{e}' :: \bar{R})$, But $[\bar{x} \mapsto \bar{e}]\text{new } S(\bar{e}' :: \bar{R}) = \text{new } S([\bar{x} \mapsto \bar{e}]\bar{e}' :: \bar{R})$. By the induction hypothesis, $\vdash [\bar{x} \mapsto \bar{e}]\bar{e}' \in \bar{R}'$ where $\vdash \bar{R}' <: \bar{R}$. So, by [GT-ANN-NEW], $\vdash \text{new } S([\bar{x} \mapsto \bar{e}]\bar{e}' :: \bar{R}) \in S$.

Case GT-Ann-Field: $e = [e' :: N].f$. Term substitution has no effect on the annotation N or field f . Also, by the induction hypothesis, $\vdash e' \in N'$ where $\vdash N' <: N$. So by [GT-ANN-FIELD], $\vdash [\bar{x} \mapsto \bar{e}]e \in S$.

Case GT-Ann-Invk: $e = [e_0 \circ V].m\langle\bar{T}\rangle(\bar{e}')$. By the induction hypothesis, $[\bar{x} \mapsto \bar{e}]e_0$ is well-typed, as is $[\bar{x} \mapsto \bar{e}]\bar{e}'$. The other premises of [GT-ANN-INVK] are not affected by term substitution, and the static type of the invocation is determined solely by m and the annotated type of the receiver, neither of which are modified by term substitution. So, by [GT-ANN-INVK], $\vdash [\bar{x} \mapsto \bar{e}]e \in S$. \square

Lemma 14 (Program Expression Groundedness) *If a program computation includes the reduction $e \rightarrow e'$ then e and e' are ground.*

Proof Initial program expressions are constrained by well-formedness to be ground, so it suffices to show that reduction preserves groundedness. We proceed by structural induction over the derivation of $e \rightarrow e'$.

Case GR-Cast: Immediate from Lemma 1.

Case GRC-Cast, GRC-Inv-Arg, GRC-Field, GRC-Inv-Recv, GRC-New-Arg: Immediate from the induction hypothesis.

Case GR-Field: $[\text{new } C\langle\bar{S}\rangle(\bar{e} :: \bar{R}) :: D\langle\bar{T}\rangle].f_i \rightarrow e'$. Then $\text{field-vals}(\text{new } C\langle\bar{S}\rangle(\bar{e} :: \bar{R}), D\langle\bar{T}\rangle) = \bar{e}''$ and $e_i'' = e'$. Because \bar{e}'' is defined in the constructor of class D , the only type variables that may appear in \bar{e}'' are those of class D . Let the type parameters of D be \bar{X} , which are substituted with \bar{T} . Because our initial expression is ground, these \bar{T} are ground by Lemma 1. Analogously, the constructor arguments are ground. So, by Lemmas 12 and 13, e_i'' is ground.

Case GR-Invk: $[\text{new } C\langle\bar{S}\rangle(\bar{e} :: P) :: D\langle\bar{T}\rangle].m\langle\bar{V}\rangle(\bar{d}) \rightarrow [\bar{x} \mapsto \bar{d}][\text{this} \mapsto \text{new } C\langle\bar{S}\rangle(\bar{e} :: \bar{P})]e_0$. Then $\text{mbody}(m\langle\bar{V}\rangle, D\langle\bar{T}\rangle) = (\bar{x}, e_0)$. Let $CT(D) = \text{class } D\langle\bar{X}\rangle \text{ extends } \bar{N} \text{ with } \{\bar{I}\} \text{ extends } U \{...\}$ and $\text{mtype}(m, D\langle\bar{T}\rangle) = \langle\bar{Y} \text{ extends } \bar{N}' \text{ with } \{\bar{I}'\}\rangle U' m(\bar{R} \bar{x})$. Then $e_0 = [\bar{Y} \mapsto \bar{V}][\bar{X} \mapsto \bar{T}]e_0'$ where e_0' is the body of m appearing in the definition of D . By [GT-METHOD], $\bar{X} \bowtie \{\bar{I}\} + \bar{Y} \bowtie \{\bar{N}'\}; \bar{X} \triangleleft \bar{N} + \bar{Y} \triangleleft \bar{N}'; \bar{x} : \bar{R} \vdash e_0' \in U''$ for some U'' s.t. $\bar{X} \triangleleft \bar{N} + \bar{Y} \triangleleft \bar{N}' \vdash U'' <: U'$. The only potential free variables in e_0' are \bar{X} , \bar{Y} , and \bar{x} . But, by Lemma 1, \bar{T} , \bar{D} , and \bar{d} are ground, so by Lemmas 12 and 13, $[\bar{x} \mapsto \bar{d}][\text{this} \mapsto \text{new } C\langle\bar{S}\rangle(\bar{e} :: \bar{P})]e_0$ is ground.

Case GR-Inv-Sub: . All that is reduced in this case is the annotated type of the receiver expression. Let the annotated type of the receiver in e be \mathcal{O} , and the annotated type of the receiver in e' be P . The only premise in [GT-ANN-INVK] that refers to the annotated type of the receiver is that $\text{mtype}(m, \mathcal{O}) = \langle\bar{X} \text{ extends } \bar{T} \text{ with } \{\bar{I}\}\rangle U m(\bar{S} \bar{x})$. But, by [GR-INV-SUB], $\text{mtype}(m, \mathcal{O}) = \text{mtype}(m, P)$. Thus, by [GT-ANN-INVK] e' is ground.

Case GR-Inv-Stop: The only reduction in this case is the form of the annotation of the receiver, which has no effect on static typing. \square

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

Theorem 1 (Preservation of Types Under 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 $\vdash S <: T$.*

Proof By structural induction over the derivation of $e \rightarrow e'$.

Case GR-Cast: $e = (0)\text{new } N(\bar{e} :: \bar{S})$. By [GT-CAST], $\vdash e \in 0$. By [GR-CAST], $\vdash N <: 0$. Finally, by [GT-ANN-NEW], $\vdash \text{new } N(\bar{e} :: \bar{S}) \in N$, which finishes the case.

Case GR-Field: $e = [\text{new } C\langle\bar{U}\rangle(\bar{e}) :: D\langle\bar{V}\rangle] . f_i$. By [GR-FIELD], $e' = \text{field-vals}(\text{new } C\langle\bar{U}\rangle(\bar{e} :: \bar{R}), D\langle\bar{V}\rangle)_i$. Let $\text{fields}(D\langle\bar{V}\rangle) = \bar{S} \bar{f}$. By [GT-ANN-FIELD], $\vdash e \in S_i$. Let

$$\begin{aligned} CT(C) &= \text{class } C\langle\bar{X}\rangle \text{ extends } \bar{V} \text{ with } \overline{\{\bar{I}\}} \text{ extends } U' \{ \dots C(\bar{T} \bar{x}) \{ \dots \} \dots \} \\ CT(D) &= \text{class } D\langle\bar{Y}\rangle \text{ extends } \bar{V}' \text{ with } \overline{\{\bar{I}'\}} \text{ extends } U'' \{ \dots \} \end{aligned}$$

We show by induction over the derivation of $e' = \text{field-vals}(\text{new } C\langle\bar{U}\rangle(\bar{e}), D\langle\bar{V}\rangle)_i$ that $\vdash S <: S_i$. There are two possibilities:

Subcase $C\langle\bar{U}\rangle = D\langle\bar{V}\rangle$: Because we are given that $e \rightarrow e'$, it must be the case that $\text{field-vals}(\text{new } C\langle\bar{U}\rangle(e), C\langle\bar{U}\rangle)_i = [\bar{X} \mapsto \bar{U}][\bar{x} \mapsto \bar{e}]e'_i$ where \bar{e}' are the expressions assigned to the fields of C in the matching constructor. By [GT-CONSTRUCTOR], we know that $\bar{X} \bowtie \overline{\{\bar{I}\}}; \bar{X} \triangleleft \bar{V}; \bar{x} : \bar{T} \vdash e'_i \in S'_i$ where $S_i = [\bar{X} \mapsto \bar{U}]S'_i$. But, because \bar{U} are ground, we know by Lemmas 12 and 13 that $\vdash [\bar{X} \mapsto \bar{U}][\bar{x} \mapsto \bar{e}]e'_i \in [\bar{X} \mapsto \bar{U}]S'_i$.

Subcase $C\langle\bar{U}\rangle \neq D\langle\bar{V}\rangle$: Then $\text{field-vals}(\text{new } C\langle\bar{U}\rangle(e), D\langle\bar{V}\rangle)_i = \text{field-vals}(\text{new } [\bar{X} \mapsto \bar{U}]U'([\bar{x} \mapsto \bar{e}]e''))$ where \bar{e}'' are the arguments passed in the super-constructor call within the matching constructor of C . But by the induction hypothesis, $\vdash \text{field-vals}(\text{new } [\bar{X} \mapsto \bar{U}]U'([\bar{x} \mapsto \bar{e}]e'')) \in S'$ where $\vdash S' <: S_i$, finishing the case.

Case GR-Invk: $e = [\text{new } C\langle\bar{S}\rangle(\bar{e}) :: P] . m\langle\bar{V}\rangle(\bar{d})$, $m\text{body}(m\langle\bar{V}\rangle, P) = (\bar{x}, e_0)$. Let

$$\begin{aligned} m\text{type}(m, P) &= D\langle\bar{R}\rangle . [\bar{X} \mapsto \bar{R}]\bar{Y} \text{ extends } \bar{N} \text{ with } \overline{\{\bar{I}\}} \text{ extends } S \text{ m}(\bar{U} \bar{x}) \\ CT(D) &= \text{class } D\langle\bar{X}\rangle \text{ extends } \bar{T}' \text{ with } \overline{\{\bar{I}'\}} \text{ extends } U' \{ \dots \} \end{aligned}$$

By [GT-ANN-INVK], $\vdash e \in [\bar{Y} \mapsto \bar{V}][\bar{X} \mapsto \bar{R}]S$. Let $\Phi = \bar{X} \bowtie \overline{\{\bar{I}\}} + \bar{Y} \bowtie \overline{\{\bar{I}\}}$, $\Delta = \bar{X} \triangleleft \bar{I}' + \bar{Y} \triangleleft \bar{N}$, and $\Gamma = \bar{x} : \bar{U} + \text{this} : D\langle\bar{R}\rangle$. By [GT-METHOD], $\Phi; \Delta; \Gamma \vdash e_0 \in S'$ where $\Delta \vdash S' <: S$. By Lemma 12, $\Gamma \vdash [\bar{Y} \mapsto \bar{V}][\bar{X} \mapsto \bar{R}]e_0 \in [\bar{Y} \mapsto \bar{V}][\bar{X} \mapsto \bar{R}]S'$ and by Lemma 5, $\vdash [\bar{Y} \mapsto \bar{V}][\bar{X} \mapsto \bar{R}]S' <: [\bar{Y} \mapsto \bar{V}][\bar{X} \mapsto \bar{R}]S$. Also, by [GT-ANN-INVK], $\vdash \bar{e} \in \bar{U}'$ where $\vdash \bar{U}' <: [\bar{Y} \mapsto \bar{V}][\bar{X} \mapsto \bar{R}]\bar{U}$. Then by Lemma 13, $\vdash [\bar{x} \mapsto \bar{e}][\text{this} \mapsto \text{new } C\langle\bar{S}\rangle(\bar{e})][\bar{Y} \mapsto \bar{V}][\bar{X} \mapsto \bar{R}]e_0 \in S''$, where $\vdash S'' <: [\bar{Y} \mapsto \bar{V}][\bar{X} \mapsto \bar{R}]S'$. Finally, by transitivity of subtyping, $\vdash S'' <: [\bar{Y} \mapsto \bar{V}][\bar{X} \mapsto \bar{R}]S'$, finishing the case.

Case GR-Sub: $e = [e'' \in [\bar{X} \mapsto \bar{V}]0] . m\langle\bar{V}\rangle(\bar{d})$. $e' = [e'' \in C\langle\bar{V}\rangle] . m\langle\bar{V}\rangle(\bar{d})$.

Let $m\text{type}(m, [\bar{X} \mapsto \bar{V}]0) = P . \bar{X} \text{ extends } \bar{N} \text{ with } \overline{\{\bar{I}\}} \text{ extends } T \text{ m}(\bar{U} \bar{x})$. By [GT-ANN-INVK], $\vdash e \in T$. But by [GR-SUB], $m\text{type}(m, [\bar{X} \mapsto \bar{V}]T) = m\text{type}(C\langle\bar{V}\rangle)$, finishing the case.

Case GR-Stop: $e = [e'' \in [\bar{X} \mapsto \bar{V}]0] . m\langle\bar{V}\rangle(\bar{d})$. The only reduction in this case is the alteration of the form of the annotation. But since [GT-ANN-INVK] applies to either form of annotation, the type of the expression is preserved.

Case GRC-Cast: $e = (S)e_0$, $e' = (S)e'_0$. By the induction hypothesis, e' is well-typed, so the type of $(S)e'_0$ is S by [GT-CAST].

Case GRC-Field: $e = (e :: N) . f_i$. Because [GT-ANN-FIELD] determines the field type based solely on the annotated static type (which is not altered by [GRC-FIELD]) the type of e' is identical to that of e .

Case GRC-New-Arg: $e = \text{new } T(\bar{e} :: \bar{S})$. Let e_i be the reduced subexpression of e , and let e_i reduce to e'_i in e' . Let $\vdash e_i \in R$. By the induction hypothesis, $\vdash e'_i \in R'$ where $\vdash R' <: R$. Then [GT-ANN-NEW] applies just as well to e' as to e , with the static type preserved.

Case GRC-Inv-Recv, GRC-Inv-Arg: $e = [e_0 \in N] . m \langle \bar{V} \rangle (\bar{d})$. Let e_i be the reduced subexpression in e , and let e_i be reduced to e'_i in e' . In both of these cases, the induction hypothesis ensures that e'_i satisfies the required properties of e_i in [GT-ANN-INVK]. But since the type determined by [GT-ANN-INVK] depends solely on m and N , and neither m nor N is altered by these reductions, the static type is preserved. \square

Notice that the preservation theorem above (as well as the supporting lemmas) establish preservation for annotated terms. But since terms are not annotated until type checking, it is important to establish that the types of the annotated terms match their types before annotation. This property is established with the following two lemmas (the first is merely a small supporting lemma for the second).

Lemma 15 (Class Locations of Method Type Signatures) *For non-variable type \mathbb{O} and environments Φ, Δ where $\Phi; \Delta \vdash \mathbb{O}$ ok, if $mtype(m, N) = \mathbb{O} . \langle \bar{X} \text{ extends } \bar{P} \text{ with } \{\bar{I}\} \rangle T m(\bar{S} \bar{x})$ then for any type \mathbb{O}' s.t. $\Delta \vdash N <: \mathbb{O}' <: \mathbb{O}$, if $mtype(m, N) = mtype(m, \mathbb{O}')$ then $\mathbb{O}' = \mathbb{O}$, i.e., \mathbb{O} is the closest superclass containing m with a matching method signature.*

Proof Trivial induction on the derivation of $mtype(m, N) = \mathbb{O} . \langle \bar{X} \text{ extends } \bar{P} \text{ with } \{\bar{I}\} \rangle T m(\bar{S} \bar{x})$. \square

Theorem 2 (Preservation of Types Under Annotation) *For environments Φ, Δ, Γ ,*

1. *If $\Phi; \Delta; \Gamma \vdash e . f_i \in T$ then $\Phi; \Delta; \Gamma \vdash [e :: N] . f_i \in T$.*
2. *If $\Phi; \Delta; \Gamma \vdash \text{new } R(\bar{e}) \in T$ then $\Phi; \Delta; \Gamma \vdash \text{new } R(\bar{e} :: \bar{N}) \in T$.*
3. *If $\Phi; \Delta; \Gamma \vdash e . m \langle \bar{V} \rangle (\bar{e}) \in T$ then $\Phi; \Delta \Gamma \vdash [e \circ N] . m \langle \bar{V} \rangle (\bar{e}) \in T$.*

Proof We consider each part of the theorem in turn.

1. $\Phi; \Delta; \Gamma \vdash e . f_i \in T$. The only distinction between the antecedents of [GT-FIELD] and [GT-ANN-FIELD] is that the type N in which the accessed field is contained is explicitly determined in [GT-FIELD] by the annotated type of the receiver. But, by Lemma 11, the condition that there is no proper subtype P of N s.t. $fields(P)$ includes f_i ensures that N is unique. Because this unique type is the annotation assigned to the receiver, it is the same type referred to in [GT-ANN-FIELD], so $\Phi; \Delta; \Gamma \vdash [e :: N] . f_i \in T$.

2. $\Phi; \Delta; \Gamma \vdash \text{new } R(\bar{e}) \in T$. Because none of the argument expressions have been reduced, their static types will match the annotated types exactly and $\vdash \bar{e} \in \bar{N}$. The other antecedents of [GT-NEW] match antecedents of [GT-ANN-NEW] exactly.
3. $\Phi; \Delta; \Gamma \vdash e.m\langle\bar{V}\rangle(\bar{e}) \in T$. Again, no reduction has occurred, so by Lemma 15 the annotated type O will match the closest supertype of the bound of the static type T_0 of the receiver that contains m . Then $mtype(m, O) = mtype(m, bound_{\Delta}(T_0))$ and the case is finished by [GT-ANN-INVK].

□

7.4 Progress

We now establish a progress theorem for CMG, ensuring that well-typed programs never get “stuck”. First we need to establish the following lemmas:

Lemma 16 (Field Values) *For non-variable type N , If $fields(N) = \bar{T} \bar{f}$ and $\vdash \text{new } P(e :: R) \in P$ where $\vdash P <: N$ then $field\text{-}vals(\text{new } P(e :: R), \bar{N}) = \bar{e}$ where $|\bar{e}| = |\bar{f}|$.*

Proof Case analysis over the derivation of $fields(N) = \bar{T} \bar{f}$.

Case $N = \text{Object}$: By the constructor inclusion rules, the only valid constructor call on **Object** is to a zeroary constructor. Also, by the rules on subtyping, **Object** is a subtype of only itself. But $fields(\text{Object}) = field\text{-}vals(\text{newObject}(), \text{Object}) = \bullet$.

Case $N = C\langle\bar{R}\rangle$, $fields(N) = [\bar{X} \mapsto \bar{R}]\bar{T} \bar{f}$: We proceed by structural induction on the derivation of $\vdash P <: N$.

Subcase S-Reflex: By [GT-CONSTRUCTOR], every valid constructor in a class must initialize all fields \bar{f} with expressions \bar{e} .

Subcase S-Trans: Follows immediately from the induction hypothesis.

Subcase S-Bound: Impossible since this theorem applies only to non-variable type.

Subcase S-Class: Then N is the instantiated parent class of P . But then $field\text{-}vals(\text{new } P(\bar{e}' :: \bar{S}), N) = field\text{-}vals(\text{new } N(\dots), N)$. But $field\text{-}vals(\text{new } N(\dots), N) = \bar{T} \bar{f}$ by reasoning analogous to case [S-REFLEX] (note that we cannot employ the induction hypothesis directly since the induction is over the derivation of $\vdash P <: N$, not the derivation of $field\text{-}vals$). □

Lemma 17 (Method Bodies) *If $mtype(m, N) = R.\langle\bar{X}\rangle$ extends \bar{P} with $\{\bar{I}\} \triangleright T m(\bar{U} \bar{x})$ and $\vdash \bar{V} <: [\bar{X} \mapsto \bar{V}]\bar{P}$ then there exists some e s.t. $mbody(m\langle\bar{V}\rangle, N) = (\bar{x}, e)$.*

Proof Trivial induction over the derivation of $mtype(m, N) = R.\langle \bar{X} \text{ extends } \bar{P} \text{ with } \{\bar{I}\} \rangle T m(\bar{U} \bar{x})$. \square

Definition 3 (Value) A ground expression e is a **value** iff e is of the form $\text{new } C\langle \bar{T} \rangle(\bar{e})$ 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 $\not\vdash S <: T$.

Let $\xrightarrow{*}$ be the transitive closure of the reduction relation \rightarrow . Then we can state a progress theorem for CMG as follows:

Theorem 3 (Progress) For program (CT, e) s.t. $\vdash e \in R$, if $e \xrightarrow{*} e'$ then either e' is a value, e' contains a bad cast, or there exists e'' s.t. $e' \rightarrow e''$.

Proof Because $e \xrightarrow{*} e'$, we know that e' is ground. We proceed by structural induction over the form of e' .

Case $e' = [\text{new } N(\bar{e} :: \bar{S}) :: P] . f$: We know by [GT-ANN-FIELD] that $fields(P) = \bar{T} \bar{f}$ where $f = f_i$. By Lemma 16, $field\text{-}vals(\text{new } N(e), P) = \bar{e}''$ and $|\bar{e}''| = |\bar{f}|$. Then by [GR-FIELD], $e' \rightarrow e''_i$.

Case $e' = [d :: P] . f$, **d is not a new expression**: By [GT-ANN-INVK], d is well-typed, so by the induction hypothesis, either d is a value, d contains a bad cast, or there exists a d' s.t. $d \rightarrow d'$. But since d is not a new expression, it can't be a value. If d contains a bad cast, then so does e' and we are done. And if $d \rightarrow d'$ then by [GRC-FIELD], $[d :: P] . f \rightarrow [d' :: P] . f$.

Case $e' = [d \circ P] . m\langle \bar{T} \rangle(\bar{e})$, **d is not a new expression**: Analogous to the case above.

Case $e' = [\text{new } N(e) :: P] . m\langle \bar{T} \rangle(\bar{e})$: By [GT-ANN-INVK], $mtype(m, P) = 0.\langle \bar{X} \text{ extends } \bar{N} \text{ with } \{\bar{I}\} \rangle S m(\bar{U} \bar{x})$. Then by Lemma 17, $mbody(m, P) = (x, e'')$, and by [GR-INVK], $e' \rightarrow e''$.

Case $e' = [\text{new } N(e) \in P] . m\langle \bar{T} \rangle(\bar{e})$: By [GT-ANN-INVK] and [GT-ANN-NEW], $\vdash \text{new } N(e) \in N$. By Theorem 1, $\vdash N <: P$. If $mtype(m, N) = mtype(m, P)$ then $e' \rightarrow [\text{new } N(e) \in N] . m\langle \bar{T} \rangle(\bar{e})$ by [GR-INVK-SUB]. Otherwise, $e' \rightarrow [\text{new } N(e) :: P] . m\langle \bar{T} \rangle(\bar{e})$ by [GR-INVK-STOP], finishing the case.

Case $e' = \text{new } N(\bar{e} :: \bar{T})$: Then either e' is a value and we are finished, or there is some e_i that is not a value. Then by the induction hypothesis, either e_i contains a bad cast (and then so does e') or there exists some e'_i s.t. $e_i \rightarrow e'_i$. Then by [GRC-NEW-ARG], $\text{new } N(e :: T) \rightarrow \text{new } N(e_0 :: T_0, \dots, e'_i :: T_i, \dots, e_N :: T_N)$, finishing the case.

Case $e' = (N)e''$: Because e' is well typed, we know there is some P s.t. $\vdash e'' \in P$. If $\not\vdash P <: N$ then e' is a bad cast and we are done. Otherwise, $e' \rightarrow e''$ by [GR-CAST]. \square

7.5 Type Soundness

From the theorems established above, we conclude with the following type soundness theorem for CMG:

Theorem 4 (Type Soundness) *For program (CT, e) s.t. $\vdash e \in \mathbb{T}$, 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. $e' \rightarrow e''$.

Proof Immediate from Theorems 1, 2, and 3. \square

References

- [1] O. Agesen, S. Freund and J. Mitchell. Adding Type Parameterization to the Java Language. In OOPSLA'97.
- [2] D. Ancona and E. Zucca. A Theory of Mixin Modules: Basic and Derived Operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.
- [3] E. Allen, R. Cartwright, B. Stoler. Efficient Implementation of Run-time Generic Types for Java. IFIP WG2.1 Working Conference on Generic Programming, July 2002.
- [4] E. Allen, J. Bannet, R. Cartwright. A First-Class Approach to Genericity. Submitted to ECOOP 2003.
- [5] E. Allen, R. Cartwright. The Case for Run-time Types in Generic Java. *Principles and Practice of Programming in Java*, June 2002.
- [6] D. Ancona, G. Lagorio, E. Zucca. JAM-A Smooth Extension of Java with Mixins. ECOOP 00, LNCS, Springer Verlag, 2000.
- [7] J. Bloch, N. Gafter. Personal communication.
- [8] G. Bracha W. Cook. Mixin-based inheritance. OOPSLA '90, October 1990 .
- [9] R. Cartwright, G. Steele. Compatible genericity with run-time types for the Java programming language. In *OOPSLA '98*, October 1998.
- [10] G. Bracha. The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. Ph.D. dissertation, Dept. of Computer Science, University of Utah 1992.

- [11] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98*, October 1998.
- [12] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. GJ Specification. Online manuscript available at <http://www.cis.unisa.edu.au/pizza/gj/Documents/>, May 1998.
- [13] P. Cuning, W. Cook, W. Hill, W. Olthoff, J. Mitchell. F-bounded quantification for object-oriented programming. In *Proc. of the ACM FPCA*. pp. 273-280. September 1989.
- [14] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass. 1995.
- [15] M. Flatt, S. Krishnamurthi, M. Felleisen. Classes and Mixins. In *POPL 1998*, January 1998.
- [16] M. Flatt, S. Krishnamurthi, M. Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. Formal Syntax and Semantics of Java, volume 1523, June 1999.
- [17] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *POPL 1997*, January 1997, 146–159.
- [18] A. Igarashi, B. Pierce, P. Wadler Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA '99*, November 1999.
- [19] S. McDirmid, M. Flatt, W. Hsieh. Jiazzzi: New Age Components for Old Fashioned Java. In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2001.
- [20] D. Moon. Object-oriented Programming with Flavors. In *OOPSLA '86*, 1986.
- [21] A. Snyder. CommonObjects: An Overview. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, Sigplan Notices 21(10), 19-28, 1986.
- [22] G. Steele. Growing a Language. In *Journal of Higher-Order and Symbolic Computation* (Kluwer) **12** (3), October 1999, 221–236.
- [23] Sun Microsystems, Inc. JSR 14: Add Generic Types To The Java Programming Language. Available at <http://www.jcp.org/jsr/detail/14.jsp>.