Component NextGen: A Sound and Expressive Component Framework for Java [*]

James Sasitorn
Rice University
camus@rice.edu

Robert Cartwright
Rice University
cork@rice.edu

Abstract

Developing a general component system for a statically typed, object-oriented language is a challenging design problem for two reasons. First, mutually recursive references across components are common in object-oriented programs—an issue that has proven troublesome in the context of component systems for functional and procedural languages. Second, inheritance across component boundaries can cause accidental method overrides. Our recent research shows that a component framework can be constructed for a nominally typed object-oriented language supporting *first-class*[1] generic types simply by adding appropriate annotations, syntactic sugar, and component-level type-checking. The fundamental semantic building blocks for constructing, type-checking and manipulating components are provided by the underlying first-class generic type system. To demonstrate the simplicity and utility of this approach we have designed and implemented an extension of Java called Component NEXTGEN (CGEN). CGEN, which is based on the Sun Java 5.0 **javac** compiler, is backwards compatible with existing Java binary code and runs on current Java Virtual Machines.

The primary contribution of this paper is a technical analysis of the subtle design issues involved in building a component framework for a nominally typed object-oriented language supporting first-class generics. In contrast to component systems for structurally typed languages, mutual recursion among components is accommodated in the type system and semantics without incorporating any special machinery. Our analysis includes a presentation of Core CGEN (CCG), a small, core language modeling the CGEN framework. It is based on Featherweight GJ and incorporates some ideas from MIXGEN. CCG adds the essential features to support components, but nothing more. Our discussion includes the type rules and semantics for CCG, as well as a proof of type safety.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects; Polymorphism; Modules, packages; Components

General Terms Design, Language

Keywords first-class generics, Java implementation, custom class loader, modules, components, signatures

1. Introduction

Java has transformed mainstream software development by supporting clean object-oriented design, comprehensive static type checking, safe program execution, and an unprecedented degree of portability. Despite these significant achievements, the Java language has been handicapped as a vehicle for writing large applications during its brief history by two major shortcomings: (i) the lack of a generic type system (classes and methods parameterized by type) and (ii) the absence of a component system for decomposing applications into independent units with statically checked interfaces. The first shortcoming has been partially addressed by the Java standardization process (JSR-14 [17] incorporated in Java 5.0), and more comprehensively by the programming research community (GJ [8], PolyJ[22], NEXTGEN [9], LM [33, 32]) but the issue of how to support a general purpose component system in Java has received relatively little attention.

Components are independent units of compiled code that can be "linked" to form complete programs. They have no independent state, and all of the references that cross unit boundaries must be explicitly identified in a unit's signature and linked together when the units are joined to form programs[12, 16, 30].

[*] This research has been partially supported by the National Science Foundation, the Texas Advanced Technology Program, and Sun Microsystems.

[1] A language supporting "first-class" generic types allows generic types to be used in any context where conventional types can appear. This terminology was established in [1]

Two features of the Java platform are often cited as mechanisms for supporting components: the Java package system and the JavaBeans framework. But neither of these facilities qualifies as a component system suitable for decomposing application programs into independent units of compiled code.

Java packages are simply separate name spaces; they typically contain explicit references to names in other packages (akin to hard-coded file names scattered in shell scripts), violating the principle that all external references must be explicitly identified in a component signature.

The JavaBeans framework is a "wiring standard" including an introspection mechanism for building applications in specific domains. This framework is targeted primarily at building graphical user interfaces in client programs. Like all "wiring standards", it formulates components as objects (rather than classes) and relies on an introspective mechanism (reflection) to support flexible linking with other components—compromising the effectiveness of static type checking and preventing the use of linguistic mechanisms like inheritance across component boundaries.

Developing a general component system for a statically typed, object-oriented language like Java is a challenging problem for two reasons. First, mutually recursive references across components are common in object-oriented programs—an issue that has proven troublesome in the context of component systems for functional and procedural languages. Second, inheritance across component boundaries can produce unexpected results. The name of a method introduced in a class may collide with the name of a `public` or `protected` method in an imported superclass [31, 30], breaking the behavior of the inherited superclass.

Component NEXTGEN (CGEN) avoids accidental method capture by supporting "hygienic" components based on "hygienic" mixins [12, 13, 1] that eliminate accidental method capture by systematically renaming methods. CGEN is an extension of the NEXTGEN language, a generalization of Java 5.0 that efficiently supports first-class generic types[4, 27]. By leveraging first-class generic types in NEXTGEN, the construction of a hygienic component system for CGEN is reduced to adding the appropriate annotations (signatures), syntactic sugar (modules), and the corresponding component-level type checking.

In NEXTGEN, generic types are "first-class" values that can appear in almost any context where conventional types can appear. NEXTGEN supports type casting and `instanceof` operations of parametric type, class constants of naked parametric type (*e.g.*, `T.class`), and `new` operations of naked parametric class and array types, *e.g.*, `new T()`, and `new T[]`.

A prior extension of NEXTGEN called MIXGEN[1] probed the limits of first-class genericity by incorporating *hygienic* mixins in the NEXTGEN implementation architecture. A mixin is a class definition where a type parameter `T` serves as the super class:

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

The hygienic semantics for mixins prevents accidental method capture when a mixin instantiation `C<A>` overrides a method of the superclass `A` that is not a member of the interface `I` bounding `A`.

The CGEN component language is a natural extension of the NEXTGEN generic type system that supports the parameterization of packages by type (creating components) and adds new annotations (package signatures) to the type system. The translation of these new constructs into NEXTGEN code is a local syntactic expansion, implying that the new constructs are merely syntactic sugar—modulo the extra type-checking required to ensure that the sugared syntax expands into well-typed Java code.[2] CGEN does not require a secondary module language or additional translation steps between compilation or execution. CGEN is backwards compatible with existing libraries and executes on current JVMs.

The prototype implementation of CGEN, derived from the Java 5.0 **javac** compiler, produces code comparable in efficiency and size to NEXTGEN—which is not surprising since CGEN uses a similar implementation architecture.

While a prior publication on our work [28] focused on providing a pragmatic overview of CGEN, the primary contribution of this paper is a technical analysis of the design issues involved in building a component framework for a nominally typed object-oriented language supporting first-class generics, like the NEXTGEN formulation of Generic Java.

The remainder of this paper is organized as follows. Section 2 reviews the conceptual foundations of our work and motivates the design of CGEN. Section 3 provides an overview of CGEN. Section 4 discusses the technical complications present in CGEN. Section 5.1 discusses Core CGEN (CCG), a subset of CGEN that encapsulates the core elements of CGEN. Finally, Section 6 discusses related work on component systems for Java and similar languages. Appendix A provides a more extensive discussion of CCG including an operational semantics, type inference system, and an outline of type soundness.

2. Background and Motivation

Before we discuss how components are defined in CGEN, we need to survey the technical and conceptual foundations for our work.

2.1 Nominal Typing

Mainstream object-oriented languages, namely Java, C#, and C++, rely on the *nominal* typing rather than the *structural typing* of objects, which has been the focus of most of

[2] Our actual implementation optimizes the naive translation based purely on syntactic sugar.

the technical research on type systems. In object-oriented languages with nominal typing, a class A is a subtype of a class B iff A explicitly inherits from B.[3] The programmer explicitly determines the structure of the type hierarchy.

In languages with structural typing, a class A is a subtype of a class B iff (i) A includes all of the member names in B and (ii) the type of each member in A is a subtype of the corresponding member in B. There is no relationship between subtyping and code inheritance because subtyping between classes depends only on the signatures of the members of types—not on their inheritance relationships or on subtyping relationships explicitly declared by the programmer. For a more detailed comparison of the structural and nominal typing of objects see [1].

The distinction between structural and nominal typing means that we cannot directly apply technical results derived for structural type systems to Java or putative extensions of Java like CGEN.

2.2 Independent Components

Software developers generally recognize the value of building software from reusable components as much as is practicable. But the proper definition of components is controversial; it has been extensively debated in the literature. Based on our experience as software developers, we find the definition in Szyperski's widely-cited book on the subject [30] to be the most compelling.

According to Szyperski, a component is a unit of *compiled code* that is:

- *independent*,
- composable by third parties, and
- devoid of *independent* state.

For a component to be an *independent* unit of compiled code, it must be completely separate from the environment in which it is defined and compiled precluding any explicit references to other components. Thus, a component completely encapsulates its features and can only be deployed in its entirety.

For a component to be composable by third parties, not only must it be self-contained, but it must also have well-defined, explicit specifications of its requirements (dependencies on components to be imported and their contracts) and features (provided interfaces and their contracts).

If a component has no independent state (such as an object allocation address), it can be used in any context satisfying its requirements. In any valid context (bindings of its dependencies), we are guaranteed that the component will always have the same behavior. If components had independent state, there would be no guarantee that two installations of the same component in the same context will have the same behavior. In fact, multiple copies of a component could be installed within a system and interfere with one another.

[3] We are using the term inheritance more broadly than inheriting code. In our terminology, a Java class inherits from all of its interfaces as well as its superclass.

2.3 Java Packages

In the Java programming model, packages are the primary construct used by programmers to organize a large-scale Java application into manageable units of development.

Packages provide distinct name spaces to organize collections of classes and interfaces. A class is included in a package by adding a `package` declaration at the top of the file and placing the file in the directory associated with the package. A packaged `public` class can be used outside the package by either: (i) using a fully-qualified reference to a class, or (ii) using an `import` statement[4] coupled with a non-qualified reference. In compiled class files, these two forms of usage are equivalent because the compiler translates non-qualified references into their corresponding fully-qualified names and discards all of the `import` statements.

Figure 1 and 2 shows an outline of the abstract syntax and recursive decent parser[5] for the "Jam" language, a simple functional language assigned as a series of programming exercises in the undergraduate course at Rice University on programming languages. The `Parser` resides in the package `jam.parser` and uses classes from `jam.ast`. Figure 3 shows the outline of an interpreter that references `jam.parser.Parser` and classes from `jam.ast`.

From a software engineering perspective, the fully-qualified references to `jam.ast.Exception` in Figure 2 and `jam.parser.Parser` in Figure 3 pose long-term code management problems. Each fully-qualified identifier is a hard-coded, absolute reference to an external class. To change the name of an imported class or a package, the programmer must perform a global search and replace of all the references to the class or package, which are scattered throughout the program.

At the macroscopic level, these references create a tangled web of hidden contextual dependencies across packages. As a result, packages cannot be developed in isolation. All of a package's dependencies (imported classes) must be present for both the compilation and execution of the package. A modification to classes in a package typically requires recompilation of all code that depends on it.[6]

These problems become more apparent when we consider alternative component implementations, *e.g.*, a bottom-

[4] Not to be confused with the imports to a CGEN module. Java `import` statements provide automatic expansion of non-qualified names to fully-qualified names by matching the non-qualified names against the members of the imported classes and interfaces.

[5] Due to space constraints, most of the code for this example has been elided. The complete code for this example (and others) is available at http://japan.cs.rice.edu/nextgen/examples/.

[6] Even changes to a class that preserve the existing "interface" can still force recompilation. For example, if the binding of a static final constant field is changed, all of the classes that refer to that field must be recompiled because the value of the field may be inlined in the compiled code for the classes that refer to it.

```
package jam.ast;

interface JamVal { ... }
interface AST { ...  }
interface IBinOp implements AST { ... }
class BinOpPlus implements IBinOp {
   public static final BinOpPlus ONLY;
}
...
class Exception {
   String getMessage();
}
```

Figure 1. Outline of Jam syntax

```
package jam.parser;
import jam.ast.*;

class Parser {
   Parser() { ... }
   AST parse(String url) throws ParseException { ... }
}
class ParseException implements jam.ast.Exception {
   String msg;
   ParseException(String msg) { ... }
}
```

Figure 2. Outline of Jam parser

```
import jam.ast.*;
import jam.parser.*;

class Interpreter {
  public Interpreter() { }
  public JamVal interp(String url) {
     Parser p = new Parser();
     try {
        jam.ast.AST tree = p.parse(url);
        ...
     } catch (ParseException e) { ... }
  }
}
```

Figure 3. Outline of Jam interpreter

up parser `jam.botparser.Parser`. In principle, we should be able to switch to this new parser by changing the import statement in Figure 3 to `import jam.botparser.*`. However, there is no guarantee this alternative parser provides the same interfaces, much less a similar set of classes, until we recompile the `Interpreter`. This violates our goal of components being independently compiled units of code.

2.4 Object Patterns

We could avoid recompilation by defining a common interface `IParser` and then passing an `IParser` to the constructor of the `Interpreter`, as shown in Figure 4 and 5. In essence, this approach is an idiom for manually representing components as objects. On a limited basis, this idiom can enable a Java class to accept minor changes in its imports without modifications to its source code. But as a scheme

```
package jam;
import jam.ast.*;

interface IParser {
   AST parse(String url) throws ParseException { ... }
}
```

Figure 4. Jam IParser interface

```
import jam.ast.*;
import jam.parser.*;

class Interpreter {
  IParser p;
  public Interpreter(IParser p) { this.p = p;}
  public JamVal interp(String url) {
     try {
        jam.ast.AST tree = p.parse(url);
        ...
     } catch (ParseException e) { ... }
  }
}
```

Figure 5. Jam Interpreter using IParser

for systematically eliminating explicit dependencies, it is unworkable. Since with this approach, components are objects, component linking only occurs during program execution when a class's imports (represented as objects) are passed as arguments to methods in the class's client interface. In realistic applications, a class may import dozens of classes, producing method signatures with dozens of parameters, which must be modified when a component's dependency structure changes. Even our pedagogic `Interpreter` class imports 75 classes. Furthermore, there is no mechanism in Java to prevent the introduction of hidden dependencies (explicit external class references) in a component class which may only surface when a client uses a new component configuration. Similarly, since this idiom represents components as objects, these components include independent state; it is easy to create multiple copies of the same component.[7]

2.5 JavaBeans

To provide more effective encapsulation of code units, the Java software community has developed the concept of JavaBeans. JavaBeans is an API that defines a "wiring standard" for assembling software components in Java. A Java "bean" is simply a regular Java class adhering to certain interface and coding conventions. For example, all beans must include a public 0-ary constructor and support persistence (serialization) so that the state of any bean can be saved and later reloaded. The classes `Parser` and `Interpreter` referred in Figures 2 and 3 can be converted into JavaBeans simply by making each class `implement`

[7] The singleton pattern could be used to ensure a single instance for components without imports. However, more complex bookkeeping would be needed for modules with imports.

the interface `java.io.Serializable`. An actual bean is represented by an instance of a Java bean class.

While Java beans can be deployed independently of other beans, they are not independent units of compiled code. A Java bean references other classes and types using the Java package system—just like any other class. Moreover, beans have independent state since they are conventional Java objects.

3. Architecture of CGEN

CGEN is a *first-class* formulation of Generic Java derived from NEXTGEN, a backward compatible extension of Java 5.0 supporting run-time generic types. The thesis underlying the design of CGEN is that Java components have a natural formulation as collections of generic classes with a common set of type parameters. All external references within a component class can be made manifest by expressing them as type parameters to the class that are instantiated when the component is linked. Hence, component linking can be reduced to the (type) application of generic classes to class arguments. Of course, an actual component system must include some new syntactic machinery for:

- defining components and their imported component parameters (dependencies);
- declaring the functionality (expressed at the level of member type declarations) imported and exported by components;
- linking components together; and
- checking that types of linked components match.

Fortunately, a first-class generic type system for classes (including hygienic mixins) provides the critical parts for assembling this machinery. In CGEN, we parameterize components by other components (collections of classes) instead of individual classes, but in the underlying compiled code, we can reduce component parameterization to class parameterization.

In CGEN, components are called *modules* rather than *components* because the name `Component` is extensively used in the Java GUI libraries. CGEN modules generalize Java packages. A *module* is a bundle of classes with a name qualifier (prefix) just like a package, but with a critically important difference. A *module* uses *signatures* to specify the functionality of the modules that it imports and the functionality that it exports to other modules. More succinctly, signatures provide the crucial machinery to support separate compilation of modules in CGEN.

In CGEN, modules and signatures have second-class status; they cannot be instantiated at runtime, used as arguments to methods, or used in any object-passing protocol. Signatures are annotations which are used exclusively during the compilation process. Modules exist at runtime only in the restricted sense that the members of the module are conventional Java classes and interfaces that are dynamically loaded at runtime. This restriction on modules and signatures simplifies the semantics and implementation of CGEN, while providing sufficient expressiveness to support a rich component system.

Besides modules and signatures, CGEN introduces one other construct to NEXTGEN, the notion of *binding* an identifier to a module instantiation. The `bind` construct is used to build an executable module definition with no dependencies (imports). In addition, it provides concise names for module instantiations and a simple mechanism for linking mutually recursive modules.

3.1 Signatures

A *signature* is a template specifying the classes (and interfaces) in a module and their members in prototype form. This template constrains the "shape" of a matching module. The syntax of a signature definition has the form:

```
signature S<V implements E,..., V implements E>
   [extends E, ..., E];
sigMember*
```

where *S* is a fully-qualified name for the signature (just like a package declaration in conventional Java); each *V* is an identifier denoting a module parameter (signature import); each *E* is a signature instantiation; and each *sigMember* is either an interface or class prototype, a `bind` statement, or an `import` statement. The *E* in each `implements` clause bounds the preceding parameter. The list of *E*s following the `extends` clause specifies a set of signatures from which *S* inherits, just as Java interfaces inherit from other interfaces. Import statements are used to access legacy packages, *e.g.*, classes in `java.io`.

A *signature instantiation* defines a *ground* (fully instantiated) signature by linking the import parameters of a signature with appropriate modules. A signature instantiation has exactly the same syntactic form as a generic type instantiation:

```
   sName<mExpr, ..., mExpr>
```

where *sName* is a fully-qualified signature name and each *mExpr* is either a module instantiation or an imported module parameter (which is bound to a module instantiation). The non-parametric signature `S<>` is abbreviated simply as `S`.

A *module instantiation* has the same syntactic form as a signature instantiation:

```
   mName<mExpr, ..., mExpr>
```

where *mName* is a fully-qualified module name and each *mExpr* is either a module instantiation or an imported module parameter (which is bound to a module instantiation). The non-parametric module `M<>` is abbreviated simply as `M`.

Note, parameters for signature and module instantiations are strictly first order (ground); hence a module or signature parameter can never be applied to arguments.

The intuitive meaning of a module instantiation is literally the set of classes in the instantiated module. It is equiv-

```
signature SSyntax;

interface JamVal { ... }
interface AST { ...  }
interface IBinOp implements AST { ... }
class BinOpPlus implements IBinOp {
   public static final BinOpPlus ONLY;
}
...
class Exception {
   String getMessage();
}
```

Figure 6. Signature for Jam syntax

```
signature SParser<A implements SSyntax>;

class Parser {
  Parser();
  A.AST parse(String url) throws ParseException;
}
class ParseException extends A.Exception {
   String msg;
   ParseException(String msg);
}
```

Figure 7. Signature for Jam parser

alent to a package in ordinary Java. Hence, there is no such thing as multiple instances (or copies) of a particular module instantiation.

Interface prototypes look like ordinary Java interfaces except that they may include references to imported modules (module parameters). Members of imported modules can be extracted using the familiar Java dot (".") notion for member selection. Class prototypes look like ordinary Java classes, except that they may reference module parameters and only provide method signatures—not actual implementations. In contrast to interface prototypes, class prototypes can include constructor prototypes, dynamic and static fields, dynamic and static method prototypes, and even inner interface and class prototypes as members. All of the members of a signature have `public` visibility. The visibility of the members of class prototypes can either be `public` or `protected`. `public` visibility is the default.

Figures 6 and 7 show the module signatures `SSyntax` and `SParser` used for a component-based parser for the Jam language. The signature `SParser` imports a module `A` implementing the signature `SSyntax`.

3.2 Modules

A module is a distinct name space that contains a collection of classes and interfaces and stipulates the signatures implemented by the module. More precisely, a *module* definition has the form:

module *M*<*V* implements *E*, ..., *V* implements *E*>
   implements *S*, ..., *S*;
moduleMember∗

```
module JamParser<A implements SSyntax> implements SParser<A>;

public class Parser extends Object {
   Parser() { ...  }
   A.AST parse(String url) throws ParseException { ... }
   String[] getLog() { ... }
}
public class ParseException extends A.Exception {
   String msg;
   ParseException(String msg) { this.msg = msg; }
}
```

Figure 8. JamParser Module Definition

where *M* is a fully-qualified name for the module (just like a package declaration in conventional Java); each *V* is an identifier serving as a module parameter (module import); each *E* is a signature instantiation bounding the corresponding parameter; each *S* is a signature instantiation bounding the module; and each *moduleMember* is either a class or interface definition, an import statement, or a `bind` statement. Figure 8 shows a module `JamParser` that implements the signature `SParser`.

The subtyping relation between modules and signatures is *nominal*: a module `M` implements a signature `S` only if `M` explicitly declares that it implements `S`. Hence, CGEN modules are nominally subtyped just like Java classes.

3.2.1 Module Type Checking

As discussed earlier, a signature declares the visible shape (functionality) of a module. In Java terminology, a module *implements* a signature just as a Java class *implements* an interface. More precisely, a module `M` *implements* a signature `S` if and only if `M` contains a *matching* class or interface `C'` for each class or interface prototype `C` in `S`. A class `C'` *matches* a class prototype `C` iff `C'` has exactly the same name as `C` and has members *matching* the prototype members declared in `C`.[8] For every member `m` of the class protoype `C`, the matching class `C'` must contain a member `m'` with the same name as `m`, the same type signature, and the same attributes (`public`/`protected`, `static`/dynamic, and `final`/mutable).[9] Since a class prototype may be a member of a class prototype, the matching process is recursive. The modules and its classes may contain extra members.

CGEN allows the parent type of class `C'` declared in module `M` to be a subtype of the declared parent type of the class prototype `C` in `S`. This relaxation of the matching relation for modules against signatures gives developers more freedom when revising a module that implements a given signature. The implications of this design decision are discussed further in Section 4.2.

The contents of a module are type-checked following Java 5.0 type checking rules where module imports (the *V*s

[8] In this context and several others, we use the term *class* to refer to either a Java `class` or `interface`

[9] Several of these issues are moot for interfaces.

```
bind SSyntax JSyntax = JamSyntax;
bind SParser<JSyntax> JParser = JamParser<JSyntax>;

public class Interpreter {
   public Interpeter() { }
   public JSyntax.JamVal interp(String url) {
      JParser.Parser parser  = new JParser.Parser(url);
      try {
         parser.parse(...);
      } catch (JParser.ParseException e) { ... }
   }
}
```

Figure 9. Module Instantiation

in the syntax rule for modules), are treated like imported packages with types synonymous with their bounding signatures and class and interface prototypes within signatures are treated exactly like ordinary class and interface definitions within packages. Then the class and interface definitions in the module must be checked against the bounds provided by the export signatures using the structural matching described earlier.

3.3 Bindings

An executable *program* module is a module definition with no dependencies (imports). Program modules typically link other modules together and usually include program specific code not intended for reuse. To facilitate the construction of program modules, CGEN includes a construct `bind` that binds an identifier to a (linked) module instantiation. A module binding has the form:

`bind` *sigExpr name* = *mExpr*;

where *sigExpr* is a fully-qualified signature instantiation, `name` is an identifier, and *mExpr* is either an imported module parameter (which is bound to a module) or a module instantiation. The *sigExpr* can be used to associate a weaker signature with *name* than the export signature of *mExpr*. In this way, the programmer can expose only the requisite part of a module with a larger export signature. Figure 9 shows a client that links and uses the module *JamParser*.

Module bindings are convenient because they provide short names for module instantiations which might otherwise be unwieldy. They also play an essential role in the construction of program modules with recursive links, *e.g.* module `A` imports module `B` and vice-versa.

4. Language Design Issues

In the evolution of CGEN, the principal technical complications we encountered were: (*i*) accidental overriding caused by inheritance across component boundaries, and (*ii*) cyclic class hierarchies caused by recursive or mutually-dependent modules.

Throughout the development of CGEN, we have endeavored to adhere the "safe-instantiation principle" [3]. Our adaptation of this principle is the following:

```
module JamDebugParser<A implements SSyntax,
                      B implements SParser<A>>
      implements SParser<A>;
public class Log { ... }

public class Parser extends B.Parser {
   Parser() { ...  }
   A.AST parse(String url) { ... }
   Log getLog() { ... }
}
public class ParseException extends B.ParseException {
   String msg;
   ParseException(String msg) { this.msg = msg; }
}
```

Figure 10. Inheritance Across Component Boundaries

> Instantiating a parametric construct with types that meet the declared constraints on the parameters should not cause an error.

But as our design progressed, we discovered that we had to relax this rule to provide the level of expressiveness that we wanted in a component system given our experience as software developers. We had to balance the benefits of early error detection against the the equally important goal of providing a component framework that supports wide latitude in refactoring component implementations while retaining compatibility with the former signatures (API's).

In the following subsections, we explore the the two major design complications that we encountered in developing CGEN and reflect on the tradeoffs among the various design possibilities.

4.1 Accidental Overriding

While the use of imports (module parameters) in CGEN superficially resembles the use of `import` statements in Java packages, they are semantically very different. In contrast to packages, references to imported modules are not resolved and bound when a module is compiled. Furthermore, type checking during compilation in CGEN is performed against the public members provided by the relevant signatures. Consequently, subtle complications can arise when components are linked (instantiated) that are not an issue in conventional Java. Inheritance across component boundaries defines a *mixin* class that can produce unexpected behavior.

Consider the following module definition:

```
module M<I1 implements S1> implements S;
class C extends I1.D { ... }
```

The module import `I1` is statically constrained by the bound `S1`. If the class `C` declared in `M` extends class `D` provided in the import `I1`, there can be unintended interference between the two classes for any member *m* not declared in the import bound `S1.D`. For example, consider the definition of module `JamDebugParser` in Figure 10, that provides a `Parser` with a more sophisticated logging facility. In the instantiated module:

```
bind SParser<JamSyntax> JParser2 =
     JamDebugParser<JamSyntax,JamParser<JamSyntax>>
```

the definition of `getLog()` in `JamDebugParser.Parser` *accidentally overrides* the method defined in its super class `JamParser.Parser`, thus breaking its super class.

This *accidental overriding* can be detected only during module linking when a module instantiation `M1` is linked against `S1`. Languages supporting separate compilation, such as Java and C#, cannot detect all such accidental overrides during compilation because the argument classes (provided by imported modules) and mixin classes (defined in modules) can be compiled in isolation from one another, and their only common interface, the type bounds on the argument classes (specified in signatures), does not mention what methods must be *excluded* to avoid accidental method overriding.

The issue of accidental overriding of methods by mixin instantiations has been extensively studied in the context of Generic Java in [1]. In this work, accidental overriding in first-class Generic Java is systematically avoided by using a *hygienic semantics* to avoid accidental name collisions, consisting of a method renaming and forwarding scheme that the class loader applies to every class as it is loaded by the JVM. Since CGEN mixins, which cross component boundaries, are more restrictive than class-level mixins, CGEN can employ exactly the same *hygiene* solution to this accidental overriding issue. Every mixin construction in a CGEN module is translated to a mixin class instantiation that is implemented using the techniques described for MIXGEN in [1]. In essence, this work shows that mixin hygiene is a clean, comprehensive solution to the accidental overriding problem. It supports sound local type checking that is consistent with the incremental class compilation and loading model in Java—conforming with the safe-instantiation principle.

CGEN supports exactly the same notion of safe instantiation with regard to accidental overrides as MIXGEN because it uses static signatures to bound module parameters and type-check their usage within modules.

4.2 Cyclic Class Hierarchies

Generic Java forces class hierarchies to form a DAG (direct acyclic graph) under the subtyping relation. All generic types are erased to their corresponding base types and these base types are then used to confirm this hierarchy.

In the presence of modules we must enforce the same constraint. However, the local analysis of the type hierarchy performed during the compilation of an individual module is less precise. The collection of classes is not completely known during the compilation of an individual module. Complete cycle detection in the class hierarchy must be deferred until the evaluation of `bind` declarations (module linking) in the third-party client.

The nominal typing of CGEN provides transparent support for recursion in signatures and modules. A signature `S<A>` can refer to itself by simply referencing its type-instantiated identifier.

```
signature S<A implements S<A>>;
```

In CGEN, the presence of second-class modules and signatures enables a single recursive module or a collection of mutually recursive (or dependent) modules to create a cyclic hierarchy. The following examples show some of the potential complications that can arise.

Example 1.
A class prototype extending an imported class in a recursive module can extend itself.

```
signature S<A implements S<A>>;
 class C extends A.C { };
```

In this example, the module import `A` is recursively bound to the type of the enclosing module instantiation `S<A>`. A module supporting `S<A>` such as:

```
module M<A implements S<A>> implements S<A>;
 class C extends A.C { ... };
```

can be instantiated using the following bind:

```
bind S<X> X = M<X>;
```

While the use of the variable `X` in both the left hand side of its signature bound and the right hand side declaring its module instantiation may seem peculiar, this pattern defines a module recursively just like a recursive definition of a function. The class `C` in the module `X` has `X.C` (itself) as its superclass, creating an illegal cyclic class hierarchy.

The cycle in this example can be detected solely by analyzing the definition of signature `S< ... >`. The signature definition states that given module instantiation `A` implementing (subtyping) `S<A>`, then `S<A>.C` extends (properly subtypes) `A.C`. But the assumption `A` implements (subtypes) `S<A>` implies that `A.C` implements (subtypes) `S<A>.C`. Hence `A.C` equals `S<A>.C` by the anti-symmetry of subtyping. Yet `S<A>.C` extends (properly subtypes) `A.C` so the two types cannot be equal. So the signature definition in Example 1 is illegal preventing `M<...>` and `X` from being defined in CGEN.

Example 2. We can revise Example 1 to generate a valid definition for `S<...>` by deferring the subtyping on `S<A>.C` until we declare `M<...>` as discussed in Section 3.2.1. In this way, a cyclic class hierarchy can be concealed by its signature. The signature in Example 1 can be perturbed as follows:

```
signature S<A implements S<A>>;
 class C { }
```

which conceals the cyclic hierarchy defined by the module `M<A>`:

```
module M<A implements S<A>> implements S<A>;
 class C extends A.C { ... }
```

As before, we can instantiate M using the following bind:

```
bind S<X> X = M<X>;
```

to create an illegal cycle. In this example, the signature S<...> is valid because class S<A>.C extends Object. The class A.C implements (subtypes) S<A>.C by assumption, but S<A>.C does not necessarily subtype A.C.

In processing Example 2, the CGEN compiler accepts the definition of the signature S<...> and the definition of M<...>, but it rejects the bind operation defining X using essentially the same analysis used to reject S<...> in Example 1.

Example 3. In this example, we demonstrate that the signature and module defined in Example 2 can be augmented by another module and a different bind to form a valid program. This example consists of the following program text:

```
signature S<A implements S<A>>;
 class C { }

module M<A implements S<A>> implements S<A>;
 class C extends A.C { ... }

module N<A implements S<A>> implements S<A>;
 class C { ... }

bind S<Y> Y = N<Y>;
bind S<Y> X = M<Y>;
```

The module instantiation Y = N<Y> implements S<Y> without creating a cycle. Then Y can be passed as an argument module M without creating a cycle.

The preceding three examples were obviously chosen for simplicity. In general, the subclassing hierarchy introduced by flexible signature matching can be arbitrarily deep. Similarly, cycles in the type hierarchy can involve an arbitrarily long chain of signatures or modules

In the compilation of an individual module, CGEN can enforce the DAG constraint on the type hierarchies determined by the available code. In Example 1, the type hierarchy for the signature definition S<...> creates a cycle, enabling the CGEN compiler to detect the error in the example immediately. In Example 2, the cycle in the type hierarchy does not appear until the bind operation defines the module instantiation X. In Example 3, the bind operation from Example 2 is dropped and replaced by a second module definition and two subsequent bind operations. This change creates a valid CGEN program with no cycles in the type hierarchy.

These three examples clearly demonstrate that cycles in the type hierarchy can sometimes be detected by analyzing signatures but in general case they cannot be detected until modules are instantiated in bind statements. Note that this process does not fully comply with the safe instantiation principle identified in [3]. The type arguments in a bind statement can satisfy their bounds (as in Example 2) yet the generated program may still be illegal because the bind statements create a cycle in the type hierarchy.

As an alternative, CGEN could prohibit modules from declaring classes with a more specific parent type than that explicitly declared in its export signature. This restriction would reduce detecting cyclic type hierarchies to checking that prototypes in signatures form a DAG—a process that can be done statically when the importing module is compiled since all relevant signatures are available during its compilation. While this restriction would make CGEN conform to the letter of the safe-instantiation principle, we believe it would impose unacceptable restrictions on module developers. Our relaxed constraints provide developers with far more flexibility in code development and refactoring while still supporting the detection of cyclic type hierarchies during compilation. It simply postpones detection of some cyclic type hierarchies until module instantiation because the cycle is created by the particular instantiation.

5. Core CGEN

In order to identify and resolve the subtle technical issues in the CGEN type system, we have distilled the component framework into a small, core language called Core CGEN, CCG for short. Indeed, several complications described in Section 4 were uncovered during a formal analysis of Component NEXTGEN.

The design of Core CGEN is based on Featherweight GJ (FGJ) [14] and incorporates ideas from Core MixGen [1]. In the remainder of this paper, we will refer to these two languages as FGJ, and CMG, respectively. CCG excludes signatures for the sake of simplicity; degenerate modules are used in place of signatures. This simplification follows the precedent established in Featherweight Java, FGJ, and CMG which exclude interfaces and rely on degenerate classes in their stead.

CCG augments FGJ with the essential infrastructure to support CGEN-style components:

- module definitions. These provide the crucial framework to bundle classes as components.
- bind definitions. These provide the ability to link (instantiate) modules with their dependencies. For simplicity, CCG programs define a single set of bind declarations.
- Multiple constructors in class definitions. In FGJ each class defines a default constructor that takes an initial value for each field as an argument. In CCG, we relax this restriction and permit multiple constructors with arbitrary signatures. This allows (i) classes to satisfy the constraints required by multiple bounding signatures, and (ii) different module implementations to provide different collections of fields.

The Appendix outlines the syntax, type system, small-step semantics, and proof of type soundness for CCG; the details of the proof are available in a technical report [29]. The semantics and proof are similar to those for MixGen in [2], except for the following:

- All class types in CCG are prefixed by their enclosing module instantiation. This convention is analogous to using fully-qualified class names including a package prefix in Java.
- CCG must check that class hierarchies form a DAG when modules are linked together with respect to the declared bind declarations.
- The small-step semantics for CCG carries a runtime "bound" environment, mapping type variables to their bounds, to support bind declarations.

5.1 Lessons Learned from Core CGEN

We did not understand the tradeoff between flexibility in matching modules against signatures and deferred detection of cycles in the type hierarchy until we worked through the semantics of CCG and the proof of type soundness. In constructing the proof, we realized that cycles could be created by the particular structure of the module instantiations in a block of `bind` statements.

The development of CCG also provided us with a model for building the CGEN type-checker. We built a prototype CGEN type checker before developing CCG, but it did not handle recursive modules properly.

We also learned that extending Java generics and components to support passing abstracted (higher-order) modules is a major technical challenge. It is not clear how to generalize our formal semantics and soundness proof to handle this extension. The class table *CT* that we trivially construct (see the Appendix) for CCG is not necessarily finite when abstracted modules can be passed as arguments.

6. Related Work

Several past and ongoing research projects at other institutions have focused on developing some notion of a component or module system for Java, but none of them are based on first-class generics and dynamic loading to link components.

JAVAMOD [6] is a statically typed extension to Java that supports mixin modules. Mixin modules are a generalization of mixin classes as introduced in [7]. These mixins are *not* hygienic, but instead rely on an explicit *hiding* operator. In contrast to CGEN where modules are nominally typed against reusable signatures specifying module imports and exports, modules in JAVAMOD each define an *interface* containing a list of imported and exported classes. Module linking, called *merging*, requires a exact structural match between the import and export *interfaces* of the relevant modules. Inconsistencies between interfaces can be mitigated [10] by *hiding* and *renaming* operations, but the use of structural matching of signatures doesn't meld seamlessly in the nominally typing of Java or other statically typed object-oriented languages.

SMARTJAVAMOD [5] is a refinement of JAVAMOD that relies on type inference, instead of statically typed interfaces, to determine the principal typing of a module. The type inference process infers type constraints for free type parameters in the module. Uninstantiated modules are compiled into polymorphic bytecode similar to NEXTGEN templates. Inferred type constraints are checked when modules are combined. Accidental overriding can be detected in this process and flagged as an error. In essence, SMARTJAVAMOD relies on free variables and type inference to simulate first-class generics at the cost of hiding the parameters of a module (the free variables) and their type bounds (which are inferred). We don't see such a facility as consistent with the spirit of explicit static typing in Java. While the use of untyped free variables in modules supports flexible usage (except in the presence of accidental overriding), it prevents module developers from verifying export and import specifications when modules are developed independently of one another.

Jiazzi [16] is a component definition and linking language for Java. Jiazzi uses a stub generator and a static linker that allow components to be written in the unextended Java language. Jiazzi's linking facilities use the *open class pattern*—a combination of mixins and "upside-down" mixins—to support the modular addition of new features to a set of variant classes. For each linked component, Jiazzi generates a *fixed-package*, a form of fixed-point representing the linked classes. As a result, the semantics of Jiazzi components—even though they are written in "ordinary Java"—is more complex than their familiar Java syntax suggests.

Using a separate linking language like Jiazzi significantly complicates the development of component code. Before a component file can be compiled, the component's signature must be declared in a Jiazzi `.sig` file, which is translated by the Jiazzi stub generator to produce the stub class files required to support the compilation of the component file by the standard `javac` compiler. The linking of compiled components is specified by a Jiazzi `.unit` file which is fed to the Jiazzi linker along with the `.sig` files and class files for the compiled components.

Scala [23, 24] is a multi-paradigm programming language implemented on top of the Java Virtual Machine. Scala has a more concise syntax than Java and supports a similar nominal type system including a richer form of interfaces that includes executable code (but no fields).

[10] Since each module declares its own *interfaces*, the requirement for an exact structural matching places a heavy burden on component linkers to ensure the *interfaces* match.

Scala *modules* are singleton classes that do not specify a static signature, which prevents them from creating independent state. And Scala does not support first-class generics, preventing modules from being parameterized by type.[11] Consequently, Scala components rely on hard-coded references to other modules and classes, making it more difficult to develop modules independently of one another.

Fortress [19] is a new programming language developed by Sun Microsystems Research targeting high performance computing (HPC), incorporating both functional and object-oriented programming methodologies. Components and APIs in Fortress are designed in much the same spirit as modules and signatures in CGEN. The principal difference is that Fortress relies on static linking instead of first-class genericity to instantiate program components. Instead of importing a list of modules, Fortress components identify an unordered set of APIs (signatures) that they import. As a result, components cannot link against multiple imports that provide the same API. In contrast to the pure dynamic linking of *bind* declarations in the CGEN, Fortress maintains a persistent database of components and requires static linking of components. Fortress uses an interactive shell to *compile* and *link* components.

The CGEN component framework has also been influenced by the design of module systems for functional languages, most notably ML. The fundamental difference between ML modules (in all the various formulations) and CGEN modules is that ML modules rely on structural typing, while CGEN relies on nominal typing. This distinction is important because it means that CGEN seamlessly supports mutual recursion among modules, a common practice in software engineering. Supporting mutual recursion among modules has proven to be a challenging technical problems in the context of structural subtyping [26, 10, 11]. Neither of the two widely used dialects of ML–Standard ML [21] and OCaml [15]–supports recursive modules.

One issue that has arisen in putative recursive extensions to the ML module system is the *double vision* problem[11], where a single type can be referenced using distinct names with differing visible types. In this scenario, the type checker can reject valid programs because the "different" types cannot be matched together. The same situation cannot arise in CGEN or Java generics because the nominal type system provides unique (canonical) names for all types. Moreover, imported class types in CGEN can reference classes that are explicitly declared in the same context by using mutually dependent signatures/modules as discussed in Section 4.2.

A potential solution to the ML recursive module problem is the module system proposed in [25], which is based on the same module architecture for PLT Scheme discussed in [12]. In keeping with the ML programming model, it relies on static, structural typing. The current design does not include signatures and therefore cannot re-use import and export specifications of units.

Recently, two Java JSR's concerning components have been submitted the Java Community Process, namely JSR-277 and JSR-294. *Java Specification Request 277: Java Module System*[18] addresses the extra-linguistic concerns of module deployment, including versioning and distribution. It is orthogonal to CGEN and could be integrated with our design. The public information on *Java Specification Request 294: Improved Modularity Support in the Java Programming Language*[20] appears closer in focus to CGEN, but the available public specifications are too fragmentary for us to compare it in detail with CGEN other than to say it is less ambitious and appears only to support information hiding akin to CGEN signatures; it does not support component assembly or replacement as discussed by Szyperski.

[11] With erasure-based generics, the meaning of a class is independent of its type instantiation (which has no impact on the executable code). Hence, a component cannot be linked with its imports by type application.

References

[1] E. Allen, J. Bannet, and R. Cartwright. First-class genericity for Java. In *OOPSLA*, 2003.

[2] E. Allen, J. Bannet, and R. Cartwright. Mixins in Generic Java are sound. Technical report, Rice University, 2003.

[3] E. Allen and R. Cartwright. Safe instantiation in Generic Java. In *PPPJ '04*, pages 61–66, 2004. Available at `http://www.cs.rice.edu/CS/PLT/Publications`.

[4] 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.

[5] D. Ancona, G. Lagorio, and E. Zucca. Smart modules for Java-like languages. In *7th Intl. Workshop on Formal Techniques for Java-like Programs*, 2005.

[6] D. Ancona and E. Zucca. True modules for java classes. In *ECOOP*, 2001.

[7] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity, and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[8] 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.

[9] R. Cartwright and G. L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In *OOPSLA*, 1998.

[10] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *PLDI*, pages 50–63, New York, NY, USA, 1999. ACM Press.

[11] D. Dreyer. *Understanding and Evolving the ML Module System.* PhD thesis, Carnegie Mellon University, 2005.

[12] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *SIGPLAN*, pages 236–248, 1998.

[13] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, 1999.

[14] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, 1999.

[15] X. Leroy. The Objective Caml system: Documentation and user's manual., 2004. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[16] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *OOPSLA*, 2001.

[17] S. Microsystems. JSR 14: Adding generic types to the Java Programming Language, 2001.

[18] S. Microsystems. JSR 277: Java Module System, 2005.

[19] S. Microsystems. The Fortress language specification, Sept 2006.

[20] S. Microsystems. JSR 294: Improved Modularity Support in the Java Programming Language, 2006.

[21] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[22] A. Myers, J. Bank, and B. Liskov. Parameterized types for Java. In *POPL*, 1997.

[23] M. Odersky. Programming in Scala, 2006. Draft.

[24] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA*, pages 41–57, New York, NY, USA, 2005. ACM Press.

[25] S. Owens and M. Flatt. From structures and functors to modules and units. In *ICFP*, volume 41, pages 87–98, New York, NY, USA, 2006. ACM Press.

[26] C. V. Russo. Recursive structures for standard ML. In *ICFP*, pages 50–61, New York, NY, USA, 2001. ACM Press.

[27] J. Sasitorn and R. Cartwright. Efficient first-class generics on stock Java virtual machines. In *SAC*, 2006.

[28] J. Sasitorn and R. Cartwright. Deriving compnents from genericity. In *SAC*, 2007.

[29] J. Sasitorn and R. Cartwright. The soundness of Component NextGen. Technical report, Rice University, 2007. `http://www.cs.rice.edu/~javaplt/papers/`.

[30] C. Szyperski. *Component Software*. Addison-Wesley, 1998.

[31] C. A. Szyperski. Import is not inheritance: Why we need both: modules and classes. In O. L. Madsen, editor, *ECOOP*, volume 615, pages 19–32, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[32] M. Viroli. Parametric polymorphism in Java: an efficient implementation for parametric methods. In *SAC*, 2001.

[33] M. Viroli and A. Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. *ACM SIGPLAN Notices*, 35(10):146–165, 2000.

A. Core CGEN

This Appendix outlines a semantics, type system, and proof of type soundness for CCG; the details of the proof are available in a technical report [29].

Syntax:

$$
\begin{array}{rcl}
\texttt{MD} & ::= & \texttt{module}\ \mathbb{D}\texttt{<}\overline{\mathbb{X} \lhd \mathbb{N}}\texttt{>} \lhd \mathbb{N}\ \{\overline{\texttt{CL}}\} \\
\texttt{CL} & ::= & \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{T}}\,\overline{\texttt{f}}\texttt{;}\ \overline{\texttt{K}}\ \overline{\texttt{M}}\} \\
\texttt{K} & ::= & \texttt{C(}\overline{\texttt{T}}\,\overline{\texttt{x}}\texttt{)}\ \{\texttt{super(}\overline{\texttt{e}}\texttt{);this.}\overline{\texttt{f}}\ \texttt{=}\ \overline{\texttt{e}'}\texttt{;}\} \\
\texttt{M} & ::= & \texttt{<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>}\ \texttt{T}\ \texttt{m(}\overline{\texttt{T}}\,\overline{\texttt{x}}\texttt{)}\ \{\texttt{return e;}\} \\
\texttt{e} & ::= & \texttt{x} \\
 & | & \texttt{e.f} \\
 & | & \texttt{e.m<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \\
 & | & \texttt{new N(}\overline{\texttt{e}}\texttt{)} \\
 & | & \texttt{(N)e} \\
\texttt{T} & ::= & \texttt{X} \quad | \quad \texttt{N} \\
\mathbb{T} & ::= & \mathbb{X} \quad | \quad \mathbb{N} \\
\texttt{N} & ::= & \mathbb{T}\texttt{.C<}\overline{\texttt{T}}\texttt{>} \\
\mathbb{N} & ::= & \mathbb{D}\texttt{<}\overline{\mathbb{T}}\texttt{>} \\
\texttt{BD} & ::= & \texttt{bind}\ \mathbb{N}\ \mathbb{X}\ \texttt{=}\ \mathbb{N}\texttt{;}
\end{array}
$$

Table 1. CCG Syntax

A.1 Syntax

The abstract syntax of CCG, shown in Table 1, consists of module declarations (`MD`), class declarations (`CL`), constructors declarations (`K`), method declarations (`M`), expressions (`e`), types (`T`), and bind declarations (`BD`). For the sake of brevity, `extends` is abbreviated by $\lhd$. Throughout all formal rules of the language, the following meta-variables are used over the following domains:

- `d`, `e` range over expressions.
- `K` ranges over constructors.
- `m`, `M` range over methods.
- `N`, `O`, `P` range over fully-qualified class types
- $\mathbb{N}$, $\mathbb{O}$, $\mathbb{P}$ range over module types
- `X`, `Y`, `Z` range over naked class type variables.
- $\mathbb{X}$, $\mathbb{Y}$, $\mathbb{Z}$ range over naked module type variables.
- `R`, `S`, `T`, `U`, `V` range over class types.
- $\mathbb{R}$, $\mathbb{S}$, $\mathbb{T}$, $\mathbb{U}$, $\mathbb{V}$ range over module types.
- `x` ranges over method parameter names.
- `f` ranges over field names.
- `C`, `D` range over class names.
- $\mathbb{C}$, $\mathbb{D}$ range over module names

Following the notation of FGJ, a meta-variable with a horizontal bar above it represents a (possibly empty) sequence of elements in the domain of that variable, and may include an arbitrary separator character. For example, $\overline{\texttt{T}}$ denotes a sequence of types $\texttt{T}_0, \ldots \texttt{T}_n$. As in CMG, we abuse this notation in different contexts. For example, $\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}$ denotes $\texttt{T}_0\ \texttt{f}_0\texttt{;} \ldots \texttt{T}_n\ \texttt{f}_n\texttt{;}$. Similiarly, the expression $\overline{\texttt{X}} \lhd \overline{\texttt{N}}$ represents $\texttt{X}_0 \lhd \texttt{N}_0, \ldots \texttt{X}_n \lhd \texttt{N}_n$.

In CCG, sequences of classes, type variables, method names, and field names are required to contain no duplicates. The set of type variables in each module includes an implicit variable `thisMod` which cannot appear as a type parameter anywhere in the module. The set of variables in each class includes an implicit variable `this` which cannot appear as a class name, field, or method parameter anywhere in the class.

Type variable bounds may reference other type parameters declared in the same scope; in other words they may be mutually recursive. Every module definition declares a super module using $\triangleleft$. Every class definition declares a super class using $\triangleleft$.

CCG requires explicit polymorphism on all parametric method invocations.

A.2 Valid Programs

A CCG program consists of a fixed module table, a fixed bind table and an expression, denoted $(\mathit{MT}, \mathit{BT}, \mathtt{e})$. A module table *MT* is a mapping from module names $\mathbb{D}$ to module declarations $\mathtt{MD}$. A bind table *BT* is a mapping from type variables $\mathbb{X}$ to bind declarations $\mathtt{BD}$.

A valid module table *MT* must satisfy the following constraints: (i) for every $\mathbb{D}$ in *MT*, $\mathit{MT}(\mathbb{D}) = \mathtt{module}\ \mathbb{D}...$, (ii), $\mathtt{Mod} \notin \mathit{dom}(\mathit{MT})$, (iii) every module appearing in *MT* is in $\mathit{dom}(\mathit{MT})$, and (iv) the subtyping relation $<:$ induced by *MT* is antisymmetric and forms a tree rooted at `Mod`. The root module `Mod` is modeled without a corresponding module definition in the module table and contains no classes.

A valid bind table *BT* must satisfy the following constraints: (i) for every $\mathbb{X}$ in *BT*, $\mathit{BT}(\mathbb{X}) = \mathtt{bind}\ \mathbb{N}_0\ \mathbb{X}\ \mathtt{=}\ \mathbb{N}$; and (ii) every type variable $\mathbb{X}$ appearing in *BT* is in $\mathit{dom}(\mathit{BT})$. Given the set *BT* of binds $\mathtt{bind}\ \overline{\mathbb{N}_0}\ \overline{\mathbb{X}}\ \mathtt{=}\ \overline{\mathbb{N}}$, the initial bound environment, mapping module type variables to their upper bounds is defined as $\Delta_{BT} = \overline{\mathbb{X}} \triangleleft \overline{\mathbb{N}}$. The bound environment Δ is discussed below in Section A.4.

Program execution consists of evaluating `e` in the context of *MT* and the initial bound environment Δ_{BT} for *BT*.

A.3 Valid Module Binds

To determine if a set of binds *BT* can be safely linked to produce an acyclic set of classes[12], an implicit class table *CT*, defining a mapping from fully-qualified class names to definitions, is generated by evaluating *MT* in the presence of *BT*. For each bind declaration $\mathtt{bind}\ \mathbb{N}_0\ \mathbb{X}\ \mathtt{=}\ \mathbb{D}\mathtt{<}\overline{\mathbb{T}}\mathtt{>;}$ in *BT* where $\mathit{MT}(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\mathtt{>} \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\}$ and each $\mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{...\} \in \overline{\mathtt{CL}}$:

$$\mathit{CT}(\mathbb{D}\mathtt{<}\overline{\mathbb{T}}\mathtt{>.C}) = [\overline{\mathbb{Y}} \mapsto \mathit{bound}_{\Delta_{BT}}(\overline{\mathbb{T}})]$$
$$\mathtt{class}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}}\mathtt{>.C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{...\}.$$

The substitution above replaces module type parameters with their *bind*-ed instantiations so that the parent type of a class in an instantiated module can be looked up in *CT*. The domain of *CT* is finite; It simply consists of the set of classes defined in the module instantiations in the right hand sides of `bind` declarations.[13]

$$\Delta \vdash \mathtt{T} <: \mathtt{T}\ \text{[SCReflex]} \qquad \frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \quad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}}\ \text{[SCTrans]}$$

$$\Delta \vdash \mathbb{T} <: \mathbb{T}\ \text{[SMReflex]} \qquad \frac{\Delta \vdash \mathbb{S} <: \mathbb{T} \quad \Delta \vdash \mathbb{T} <: \mathbb{U}}{\Delta \vdash \mathbb{S} <: \mathbb{U}}\ \text{[SMTrans]}$$

$$\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})\ \text{[SCBound]} \qquad \Delta \vdash \mathbb{X} <: \Delta(\mathbb{X})\ \text{[SMBound]}$$

$$\frac{\begin{array}{c}\mathit{bound}_{\Delta}(\mathbb{T})\ \mathtt{=}\ \mathbb{D}\mathtt{<}\overline{\mathbb{T}}\mathtt{>} \\ \mathit{MT}(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\mathtt{>} \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{...\} \in \overline{\mathtt{CL}}\end{array}}{\Delta \vdash \mathbb{T}\mathtt{.C<}\overline{\mathtt{S}}\mathtt{>} <: [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{T}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\mathtt{N}}\ \text{[SClass]}$$

$$\frac{\mathit{MT}(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\mathtt{>} \triangleleft \mathbb{N}\ \{...\}}{\Delta \vdash \mathbb{D}\mathtt{<}\overline{\mathbb{T}}\mathtt{>} <: [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{T}}]\mathbb{N}}\ \text{[SModule]}$$

$$\mathit{bound}_{\Delta}(\mathtt{X}) = \Delta(\mathtt{X})$$

$$\mathit{bound}_{\Delta}(\mathbb{V}\mathtt{.C<}\overline{\mathtt{T}}\mathtt{>}) = \mathit{bound}_{\Delta}(\mathbb{V})\mathtt{.C<}\overline{\mathtt{T}}\mathtt{>}$$

$$\mathit{bound}_{\Delta}(\mathbb{N}) = \mathbb{N}$$

Table 2. Subtyping and Type Bounds

A valid class table must satisfy the following constriants: (i) every class name appearing in *CT* is in $\mathit{dom}(\mathit{CT})$ and (ii) the set of class definitions must form a tree rooted at `Object`.[14]

The class `Object` is modeled as a top-level construct, located outside any module. `Object` contains no fields or methods and it acts as if it contains a special, zero-ary constructor.

A.4 Type Checking

The typing rules for expressions, method declarations, constructor declarations, class declarations, and module declarations are shown in Table 2. The typing rules in CCG includes two environments:

- A bound environment Δ mapping type variables to their upper bounds. Syntactically, this is written as $\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}} \cup \overline{\mathbb{X}} \triangleleft \overline{\mathbb{N}}$. The bound of a type variable is always a non-variable. The bound of a non-variable module type $\mathbb{N}$ is $\mathbb{N}$, and a non-variable class type $\mathbb{V}\mathtt{.C<}\overline{\mathtt{T}}\mathtt{>}$ is the class type $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ prefixed by the bound of the enclosing module $\mathbb{V}$.
- A type environment Γ mapping program variables to their static types. Syntactically, these mappings have the form $\overline{\mathtt{x}} : \overline{\mathtt{T}}$

CCG contains two disjoint sets of types: module types and class types. Class types are always qualified with their enclosing module instantiation. Class type variables are

[12] Since mixin classes in MIXGEN syntactically encoded their parent instantiations, formalizing properties on acyclic type hierarchies is relatively easy. CCG, just like FGJ, does not use a syntactic encoding.

[13] Modules *do not* contain any module type applications because they can only reference imported modules which are fully-qualified.

[14] The special class object is not in *CT*, because `Object` is a keyword.

bound by class types and module type variables are bound by module types. Figure 2 shows the rules for the subtyping relation $<:$. Subtyping in CCG is reflexive and transitive. Classes and modules are subtypes of the instantiations of their respective parent types.

A.5 Well-formed Types and Declarations

The rules for well-formed constructs appear in Table 3. Class and module instantiations are well-formed in the environment Δ if all instantiations of type parameters are subtypes of their bounds in Δ. Type variables are well-formed if they are present in Δ.

A method `m` is well-formed with respect to its enclosing module $\mathbb{D}$ and class `C` if its constituent types are well-formed; the type of the body in Γ is a subtype of the declared return type; and `m` is a *valid override* of any method of the same name in the static type of parent class for `C`. An overriding method definition is valid if it preserves the signature in the parent. This notion is formalized in section A.6.

CCG 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. In order 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, `this`, or `thisMod`).

A class definition `CL` is well-formed in the context of the enclosing module if the constituent elements are well-formed; none of the fields known statically to occur in ancestors are shadowed[15].; and every constructor has a distinct signature.

Module definitions are well-formed if their constituent elements are well-formed; each member class has a distinct name; and the module provides *valid class overrides* for all classes defined in its super module. This notion is formalized in section A.6.

Bind declarations are well-formed if the instantiated types are well-formed and subtypes of their formal types with respect to Δ.

A program is well-formed if all module definitions are well-formed, the induced module table is well-formed, the set of binds is well-formed, and the trailing expression can be typed with empty type environment (Γ) and bound environment Δ_{BT}.

A.6 Class and Module Auxiliary Functions

The auxiliary functions defining field name and value lookup, method typing, method lookup, valid method overrides, constructor inclusion, and method inclusion appear in Table 5. The auxiliary functions defining valid class overrides appear in Table 6. These functions are passed a bound environment Δ to determine the module bounds of type parameters.

The mapping *fields* returns only the fields directly defined in a class definition. The mapping *fieldVals* is used to retrieve the field values for a given object. A static type is passed to *fieldVals* to disambiguate field names in the presence of accidental shadowing.

Method types are resolved by searching upward the class inheritance chain (which may cross modules via imports) starting from the provided static receiver type. The type of a method includes the enclosing module and class instantiation in which the method occurs, as well as the parameter types and return types. The included module and class names are used to annotate receiver expressions in the typing rules for method invocation. As explained later in Section A.7, the annotated type of the 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 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.

For a given module $\mathbb{D}$, a class definition `CL` defines a *valid class override* if `CL` provides a super set of the fields, constructors, and methods provided by the class of the same name, if it exists, in the static type of the super module for $\mathbb{D}$.

A.7 Expression Typing

The rules for expression typing are given in Table 4. Naked type variables may occur in casts, but are prohibited in `new` operations.

The expression typing rules annotate the receiver expression for field lookups and method invocations 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. 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 A.6.

Notice that the receiver expression of a method invocation is annotated not with its precise static type, but instead with the closest supertype of the static type in which the called method is explicitly 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 annotation are used at run-time to determine which constructor is referred to by the new operation. This is because the semantics require an exact match. There could be cases where multiple

[15] This restriction is not necessary in principle, but it significantly simplifies the proof of type safety.

$$\Delta \vdash \mathtt{Object}\ \text{ok}_{[\text{WFObject}]} \qquad \Delta \vdash \mathtt{Mod}\ \text{ok}_{[\text{WFMod}]} \qquad \frac{\mathtt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \mathtt{X}\ \text{ok}}\,[\text{WFVar}] \qquad \frac{\mathbb{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \mathbb{X}\ \text{ok}}\,[\text{WFMVar}]$$

$$\frac{\begin{array}{c} \mathit{bound}_{\Delta}(\mathbb{T}) = \mathbb{D}<\overline{\mathbb{T}}> \quad \Delta \vdash \mathbb{D}<\overline{\mathbb{T}}>\ \text{ok} \\ MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}<\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}> \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \mathtt{class}\ \mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}> \triangleleft \mathtt{N}\ \{\ldots\} \in \overline{\mathtt{CL}} \\ \Delta \vdash \overline{\mathtt{S}}\ \text{ok} \quad \Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{T}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\overline{\mathtt{N}} \end{array}}{\Delta \vdash \mathbb{T}.\mathtt{C}<\overline{\mathtt{S}}>\ \text{ok}}\,[\text{WFClass}] \qquad \frac{\begin{array}{c} MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}<\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}> \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \Delta \vdash \overline{\mathbb{T}}\ \text{ok} \quad \Delta \vdash \overline{\mathbb{T}} <: [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{T}}]\overline{\mathbb{N}} \end{array}}{\Delta \vdash \mathbb{D}<\overline{\mathbb{T}}>\ \text{ok}}\,[\text{WFModule}]$$

$$\frac{\begin{array}{c} MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}<\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}> \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \quad \mathtt{class}\ \mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}> \triangleleft \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{K}}\ \ \overline{\mathtt{M}}\} \in \overline{\mathtt{CL}} \\ \Delta = \overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}} + \mathtt{thisMod} \triangleleft\ \mathbb{D}<\overline{\mathbb{Y}}> + \overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}} \quad \mathtt{N} = \mathbb{V}.\mathtt{D}<\overline{\mathtt{S}}> \quad \mathit{bound}_{\Delta}(\mathbb{V}) = \mathbb{C}<\overline{\mathbb{Z}}> \\ \overline{\mathtt{x}} \cap \mathtt{this} = \emptyset \quad \Delta \vdash \mathit{override}(\mathbb{C}<\overline{\mathbb{Z}}>.\mathtt{D}<\overline{\mathtt{S}}>, <\overline{\mathtt{X}'} \triangleleft \overline{\mathtt{R}'}>\ \mathtt{V}'\ \mathtt{m}(\overline{\mathtt{T}'}\ \overline{\mathtt{x}})) \\ \Delta_1 = \Delta + \overline{\mathtt{X}'} \triangleleft \overline{\mathtt{R}'} \quad \Gamma = \overline{\mathtt{x}} : \overline{\mathtt{T}'} + \mathtt{this}{:}\,\mathbb{D}<\overline{\mathbb{Y}}>.\mathtt{C}<\overline{\mathtt{X}}> \\ \Delta_1 \vdash \overline{\mathtt{R}'}\ \text{ok} \quad \Delta_1 \vdash \overline{\mathtt{R}'} <: \mathtt{Object} \quad \Delta_1 \vdash \mathtt{V}'\ \text{ok} \quad \Delta_1 \vdash \overline{\mathtt{T}'}\ \text{ok} \quad \Delta_1; \Gamma \vdash \mathtt{e} \in \mathtt{U} \quad \Delta_1 \vdash \mathtt{U} <: \mathtt{V}' \end{array}}{<\overline{\mathtt{X}'} \triangleleft \overline{\mathtt{R}'}>\ \mathtt{V}'\ \mathtt{m}(\overline{\mathtt{T}'}\ \overline{\mathtt{x}})\ \{\mathtt{return}\ \mathtt{e};\}\ \text{ok in}\ \mathbb{D}<\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}>.\mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}>}\,[\text{TMethod}]$$

$$\frac{\begin{array}{c} MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}<\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}> \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \quad \mathtt{class}\ \mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}> \triangleleft \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{K}}\ \ \overline{\mathtt{M}}\} \in \overline{\mathtt{CL}} \\ \mathtt{N} = \mathbb{V}.\mathtt{D}<\overline{\mathtt{S}}> \qquad \mathit{bound}_{\Delta}(\mathbb{V}) = \mathbb{C}<\overline{\mathbb{Z}}> \\ \Delta = \overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}} + \mathtt{thisMod} \triangleleft\ \mathbb{D}<\overline{\mathbb{Y}}> + \overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}} \quad \Gamma = \overline{\mathtt{x}} : \overline{\mathtt{V}} \quad \overline{\mathtt{x}} \cap \mathtt{this} = \emptyset \\ \Delta \vdash \overline{\mathtt{V}}\ \text{ok} \quad \Delta; \Gamma \vdash \overline{\mathtt{e}'} \in \overline{\mathtt{U}'} \quad \vdash \mathbb{C}<\overline{\mathbb{Z}}>.\mathtt{D}<\overline{\mathtt{S}}>\ \mathit{includes}\ \mathtt{init}(\overline{\mathtt{U}'}) \quad \Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{U}} \quad \Delta \vdash \overline{\mathtt{U}} <: \overline{\mathtt{T}} \end{array}}{\mathtt{C}(\overline{\mathtt{V}}\ \overline{\mathtt{x}})\{\mathtt{super}(\overline{\mathtt{e}'});\mathtt{this}.\overline{\mathtt{f}}\ \mathtt{=}\ \overline{\mathtt{e}};\}\ \text{ok in}\ \mathbb{D}<\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}>.\mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}>}\,[\text{TConstructor}]$$

$$\frac{\begin{array}{c} MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}<\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}> \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \overline{\mathtt{K}}\ \text{ok in}\ \mathbb{D}<\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}>.\mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}> \quad \overline{\mathtt{M}}\ \text{ok in}\ \mathbb{D}<\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}>.\mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}> \\ \Delta = \overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}} + \mathtt{thisMod} \triangleleft\ \mathbb{D}<\overline{\mathbb{Y}}> + \overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}} \quad \Delta \vdash \overline{\mathtt{R}}\ \text{ok} \quad \Delta \vdash \overline{\mathtt{R}} <: \mathtt{Object} \quad \Delta \vdash \mathtt{N}\ \text{ok} \\ \Delta \vdash \overline{\mathtt{T}}\ \text{ok} \quad \overline{\mathtt{X}} \cap \mathtt{thisMod} = \emptyset \quad \overline{\mathtt{f}} \cap \mathtt{this} = \emptyset \\ \Delta \vdash \mathbb{D}<\overline{\mathbb{Y}}>.\mathtt{C}<\overline{\mathtt{X}}> <: \mathtt{V}\ \text{and}\ \Delta \vdash \mathit{fields}(\mathtt{V}) = \overline{\mathtt{T}'}\ \overline{\mathtt{f}'}\ \text{implies}\ \mathtt{f} \cap \mathtt{f}' = \emptyset \\ \mathtt{K}_i = \mathtt{C}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\ldots\}\ \text{and}\ \mathtt{K}_j = \mathtt{C}(\overline{\mathtt{T}}\ \overline{\mathtt{x}'})\ \{\ldots\}\ \text{implies}\ i = j \end{array}}{\mathtt{class}\ \mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}> \triangleleft \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{K}}\ \ \overline{\mathtt{M}}\}\ \text{ok in}\ \mathbb{D}<\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}>}\,[\text{TClass}]$$

$$\frac{\Delta_0 \vdash \mathbb{N}\ \text{ok} \quad \Delta_0 \vdash \mathbb{N} <: \mathbb{N}_0}{\mathtt{bind}\ \mathbb{N}_0\ \mathbb{X}\ \ \mathtt{=}\ \mathbb{N};\ \text{ok}}\,[\text{TBind}] \qquad \frac{\begin{array}{c} \overline{\mathtt{CL}}\ \text{ok in}\ \mathbb{D}<\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}> \quad \overline{\mathbb{N}} <: \mathtt{Mod} \quad \overline{\mathbb{Y}} \cap \mathtt{thisMod} = \emptyset \\ \Delta = \overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}} \quad \Delta \vdash \overline{\mathbb{N}}\ \text{ok} \quad \Delta \vdash \mathbb{N}\ \text{ok} \quad \Delta \vdash \mathit{classOverride}(\mathbb{N}, \overline{\mathtt{CL}}) \\ \mathtt{CL}_i = \mathtt{class}\ \mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{T}}> \triangleleft \mathtt{N}\ \{\ldots\}\ \text{and}\ \mathtt{CL}_j = \mathtt{class}\ \mathtt{C}<\overline{\mathtt{X}'} \triangleleft \overline{\mathtt{T}'}> \triangleleft \mathtt{N}'\ \{\ldots\} \\ \text{implies}\ i = j \end{array}}{\mathtt{module}\ \mathbb{D}<\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}> \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\}\ \text{ok}}\,[\text{TModule}]$$

Table 3. Well-formed Declarations

constructors match the required signature of a new expression.

In order to allow for a subject-reduction theorem over the CCG small-step semantics, it is necessary to provide separate typing rules for annotated field lookup and method invocation expressions. Notice that it is not possible 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 type annotations play a crucial rule in preserving information in the computation rules, they play an analogous role in typing expressions during computation.

In FGJ, "stupid casts" (the casting of an expression to an incompatible type) were identified as a possible result during subject reduction. In CCG, it is not possible to statically detect "stupid casts" in modules because class types cannot be completely resolved until module linking at run-time. For the sake of brevity, all casts are accepted during typing.

To avoid the complications of matching multiple constructors of an object, CCG requires an exact match between the parameter types of a constructor and the static types of the provided arguments. Casts can be inserted to coerce argument types.

A.8 Computation

The rules for Computation are contained in Figure 7. Computation is specified by a small-step semantics. 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. In contrast to FGJ and CMG, the small-step semantics for CGEN carry a

$$\Delta;\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})\ \text{[TVar]}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e} \in \mathtt{S} \qquad \Delta \vdash \mathtt{T}\ \text{ok}}{\Delta;\Gamma \vdash (\mathtt{T})\mathtt{e} \in \mathtt{T}}\ \text{[TUCast]}$$

$$\frac{\begin{array}{c} \mathit{bound}_\Delta(\mathbb{V}) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \\ \vdash \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{T}}\mathtt{>}\ \mathit{includes}\ \mathtt{init}(\overline{\mathtt{S}}) \end{array}}{\Delta;\Gamma \vdash \mathtt{new}\ \mathbb{V}\mathtt{.C<}\overline{\mathtt{T}}\mathtt{>}(\overline{\mathtt{e}}) \in \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{T}}\mathtt{>}\ \mathit{annotate}\ [\overline{\mathtt{e}} :: \overline{\mathtt{S}}]}\ \text{[TNew]}$$

$$\frac{\begin{array}{c} \Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{N} \qquad \mathtt{N} = \mathbb{V}\mathtt{.C<}\overline{\mathtt{U}}\mathtt{>} \\ \mathit{bound}_\Delta(\mathbb{V}) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>} \qquad \Delta \vdash \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \Delta \vdash \mathtt{P} <: \mathtt{N} \text{ and } \mathtt{f}_i \in (\Delta \vdash \mathit{fields}(\mathtt{P})) \text{ implies } \mathtt{P} = \mathtt{N} \end{array}}{\Delta;\Gamma \vdash \mathtt{e.f}_i \in \mathtt{T}_i\ \mathit{annotate}\ [\mathtt{e} :: \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{U}}\mathtt{>}]}\ \text{[TField]}$$

$$\frac{\begin{array}{c} \Delta \vdash \overline{\mathtt{T}}\ \text{ok} \qquad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{R}} \\ \mathit{bound}_\Delta(\mathtt{T}_0) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{S}}\mathtt{>} \\ \Delta \vdash \mathit{mtype}(\mathtt{m}, \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{S}}\mathtt{>}) = \mathtt{P.<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}\ \mathtt{S}\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}}) \\ \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{N}} \qquad \Delta \vdash \overline{\mathtt{R}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}} \end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{T}}\mathtt{>}(\overline{\mathtt{e}}) \in [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{S}\ \mathit{annotate}\ [\mathtt{e}_0 \in \mathtt{P}]}\ \text{[TInv]}$$

$$\frac{\begin{array}{c} \mathit{bound}_\Delta(\mathbb{V}) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>} \qquad \Delta \vdash \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{R}} \qquad \Delta \vdash \overline{\mathtt{R}} <: \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}}\ \text{ok} \\ \vdash \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{T}}\mathtt{>}\ \mathit{includes}\ \mathtt{init}(\overline{\mathtt{S}}) \end{array}}{\Delta;\Gamma \vdash \mathtt{new}\ \mathbb{V}\mathtt{.C<}\overline{\mathtt{T}}\mathtt{>}(\overline{\mathtt{e}} :: \overline{\mathtt{S}}) \in \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{T}}\mathtt{>}}\ \text{[TAnnNew]}$$

$$\frac{\begin{array}{c} \Delta \vdash \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Delta \vdash \mathtt{N}\ \text{ok} \\ \Delta;\Gamma \vdash \mathtt{e} \in \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{N} \end{array}}{\Delta;\Gamma \vdash [\mathtt{e} :: \mathtt{N}]\mathtt{.f}_i \in \mathtt{T}_i}\ \text{[TAnnField]}$$

$$\frac{\begin{array}{c} \Delta \vdash \overline{\mathtt{T}}\ \text{ok} \qquad \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{R}} \\ \Delta \vdash \mathit{mtype}(\mathtt{m}, \mathtt{O}) = \mathtt{P.<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}\ \mathtt{S}\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}}) \\ \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{N}} \qquad \Delta \vdash \overline{\mathtt{R}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\overline{\mathtt{U}} \end{array}}{\Delta;\Gamma \vdash [\mathtt{e}_0 \circ \mathtt{O}]\mathtt{.m<}\overline{\mathtt{T}}\mathtt{>}(\overline{\mathtt{e}}) \in [\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}]\mathtt{S}}\ \text{[TAnnInv]}$$

Table 4. Expression Typing

bound environment Δ during computation representing the available bind declarations.

When evaluating 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 $\in \mathtt{T}$. This form of annotation is kept until no further reduction of the static type is possible. At this 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 are 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.

A.9 Type Soundness

A full proof of CCG type soundness is available in an accompanying technical report. The notion of subject reduction is formalized in the following theorem:

THEOREM 1 (Subject Reduction). *If* $\Delta \vdash \mathtt{e} \in \mathtt{T}$ *and* $\Delta \vdash \mathtt{e} \rightarrow \mathtt{e}'$ *then* $\Delta \vdash \mathtt{e}' \in \mathtt{S}$ *where* $\Delta \vdash \mathtt{S} <: \mathtt{T}$.

We also state a progress theorem, which relies on the following definitions:

DEFINITION 1 (Value). *A well-typed expression* e *is a* ***value*** *iff* e *is of the form* $\mathtt{new}\ \mathbb{T}\mathtt{.C<}\overline{\mathtt{T}}\mathtt{>}(\overline{\mathtt{e}})$ *where* $\mathit{bound}_\Delta(\mathbb{T}) = \mathbb{D}\mathtt{<}\overline{\mathbb{V}}\mathtt{>}$ *and all* $\overline{\mathtt{e}}$ *are values.*

DEFINITION 2 (Bad Cast). *A well-typed expression* e *is a* ***bad cast*** *iff* e *is of the form* $(\mathtt{T})\mathtt{e}'$ *where* $\Delta \vdash \mathtt{e}' \in \mathtt{S}$ *and* $\Delta \not\vdash \mathtt{S} <: \mathtt{T}$.

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

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

THEOREM 2 (Progress). *For program* $(\mathit{MT}, \mathit{BT}, \mathtt{e})$ *s.t.* $\Delta_{BT} \vdash \mathtt{e} \in \mathtt{R}$, *if* $\Delta_{BT} \vdash \mathtt{e} \xrightarrow{*} \mathtt{e}'$ *then either* $\mathtt{e}'$ *is a value,* $\mathtt{e}'$ *contains a bad cast, or there exists* $\mathtt{e}''$ *s.t.* $\Delta_{BT} \vdash \mathtt{e}' \rightarrow \mathtt{e}''$.

THEOREM 3 (Type Soundness). *For program* $(\mathit{MT}, \mathit{BT}, \mathtt{e})$ *s.t.* $\Delta_{BT} \vdash \mathtt{e} \in \mathtt{T}$, *evaluation of* $(\mathit{MT}, \mathit{BT}, \mathtt{e})$ *yields one of the following results:*

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

$$\Delta \vdash \mathit{fields}(\mathtt{Object}) = \bullet$$

$$\frac{\begin{array}{c}\mathit{bound}_{\Delta}(\mathbb{V}) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>} \quad MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\mathtt{>} \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{S}}\mathtt{>} \triangleleft \mathtt{U}\ \{\overline{\mathtt{T}}\,\overline{\mathtt{f}};\ \overline{\mathtt{K}}\ \overline{\mathtt{M}}\} \in \overline{\mathtt{CL}}\end{array}}{\Delta \vdash \mathit{fields}(\mathbb{V}\mathtt{.C<}\overline{\mathtt{R}}\mathtt{>}) = [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\overline{\mathtt{T}}\,\overline{\mathtt{f}}}$$

$$\Delta \vdash \mathit{fieldVals}(\mathtt{new\ Object()}, \mathtt{Object}) = \bullet$$

$$\frac{\begin{array}{c}\mathit{bound}_{\Delta}(\mathbb{V}) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>} \quad \mathit{bound}_{\Delta}(\mathtt{N}) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{R}}\mathtt{>} \\ MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\mathtt{>} \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{S}}\mathtt{>} \triangleleft \mathtt{U}\ \{\ldots\mathtt{C(}\overline{\mathtt{T}}\,\overline{\mathtt{x}}\mathtt{)}\ \{\mathtt{super(}\overline{\mathtt{e}}\mathtt{);this.}\overline{\mathtt{f}\ \mathtt{=}\ \mathtt{e}'}\mathtt{;}\}\ldots\} \in \overline{\mathtt{CL}}\end{array}}{\Delta \vdash \mathit{fieldVals}(\mathtt{new}\ \mathbb{V}\mathtt{.C<}\overline{\mathtt{R}}\mathtt{>(}\overline{\mathtt{e}''} :: \overline{\mathtt{T}}\mathtt{)}, \mathtt{N}) = [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}][\overline{\mathtt{x}} \mapsto \overline{\mathtt{e}''}]\overline{\mathtt{e}'}}$$

$$\frac{\begin{array}{c}\mathit{bound}_{\Delta}(\mathbb{V}) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>} \quad MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\mathtt{>} \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{S}}\mathtt{>} \triangleleft \mathtt{U}\ \{\ldots\mathtt{C(}\overline{\mathtt{T}}\,\overline{\mathtt{x}}\mathtt{)}\ \{\mathtt{super(}\overline{\mathtt{e}}\mathtt{);this.}\overline{\mathtt{f}\ \mathtt{=}\ \mathtt{e}'}\mathtt{;}\}\ldots\} \in \overline{\mathtt{CL}} \\ \overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}} + \overline{\mathtt{X}} \triangleleft \overline{\mathtt{S}}; \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \overline{\mathtt{e}} \in \overline{\mathtt{V}} \quad \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{R}}\mathtt{>} \neq \mathtt{N} \\ \Delta \vdash \mathit{fieldVals}(\mathtt{new}\ [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\mathtt{U(}[\overline{\mathtt{x}} \mapsto \overline{\mathtt{e}''}]\overline{\mathtt{e}} :: \overline{\mathtt{V}}\mathtt{)}, \mathtt{N}) = \mathtt{e}'''\end{array}}{\Delta \vdash \mathit{fieldVals}(\mathtt{new}\ \mathbb{V}\mathtt{.C<}\overline{\mathtt{R}}\mathtt{>(}\overline{\mathtt{e}''} :: \overline{\mathtt{S}}\mathtt{)}, \mathtt{N}) = \mathtt{e}'''}$$

$$\frac{\begin{array}{c}\mathit{bound}_{\Delta}(\mathbb{V}) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>} \quad MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\mathtt{>} \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{T}\ \{\overline{\mathtt{T}}\,\overline{\mathtt{f}};\ \overline{\mathtt{K}}\ \overline{\mathtt{M}}\} \in \overline{\mathtt{CL}} \\ \mathtt{<}\overline{\mathtt{Y}'} \triangleleft \overline{\mathtt{N}'}\mathtt{>}\ \mathtt{T}'\ \mathtt{m(}\overline{\mathtt{R}}\,\overline{\mathtt{x}}\mathtt{)}\ \{\mathtt{return\ e;}\} \in \overline{\mathtt{M}}\end{array}}{\Delta \vdash \mathit{mtype}(\mathtt{m}, \mathbb{V}\mathtt{.C<}\overline{\mathtt{U}}\mathtt{>}) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{U}}\mathtt{>.}[\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}](\mathtt{<}\overline{\mathtt{Y}'} \triangleleft \overline{\mathtt{N}'}\mathtt{>}\ \mathtt{T}'\ \mathtt{m(}\overline{\mathtt{R}}\,\overline{\mathtt{x}}\mathtt{)})}$$

$$\frac{\begin{array}{c}\mathit{bound}_{\Delta}(\mathbb{V}) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>} \quad MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\mathtt{>} \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{S}}\mathtt{>} \triangleleft \mathtt{T}\ \{\overline{\mathtt{T}}\,\overline{\mathtt{f}};\ \overline{\mathtt{K}}\ \overline{\mathtt{M}}\} \in \overline{\mathtt{CL}} \\ \mathtt{m\ is\ not\ defined\ in}\ \overline{\mathtt{M}}\end{array}}{\Delta \vdash \mathit{mtype}(\mathtt{m}, \mathbb{V}\mathtt{.C<}\overline{\mathtt{U}}\mathtt{>}) = \Delta \vdash \mathit{mtype}(\mathtt{m}, [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{T})}$$

$$\frac{\begin{array}{c}\mathit{bound}_{\Delta}(\mathbb{V}) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>} \quad MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\mathtt{>} \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{S}}\mathtt{>} \triangleleft \mathtt{T}\ \{\overline{\mathtt{T}}\,\overline{\mathtt{f}};\ \overline{\mathtt{K}}\ \overline{\mathtt{M}}\} \in \overline{\mathtt{CL}} \\ \mathtt{<}\overline{\mathtt{Y}'} \triangleleft \overline{\mathtt{S}'}\mathtt{>}\ \mathtt{T}'\ \mathtt{m(}\overline{\mathtt{R}}\,\overline{\mathtt{x}}\mathtt{)}\ \{\mathtt{return\ e;}\} \in \overline{\mathtt{M}}\end{array}}{\Delta \vdash \mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathbb{V}\mathtt{.C<}\overline{\mathtt{U}}\mathtt{>}) = (\overline{\mathtt{x}}, [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}][\overline{\mathtt{Y}'} \mapsto \overline{\mathtt{V}}]\mathtt{e})}$$

$$\frac{\begin{array}{c}\mathit{bound}_{\Delta}(\mathbb{V}) = \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>} \quad MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\mathtt{>} \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{S}}\mathtt{>} \triangleleft \mathtt{T}\ \{\overline{\mathtt{T}}\,\overline{\mathtt{f}};\ \overline{\mathtt{K}}\ \overline{\mathtt{M}}\} \in \overline{\mathtt{CL}} \\ \mathtt{m\ is\ not\ defined\ in}\ \overline{\mathtt{M}}\end{array}}{\Delta \vdash \mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathbb{V}\mathtt{.C<}\overline{\mathtt{U}}\mathtt{>}) = \mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{T})}$$

$$\frac{\begin{array}{c}\Delta \vdash \mathit{mtype}(\mathtt{m}, \mathtt{N}) = \mathtt{P.<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{T}}\mathtt{>}\ \mathtt{R}\ \mathtt{m(}\overline{\mathtt{U}}\,\overline{\mathtt{x}}\mathtt{)}\ \text{implies} \\ \overline{\mathtt{T}'}, \overline{\mathtt{U}'} = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{Y}}](\overline{\mathtt{T}}, \overline{\mathtt{U}})\ \text{and}\ \Delta + \overline{\mathtt{Y}} \triangleleft \overline{\mathtt{T}'} \vdash \mathtt{R}' <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{Y}}]\mathtt{R}\end{array}}{\Delta \vdash \mathit{override}(\mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{T}'}\mathtt{>}\ \mathtt{R}'\ \mathtt{m(}\overline{\mathtt{U}'}\,\overline{\mathtt{x}}\mathtt{)})}$$

$$\vdash \mathtt{Object}\ \mathit{includes}\ \mathtt{init()}$$

$$\frac{\begin{array}{c}MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\mathtt{>} \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{S}}\mathtt{>} \triangleleft \mathtt{S}\ \{\ldots\mathtt{C(}\overline{\mathtt{T}}\,\overline{\mathtt{x}}\mathtt{)}\ \{\ldots\}\ldots\} \in \overline{\mathtt{CL}}\end{array}}{\vdash \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{R}}\mathtt{>}\ \mathit{includes}\ [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\mathtt{init(}\overline{\mathtt{T}}\mathtt{)}}$$

$$\frac{\begin{array}{c}MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\mathtt{>} \triangleleft \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{S}}\mathtt{>} \triangleleft \mathtt{T}\ \{\ldots\mathtt{M}\ldots\} \in \overline{\mathtt{CL}} \\ \mathtt{M} = \mathtt{<}\overline{\mathtt{X}'} \triangleleft \overline{\mathtt{S}'}\mathtt{>}\ \mathtt{T}'\ \mathtt{m(}\overline{\mathtt{R}'}\,\overline{\mathtt{x}}\mathtt{)}\ \{\mathtt{return\ e;}\}\end{array}}{\vdash \mathbb{D}\mathtt{<}\overline{\mathbb{Z}}\mathtt{>.C<}\overline{\mathtt{R}}\mathtt{>}\ \mathit{includes}\ [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\mathtt{<}\overline{\mathtt{X}'} \triangleleft \overline{\mathtt{S}'}\mathtt{>}\ \mathtt{T}'\ \mathtt{m(}\overline{\mathtt{R}'}\,\overline{\mathtt{x}}\mathtt{)}}$$

Table 5. Class Auxiliary Functions

$$\frac{\mathbb{D}\texttt{<}\overline{\mathbb{Y}'}\texttt{>.C<}\overline{\texttt{T}'}\texttt{>}\ \textit{includes}\ \texttt{init(}\overline{\texttt{T}}\texttt{)}}{\vdash \mathbb{D}\texttt{<}\overline{\mathbb{Y}'}\texttt{>.C<}\overline{\texttt{T}'}\texttt{>}\ \textit{provides}\ \texttt{C(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)}\ \{\ldots\}}$$

$$\frac{\mathbb{D}\texttt{<}\overline{\mathbb{Y}'}\texttt{>.C<}\overline{\texttt{T}'}\texttt{>}\ \textit{includes}\ \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{T}'}\texttt{>}\ \texttt{R}'\ \texttt{m(}\overline{\texttt{U}'}\ \overline{\texttt{x}}\texttt{)}}{\vdash \mathbb{D}\texttt{<}\overline{\mathbb{Y}'}\texttt{>.C<}\overline{\texttt{T}'}\texttt{>}\ \textit{provides}\ \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{T}'}\texttt{>}\ \texttt{R}'\ \texttt{m(}\overline{\texttt{U}'}\ \overline{\texttt{x}}\texttt{)}\ \{\texttt{return e;}\}}$$

$$\frac{\begin{array}{c} MT(\mathbb{D}) = \texttt{module}\ \mathbb{D}\texttt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\texttt{>} \triangleleft \mathbb{N}\ \{\overline{\texttt{CL}}\}\ \text{and} \\ \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{S}}\texttt{>} \triangleleft \texttt{U}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{K}}\ \ \overline{\texttt{M}}\} \in \overline{\texttt{CL}}\ \text{implies} \\ (\overline{\texttt{S}'}, \texttt{U}') = [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Y}'}][\overline{\texttt{X}} \mapsto \overline{\texttt{X}'}](\overline{\texttt{S}}, \texttt{U})\ \text{and}\ [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Y}'}][\overline{\texttt{X}} \mapsto \overline{\texttt{X}'}]\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;} \subseteq \overline{\texttt{T}'}\ \overline{\texttt{f}'}\texttt{;} \\ \text{and}\ \mathbb{D}\texttt{<}\overline{\mathbb{Y}'}\texttt{>.C<}\overline{\texttt{X}'}\texttt{>}\ \textit{provides}\ [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Y}'}][\overline{\texttt{X}} \mapsto \overline{\texttt{X}'}](\overline{\texttt{K}}, \overline{\texttt{M}}) \end{array}}{\Delta \vdash \textit{classOverride}(\mathbb{D}\texttt{<}\overline{\mathbb{Y}'}\texttt{>}, \texttt{class C<}\overline{\texttt{X}'} \triangleleft \overline{\texttt{S}'}\texttt{>} \triangleleft \texttt{U}'\ \{\overline{\texttt{T}'}\ \overline{\texttt{f}'}\texttt{;}\ \overline{\texttt{K}'}\ \ \overline{\texttt{M}'}\})}$$

Table 6. Module Auxiliary Functions

$$\frac{\Delta \vdash \textit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{N}) = (\overline{\texttt{x}}, \texttt{e}_0)}{\begin{array}{c}\Delta \vdash \texttt{[new}\ \mathbb{V}\texttt{.C<}\overline{\texttt{S}}\texttt{>(}\overline{\texttt{e}} :: \texttt{P)} :: \texttt{N].m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{d}}\texttt{)} \longrightarrow \\ [\overline{\texttt{x}} \mapsto \overline{\texttt{d}}][\texttt{this} \mapsto \texttt{new}\ \mathbb{V}\texttt{.C<}\overline{\texttt{S}}\texttt{>(}\overline{\texttt{e}} :: \overline{\texttt{P}}\texttt{)}]\texttt{e}_0\end{array}}\ \text{[RInv]}$$

$$\frac{\begin{array}{c}\Delta \vdash \texttt{e} \in \texttt{N} \quad \Delta \vdash \texttt{N} <: \mathbb{V}\texttt{.C<}\overline{\texttt{U}}\texttt{>} \quad \textit{bound}_\Delta(\mathbb{V})\ \texttt{=}\ \mathbb{D}\texttt{<}\overline{\mathbb{T}}\texttt{>} \\ \textit{MT}(\mathbb{D}) = \texttt{module}\ \mathbb{D}\texttt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\texttt{>} \triangleleft \mathbb{N}\ \{\overline{\texttt{CL}}\} \\ \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{S}}\texttt{>} \triangleleft \texttt{T}\ \{\ldots\} \in \overline{\texttt{CL}} \\ \Delta \vdash \textit{mtype}(\texttt{m}, \mathbb{D}\texttt{<}\overline{\mathbb{Z}}\texttt{>.C<}\overline{\texttt{U}}\texttt{>}) = \textit{mtype}(\texttt{m}, [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\texttt{X}} \mapsto \overline{\texttt{U}}]\texttt{T})\end{array}}{\Delta \vdash \texttt{[e} \in [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\texttt{X}} \mapsto \overline{\texttt{U}}]\texttt{T].m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{d}}\texttt{)} \longrightarrow \texttt{[e} \in \mathbb{D}\texttt{<}\overline{\mathbb{Z}}\texttt{>.C<}\overline{\texttt{U}}\texttt{>].m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{d}}\texttt{)}}\ \text{[RInvSub]}$$

$$\frac{\begin{array}{c}\Delta \vdash \texttt{e} \in \texttt{N} \quad \Delta \vdash \texttt{N} <: \mathbb{V}\texttt{.C<}\overline{\texttt{U}}\texttt{>} \quad \textit{bound}_\Delta(\mathbb{V})\ \texttt{=}\ \mathbb{D}\texttt{<}\overline{\mathbb{T}}\texttt{>} \\ \textit{MT}(\mathbb{D}) = \texttt{module}\ \mathbb{D}\texttt{<}\overline{\mathbb{Y}} \triangleleft \overline{\mathbb{N}}\texttt{>} \triangleleft \mathbb{N}\ \{\overline{\texttt{CL}}\} \\ \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{S}}\texttt{>} \triangleleft \texttt{T}\ \{\ldots\} \in \overline{\texttt{CL}} \\ \Delta \vdash \textit{mtype}(\texttt{m}, \mathbb{D}\texttt{<}\overline{\mathbb{Z}}\texttt{>.C<}\overline{\texttt{U}}\texttt{>})\ \text{is undefined or} \\ \Delta \vdash \textit{mtype}(\texttt{m}, \mathbb{D}\texttt{<}\overline{\mathbb{Z}}\texttt{>.C<}\overline{\texttt{U}}\texttt{>}) \neq \textit{mtype}(\texttt{m}, [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\texttt{X}} \mapsto \overline{\texttt{U}}]\texttt{T})\end{array}}{\Delta \vdash \texttt{[e} \in [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\texttt{X}} \mapsto \overline{\texttt{U}}]\texttt{T].m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{d}}\texttt{)} \longrightarrow \texttt{[e} :: [\overline{\mathbb{Y}} \mapsto \overline{\mathbb{Z}}][\overline{\texttt{X}} \mapsto \overline{\texttt{U}}]\texttt{T].m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{d}}\texttt{)}}\ \text{[RInvStop]}$$

$$\frac{\Delta \vdash \texttt{N} <: \texttt{T}}{\Delta \vdash \texttt{(T)new N(}\overline{\texttt{e}} :: \overline{\texttt{S}}\texttt{)} \longrightarrow \texttt{new N(}\overline{\texttt{e}} :: \overline{\texttt{S}}\texttt{)}}\ \text{[RCast]} \qquad \frac{\begin{array}{c}\Delta \vdash \textit{fields}(\texttt{R}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \\ \Delta \vdash \textit{fieldVals}(\texttt{new N(}\overline{\texttt{e}}\texttt{)}, \texttt{R}) = \overline{\texttt{e}'}\end{array}}{\Delta \vdash \texttt{[new N(}\overline{\texttt{e}}\texttt{)} :: \texttt{R].f}_i \longrightarrow \texttt{e}'_i}\ \text{[RField]}$$

$$\frac{\Delta \vdash \texttt{e}_i \longrightarrow \texttt{e}_i{}'}{\Delta \vdash \texttt{new T(}\ldots, \texttt{e}_i :: \texttt{S}, \ldots\texttt{)} \longrightarrow \texttt{new T(}\ldots, \texttt{e}_i{}' :: \texttt{S}, \ldots\texttt{)}}\ \text{[RCNewArg]}$$

$$\frac{\Delta \vdash \texttt{e} \longrightarrow \texttt{e}'}{\Delta \vdash \texttt{[e} \circ \texttt{N].m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{d}}\texttt{)} \longrightarrow \texttt{[e}' \circ \texttt{N].m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{d}}\texttt{)}}\ \text{[RCInvRecv]}$$

$$\frac{\Delta \vdash \texttt{e}_i \longrightarrow \texttt{e}_i{}'}{\Delta \vdash \texttt{[e} \circ \texttt{N].m<}\overline{\texttt{V}}\texttt{>(}\ldots\texttt{e}_i\ldots\texttt{)} \longrightarrow \texttt{[e} \circ \texttt{N].m<}\overline{\texttt{V}}\texttt{>(}\ldots\texttt{e}_i{}'\ldots\texttt{)}}\ \text{[RCInvArg]}$$

$$\frac{\Delta \vdash \texttt{e} \longrightarrow \texttt{e}'}{\Delta \vdash \texttt{((S)e)} \longrightarrow \texttt{((S)e}'\texttt{)}}\ \text{[RCCast]} \qquad \frac{\Delta \vdash \texttt{e} \longrightarrow \texttt{e}'}{\Delta \vdash \texttt{[e} :: \texttt{R].f} \longrightarrow \texttt{[e}' :: \texttt{R].f}}\ \text{[RCField]}$$

Table 7. Computation