RICE UNIVERSITY

Component NextGen: A Sound and Expressive Component Framework for Java

by

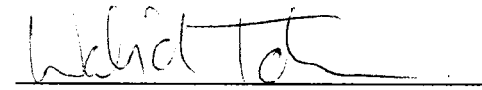
James Sasitorn

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

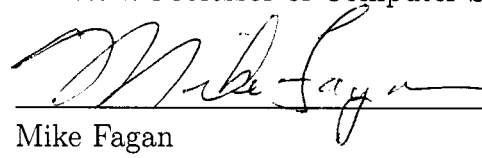
Doctor of Philosophy

Approved, Thesis Committee:

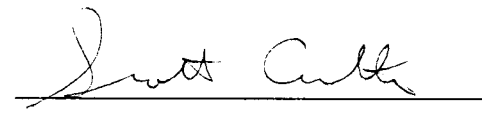
Robert Cartwright, Chair
Professor of Computer Science



Walid Taha
Assistant Professor of Computer Science



Mike Fagan
Faculty Fellow,
Computational and Applied Mathematics



Scott Cutler
Adjunct Faculty,
Electrical and Computer Engineering

Houston, Texas

April, 2007

Component NextGen: A Sound and Expressive Component Framework for Java

James Sasitorn

Abstract

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 by the absence of a component system for decomposing applications into independent units with statically checked interfaces.

Developing a general component system for an object-oriented language, such as Java, is a challenging design problem because inheritance across component boundaries can cause accidental method overrides. In addition, 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.

This thesis discusses how a component framework can be constructed for a nominally typed object-oriented language supporting *first-class* generic types simply by adding appropriate annotations and syntactic sugar. 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 generates code that can be executed on current Java Virtual Machines.

Acknowledgments

I would like to thank my advisor, Professor Robert "Corky" Cartwright, for his guidance and insight. He embodies an uncanny fusion of both the theoretical and practical aspects of software development. He gave me considerable freedom and respect in my in ideas and research.

I also would like to thank Professor Walid Taha for introducing me to the more theoretical aspects of programming language semantics. My experience with him has helped me attain a heightened level of precision in logic.

I've also had the privilege of working with many talented individuals at Rice. In particular, I would like to thank everyone in the Rice JavaPLT team, most notably Moez Abdel-Gawad, Mathias Gricken, and Michael Jensen for our many insightful discussions on NEXTGEN and software development. I would like to thank my long time friends, and Computer Science classmates, Joann Chuang and Dennis Lu, who played an important role during my growth as a Rice undergraduate.

I'd also like to thank the members of my M.S. and Ph.D. thesis committees for their feedback and hard work: Corky Cartwright, Walid Taha, Mike Fagan, Scott Cutler, and Keith Cooper. I would like to thank Keith Cooper, the chair of the Computer Science department for his generous help before and after my Ph.D. defense.

I'd like to thank the administrative staff at Rice who play an instrumental role in getting things done — and in particular, before the deadlines: Iva Jean Jorgensen, BJ Smith, Belia Martinez, Melissa Cisneros, Rhonda Guajarado, and Darnell Price.

I am very thankful to my parents, Starporn and Wanicha Sasitorn, as well as my brother Kjohn Sasitorn for their support during not only this arduous endeavor, but my entire life.

Finally, I'd like to thank Elisa Medina Sisniega for her friendship, love, and insight.

Contents

Illustrations

Tables

Chapter 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 [22] incorporated in Java 5.0), and more comprehensively by the programming research community (GJ [10], PolyJ[27], NextGen [11], LM [39, 38]) 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 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[15, 20, 35].

Developing a general component system for an object-oriented language is a challenging problem because 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 [36, 35]. This "accidental method capture" problem inhibits using classes as components because a component class often contains methods beyond those required in a particular program. In addition, mutually recursive references across components can produce degenerate type hierarchies.

In this thesis we show how a component framework can be constructed for a nominally typed object-oriented language supporting *first-class* generic types simply by adding appropriate annotations and syntactic sugar. The fundamental semantic building blocks for constructing, type-checking and manipulating components are provided by the underlying first-class generic type system. We show the resulting type system is sound for a core subset of Java supporting components. Moreover, we show how this extension can be efficiently implemented on top of the existing Java Virtual Machines. Although our work focuses on Java, the same analysis could be applied to any object-oriented language supporting incremental compilation, dynamic class loading, and a static type system with nominal subtyping.

1.1 Background

Before we discuss how a nominally typed object-oriented language supporting *first-class* generic types can be easily extended to support components, we begin with with a brief explanation of some key concepts underlying our approach.

1.1.1 Nominal versus Structural Typing

Mainstream object-oriented languages, namely Java, C#, and C++, rely on *nominal* typing rather than *structural* typing, which has been the focus on most of 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`. We are using the term inheritance more broadly than simply inheriting code. In Java, a class inherits code from its super class, using *extends*. But in our terminology, a Java class inherits from all of its interfaces as well as its superclass. In this world, 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 `A` includes all of the member names in `B` and 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.

To illustrate the differences between nominal and structural subtyping, consider the following Java classes:

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

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

Since Java is based on nominally typing, no subtyping relationship exists between the classes `MultiSet` and `Set`. In contrast, suppose Java supported structural typing. Then the class `MultiSet` would be a subtype of the class `Set` and vice-versa because the two classes contain the same public members with identical signatures. Structural typing creates subtyping relationships between class types on the basis of matching method signatures even when member contracts conflict.

1.2 Generic Types

The NEXTGEN language is a generalization of Java 5.0 that efficiently supports first-class generic types[6, 33]. 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 parametric and pure parametric class and array types, *e.g.* `new Vector<T>`, `new T()`, and `new T[]`.

A extension of NEXTGEN called MIXGEN[2] probes the limits of first-class genericity by incorporating *hygienic* mixins in the NEXTGEN implementation architecture. A mixin is a class definition where a type paramter `T` serves as the super class:

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

The hygienic semantics for mixins in MIXGEN 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`.

1.2.1 Mixins

The *mixin* construct abstracts the process of class extension. As stated previously, a mixin defintion

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

looks just like a generic class definition except for the fact that the superclass is a type parameter. The primary technical issue in formulating the semantics of mixins is defining the meaning of a mixin instantiation `C<A>` when `A` includes a `public` method `m` with exactly the same name and type as a method *introduced* by the mixin `C`, *i.e.*, a method that does not appear in the bounding interface `I`. In the naive semantics for mixins, `C<A>` *accidentally overrides* the definition of `m` in `A`—breaking the functionality of `A`.

A *hygienic* semantics[15, 16, 2] for mixins eliminates this pathology by ensuring that all of the methods introduced by a mixin (methods other than those in the bounding interface `I`) have *new* names that do not collide with the names of any of the methods in the mixin superclass. CGEN renames every method introduced in a mixin instantiation (or ordinary subclass of the mixin instantiation) by prefixing it with the (mangled) name of the mixin superclass. Since interface-based method dispatches simply use the method names provided by the interface type, a forwarding method must be generated for every renamed method that belongs to an interface. Reference [2] provides a detailed discussion of the semantics and implementation of hygienic mixins in Java.

1.2.2 Components and Objects

Software developers generally recognize the value of building software from reusable components as much as is practicable. But the proper definition of components has been extensively discussed and debated in the literature. Based on my experience as a software developer, I find the definition in Szyperski's widely-cited book on the subject [35] 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 particular valid context (bindings of its dependencies), it is 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 conflict with one another.

According to Szyperski, an object:

- is a unit of instantiation

- has state, and
- encapsulates its state and behavior.

If an object is a unit of instantiation, it has a unique identity and must always be instantiated in its entirety. Typically, objects are instantiated using a constructor or object prototype. If an object has state, it can store changes unique to each instantiation. And despite these changes, each object can be uniquely distinguished. If an object encapsulates its state and behavior, it provides a "black box" for code that interacts with it. In other words, other code has no need to know the details of how things are accomplished.

Given the differences in the above definitions, its apparent that components are not exactly synonymous with objects. Obviously though, in an object-oriented language, such as Java, components will be represented by classes and objects. However, as I will show later, objects in Java cannot provide an intuitive component framework— even using design patterns or other coding methodologies.

1.3 Roadmap

In the sequel, we will show how to construct a true component framework for a nominally typed object-oriented language supporting *first-class* genericity.

The remainder of this thesis is organized as follows. Chapter 2 discusses the inadequacies of current Java technologies for decomposing applications into components. Chapter 3 outlines the design of the NEXTGEN compiler supporting first-class genericity. Chapter 4 provides performance benchmarks for NEXTGEN showing that it is possible to provide run-time support for generic types with little or no overhead. In Chapters 5 and 6, we describe the design and architecture of Component NEXTGEN(CGEN), an extension of NEXTGEN that provides explicit linguistic support for components. The CGEN component language is a natural extension of the NEXTGEN generic type system that supports the parameterization of packages by type (creating modules) and adds new annotations (package signatures) to the type system. Chapter 7, we discuss the underlying implementation of CGEN. By leveraging

first-class generic types in NEXTGEN, the construction of a hygienic component system in CGEN is reduced to adding the appropriate annotations (signatures), syntatic sugar (modules), and the corresponding component-level type checking.* CGEN does not require a secondary module language or additional steps between compilation and execution. CGEN is backwards compatible with existing libraries and executes on current JVMs. CGEN avoids method capture by supporting "hygienic" components based on "hygienic" mixins [15, 16, 2] that eliminate accidental method capture by systematically renaming methods. Chapter 8 analyzes the semantic properties of CGEN by introducing Core CGEN (CCG), a core subset of CGEN, and proving that it has a sound type system. Chapter 9 discusses related work on component systems for Java and similar languages. Finally, Chapter 10 discusses directions for work built on the results of this research project. A discussion of the organization of the CGEN project and code base is provided in Appendix A.

*Our actual implementation optimizes the naive translation based purely on syntactic sugar.

Chapter 2

Motivation for Components

Let us examine in more detail the support provided by the existing Java platform (Java 5.0) for software components.

2.1 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.* 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 2.1 and 2.2 shows an outline of the abstract syntax and recursive decent parser† 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 2.3 shows

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

†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/.

```
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 2.1 : 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 2.2 : Outline of Jam parser

the outline of an interpreter that references `jam.parser.Parser` and classes from `jam.ast`.

From a software engineering perspective, fully-qualified references, such as the reference to `jam.ast.Exception` in Figure 2.2 and `jam.parser.Parser` in Figure 2.3 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 exe-

```
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 2.3 : Outline of Jam Interpreter

cution of the package. A modification to classes in a package typically require recompilation of all code that depends on it.[‡]

These problems become more apparent when we consider alternative component implementations, *e.g.*, a bottom-up parser `jam.botparser.Parser`. In principle, we should be able to switch to this new parser by changing the import statement in Figure 2.3 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.

2.2 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 2.4 and 2.5. 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

[‡]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.

without modifications to its source code. But as a scheme for systematically eliminating explicit dependencies, it is unworkable. Since 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 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.[§]

2.3 JavaBeans

To provide more effective encapsulation of code units, the Java software community has developed the concept of JavaBeans. JavaBeans[21] 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 support persistence (serialization) so that the state of any bean can be saved and later reloaded. The classes `Parser` and `Interpreter` referred in Figures 2.2 and 2.3 can be converted into JavaBeans simply by making each class `implement` 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.

[§]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.

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

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

Figure 2.4 : Jam IParser interface

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

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

Figure 2.5 : Jam Interpreter using IParser

While the "beans" architecture was originally developed for GUI components[21], the architecture has been applied to other contexts. For example, the Enterprise JavaBeans framework is a beans architecture for transaction oriented applications in business enterprises. It shares the same core methodology and is subject to the same limitations.

Chapter 3

NEXTGEN Compiler Design

In this chapter, we discuss the design of the NEXTGEN compiler and how it supports first-class generics while maintaining compatibility with existing Java byte code and Java Virtual Machines (JVMs).

Generics provide the ability to paramterize classes, interfaces, and methods with respect to types. Prior to Java 5.0, the Java platform did not support generic (parameterized) types, preventing expression of many statically checkable program invariants within the type system and inhibiting the use of type abstraction. Numerous proposals were made by the research community on how to add generics to Java, and a few prototype implementations were built: e.g., the Mitchell-Agesen hetereogenous formulation[1], Pizza[28], GJ[10], and PolyJ[27]. Of these, only the GJ proposal for Generic Java by Bracha, Odersky, Stoutamire, and Wadler, which was very faithful to the spirit of the original Java language design, was embraced by the Java development team at Sun Microsystems and incorporated into JSR14[22].

NEXTGEN extends GJ to propagate parametric type information to the JVM run-time environment. NEXTGEN supports the same source language, Generic Java, but with greater expressiveness. In fact, GJ and NEXTGEN were designed in concert with one another [10, 11], so that NEXTGEN would be a graceful extension of GJ.

3.1 Generic Classes

In Generic Java, class types can be parameterized by type variables and code may use generic types in place of conventional, non-generic types. A generic type is defined to be either a type variable or an application of a generic class/interface name to type arguments that may also be generic. The syntax for the class name appearing in the header of a class definition

is generalized from:

```
Identifer
```

to

```
Identifier < TypeParameters >
```

where

```
TypeParameters → TypeParm | TypeParm, TypeParameters
TypeParm → TypeVar { TypeBound }
TypeBound → extends ClassType | implements InterfaceType
TypeVar → Identifier
```

and { } enclose optional phrases. Interface declarations are similarly generalized. In addition, the definition of ReferenceType is generalized from

```
ReferenceType → ClassOrInterfaceType | ArrayType
```

to

```
ReferenceType → ClassOrInterfaceType | ArrayType | TypeVar
TypeVar → Identifier
ClassOrInterfaceType → ClassOrInterface { <TypeParameters> }
ClassOrInterface → Identifier | ClassOrInterfaceType.Identifier
```

and the syntax for *new* operations now includes the additional phrase

```
new TypeVar ( { ArgumentList } )
```

The new generic `ReferenceType` can appear in any context a class or interface name can in ordinary Java except in the `extends` or `implements` clause of a class definition.

Type parameters introduced in the header of a generic class or interface declaration are in scope for the entire body of the declaration, including the bounding types of other type parameters.

In a generic type application, type parameters can be instantiated to any valid reference type that is a subtype of the declared bounding type. If the parameter does not explicitly declare a bounding type, the universal type `java.lang.Object` is assumed.

3.2 GJ Implementation Scheme

While the Generic Java language extension supported by the GJ compiler is simple and elegant, the underlying implementation is restrictive and introduces unexpected complications. The GJ compiler erases all of the parametric type information in the generate byte code. The generated code relies on casting operations (which can never fail in type correct programs). Each parametric class `C<T>` generates a single class file containing the erased base class `C`. The erasure of a parametric type `T` is obtained by replacing each type parameter `T` with its upper bound. All references to parametric classes or methods are replaced with references to their erased versions. For example, the class `Stack<T>` defined as

```
class Stack<T> extends Vector<T> {
   public T peek() { ... }
   ...
}
```

and all possible instantiations of it, would be represented using the erased code

```
class Stack extends Vector {
   public Object peek() { ... }
   ...
}
```

where all occurrences of the type parameter `T` are replaced by its upper bound (in this case implicit) `java.lang.Object`. All references to generic instantiations are similarly erased and casts are inserted to coerce erased types. For example, the code

```
Vector<String> v = ...;
String peek = v.peek();
```

would be translated into

```
Vector v = ...;
String peek = (String) v.peek();
```

3.3 Implications of GJ Type Erasure

Erasure of parametric type information creates the possibility of run-time type errors. Consider the generic class `Stack<T>` that extends a generic class `Vector<T>`. Intuitively an instantation `Stack<E>` should be a subtype of `Vector<E>`. Correct subtyping is possible under erasure; `Stack` is a subtype of `Vector`. However, by the same logic any instantiation class `Stack<F>` is also considered a subtype of `Vector<E>`.

A more dramatic problem occurs on operations involving "naked" parametric types such as `new T()` and `new T[]`. The type parameter `T` is erased to its upper bound, in this case `Object`. So these two operations are respectively erased to `new Object()` and `new Object[]`, respectively. This results in unexpected run-time behavior and exceptions.

To minimize the problems of erasure, GJ, and its descendent Java 5.0, prohibit certain operations that depend on run-time information. The prohibited type-dependent operations include:

- parametric `instanceof` tests;
- `instanceof` tests on "naked" parametric types such as `(T)x`;
- `catch` operations that take parameterized exception types;
- `new` operations on "naked" parametric types such as `new T()` and `new T[]`;
- static operations on generic classes, such as `T.class`

Furthermore, certain permitted operations, including:

- casts to a naked type parameter
- casts to generic type

generate `unchecked` warnings during compilation. An `unchecked` warning indicates that a hidden (not present in the original source code) cast inserted to support generic typing may fail and generate a run-time error.

3.4 NEXTGEN Implementation Scheme

The NEXTGEN implementation of Generic Java retains all parametric typing information for source programs in the compiled code and eliminates the restrictions on type-dependent operations imposed by GJ. NEXTGEN improves on GJ type erasure by making the erased base class `C` abstract and creating lightweight classes that extend `C` to represent instantiations `C<E>`. The type-dependent operations in `C<T>` are not erased in `C`, but rather translated into calls on synthetically generated `snippet` methods. The instantiation classes `C<E>` overload these `snippet` methods in `C` to provide specialized code encapsulating the type-dependent operations for `C<E>`.

3.5 NEXTGEN Support for Parametric Types

While the use of lightweight template classes preserves run-time type information, it also breaks intuitive subtyping relationships. The earlier example using `Stack<T>` and `Vector<T>` illustrates the problem. The instantiations `Stack<E>` and `Vector<E>` must inherit code and thus extend the base classes Stack and Vector, respectively. Also, intuitively `Stack<E>` is a subtype of `Vector<E>`. Figure 3.1 shows this illegal hierarchy. However, this multiple inheritance is impossible in the single inheritance object model in Java.

In the original NEXTGEN paper, Cartwright and Steele demonstrated how multiple interface inheritance can be used to solve this multiple inheritance predicament[11]. By introducing an empty interface `C<E>$` which is implemented by the instantiation class `C<E>`, NEXTGEN effectively decouples the generic instance `C<E>` from its associated type. References to `C<E>` are replaced with references to `C<E>$`. Figure 3.2 shows how the class hierarchy in 3.1 can be reformulated into a single class inheritance hierarchy. The NEXTGEN interpretation relies on the the presence of multiple inheritance of interfaces in Java NEXTGEN to provide correct run-time subtyping. Since these light-weight interfaces contain no fields or methods, their use only marginally effects program code size. A more detailed discussion is available in the earlier NEXTGEN papers [11, 6].

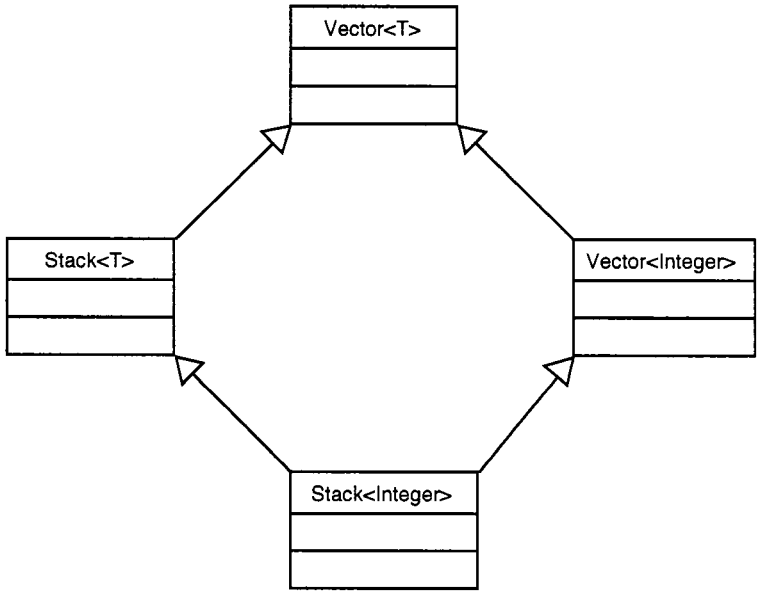


Figure 3.1 : Illegal Multiple Inheritance Class Hierarchy

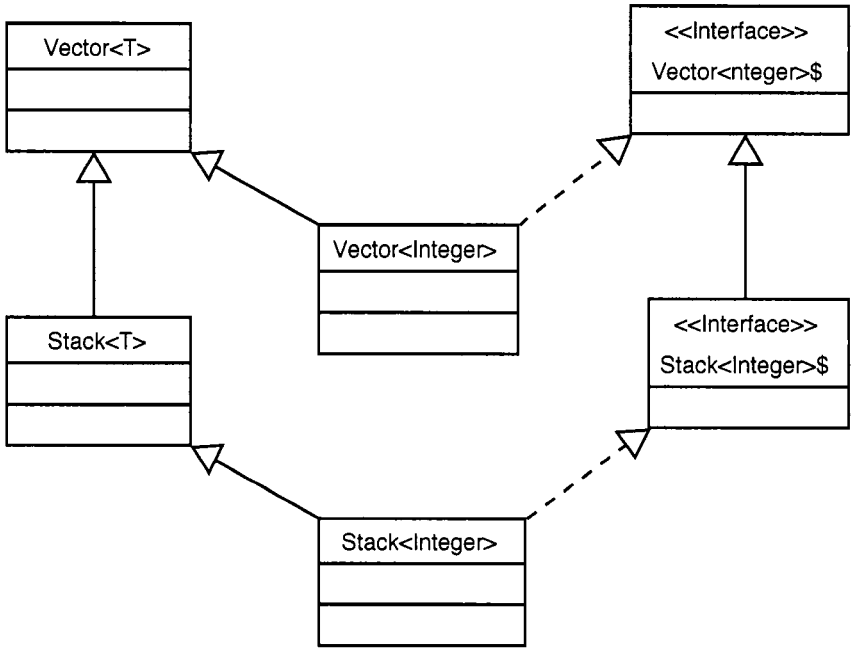


Figure 3.2 : Intuitive Parametric Type Hierarchy using Interfaces for Typing

3.6 Polymorphic Methods

Polymorphic methods provide a mechanism whereby type-dependent operations in the method body can depend on types provided at a call site. As discussed in section 3.3, type-dependent operations include `new`, `instanceof`, or casting instructions with a naked type parameter or a generic type parameterized by a type parameter.

In Generic Java, the syntax for the header of method declarations is generalized from

```
{modifiers} Type Identifier ( {ArgumentList} )
```

to

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

where `Type` can be any conventional type or `void`. The scope of the type variables specified in `TypeParameters` above is the header and body of the method. The type variables defined by a method shadow those of the enclosing class. Method invocations are generalized from

```
{ScopeIdentifier .} Identifier ( {ArgumentList} )
```

to

```
{ScopeIdentifier .} {<TypeParameters>} Identifier ({ArgumentList})
```

For most polymorphic method invocations, the type parameters can be left unspecified since the type inference algorithms used by Generic Java can infer the values of the type arguments from the types of the argument values in the invocation.

Conceptually, polymorphic methods can be divided into two categories: static polymorphic methods which are declared as static methods (essentially conventional procedures); and dynamic polymorphic methods which involve object-oriented dispatch in the context of a recevier. For static polymorphic methods, the bindings of the type variables used in type-dependent operations can be inferred statically from the call site. But for dynamic polymorphic methods, the static typing of the call site is insufficient if the receiver object introduces additional class parameterization (through sub-classing).

Figure 3.3 presents a static version of `List.zip`, the function that creates a new list containing a lexicographic pairing of the elements from the two input lists. The type variables `T` and `U` are used in the type-dependent instantiations of `new Pair<T,U>` and `new List<Pair<T,U>>`.

```
class List<T> {
   T first;
   List<T> rest;

   List(f, r) { first = f; rest = r; }

   static <T, U> List<Pair<T,U>> zip (List<T> left, List<U> right) {
       ...
      return new List<Pair<T,U>>( new Pair<T,U>(left.first, right.first),
                      List.<T,U>zip(left.rest, right.rest));
   }
}

class Pair<A,B> {
   A x;
   B y;
   Pair(x,y) { this.x = x; this.y = y; }
}

class Client {
   public static void main(String[] args) {
      List<Integer> i = new List<Integer>(new Integer(1), ... );
      List<String> s = new List<String>("A", ... );

      List<Pair<Integer, String>> p = List.<Integer, String>zip(i, s);
   }
}
```

Figure 3.3 : Static Polymorphic Method with type-dependent operations

3.7 GJ Implementation of Polymorphic Methods

In GJ, a polymorphic method `<T>m` generates a single erased method `m`. Using the GJ translation scheme, the source code in Figure 3.3 is converted into the code shown in Figure 3.4. In the translation, the parametric parameters and return type are all replaced by

their erased upper bound `List`. Perhaps, the most significant change is the conversion of `List<Pair<Integer, String>>` to `List`— a loss of two levels of type parameterization.

```
class List {
   Object first;
   List rest;

   List(f, r) { first = f; rest = r; }

   static List zip (List left, List right) {
       ...
      return new List( new Pair(left.first, right.first),
                       zip(left.rest, right.rest));
   }
}

class Pair {
   Object x;
   Object y;
   Pair(x,y) { this.x = x; this.y = y; }
}

class Client {
   public static void main(String[] args) {
      List i = new List(new Integer(1), ... );
      List s = new List("A", ... );

      List p = List.zip(i, s);
   }
}
```

Figure 3.4 : Type-erasure of a Static Polymorphic Method

3.8 NextGen Support for Polymorphic methods

The translation of polymorphic methods is very similar to the translation of parametric classes. Type-dependent operations in the body of the method are "snippetized". The generated snippet methods are stored in "snippet environments" since the type parameters declared by polymorphic methods are not visible by the enclosing class. A snippet environment is simply a light-weight singleton class containing only snippet methods and a static

field bound to the only instance of the class. The name of the snippet environment encodes the instantiated parametric types. For the sake of brevity, this chapter uses the naming convention of `[ClassIdentifier]$env` to denote a snippet environment. For operations that depends only on class-level parameterization, the snippet methods are stored in the instantiation class associated with the enclosing class. The formal parameters of the method are prepended with a special interface, specifying the set of type-dependent snippets and implemented by all relevant snippet environments. This differs from the support for generic classes, where interfaces are used simply to ensure correct typing of class hierarchies. The name of the interface encodes the upper bounds of the polymorphic type parameters it represents. For simplicity, this chapter uses the naming convention of `[ClassIdentifier]$env$` to denote these interfaces. The remaining code of the polymorphic method is unaltered by the snippetizing process.

An instantiated snippet environment is created at each polymorphic call site based on the statically inferred types at that site and is passed as an argument in the method invocation. The name of the instantiated snippet environment encodes the instantiated type parameters, as well as typing information related to the statically inferred receiver type.

While this transformation is more complicated than type erasure, it performs with minimal additional overhead. The overhead of light-weight snippet environments can be easily removed by specialized optimizations in the Just-In-Time (JIT) compiler. Furthermore, although Generic Java programs create a large number of generic instantiations, these instantiations are based on a small defined set of parametric types. Since the instantiation of snippet environments is deferred until run-time, NEXTGEN generates instances of a snippet environment only on demand. This demand-driven approach prevents the accidental creation of infinite hierarchies that may arise in chains of polymorphic recursion.

The following sections describe in detail how snippet environments are generated for static and dynamic polymorphic methods.

3.9 NEXTGEN Translation of Static Polymorphic Methods

Static polymorphic methods include all polymorphic methods declared private, static, or final. Since static polymorphic methods have no dynamic receiver, all of the polymorphic type information for a method call can be inferred statically. As a result, the correct snippet environment can be efficiently inferred and generated.

Static polymorphic methods are encoded using the translation outlined in the previous section. Figure 3.5 shows the the NEXTGEN translation of the source code show in Figure 3.3. The invocation of `List.zip(...)` has the statically inferred type arguments: `T` is instantiated to `Integer` and `U` is instantiated to `String`. Thus, the snippet environment `List$SE<<Integer,String>>.ONLY` is loaded at the call site. The syntax << and >> denote the types encoded by the `snippet environment`.

3.9.1 Propagation of Run-time Types

A subtlety in the NEXTGEN translation is the propagation of run-time types through recursive method invocations. In the zip function defined in Figure 3.3 there is a recursive call using the current type arguments. In the translation shown in Figure 3.5, NEXTGEN passes the current snippet environment, which represents the current type arguments, to provide the required instantiated types.

Direct passing of snippet environments characterizes a larger set of invocation call graphs typical in object-oriented paradigms. NEXTGEN passes a snippet environment anytime a method invocation instantiates the same exact type parameters. This applies to recursive methods, helper methods, and chains of overloaded methods that provide additional initialization.

In general, `snippet environments` provides the glue to carry polymorphic type information from a call site to a subsequent method invocation. When the parametric types can be inferred statically, NEXTGEN can explicitly generate the correct snippet environment. If the type arguments statically inferred at the call site are themselves dependent on run-time type information, provided by an enclosing generic class or polymorphic method, the creation of

```
class List<T> {
   T first
   List<T> rest;

   List(f, r) { first = f; rest = r; }

   static <T,U> List<Pair<T,U>> zip (List$env$ $env, List<T> left, List<U> right) {
       ...
      return $env.new$List<Pair<T,U>>(new Pair<T,U>(this.first, other.first),
                      zip($env, left.rest, right.rest));
    }

class Pair<A,B> {
   A x;
   B y;
   Pair(x,y) { this.x = x; this.y = y; }
}

class Client {
   public static void main(String[] args) {
      List<Integer> i = new List<Integer>(new Integer(1), ... );
      List<String> s = new List<String>("A", ... );

      List<Pair<Integer, String>> p =
                i.<String>zip(List$env<Integer,String>.ONLY, s);
   }
}

interface List$env$ {
    List<Pair> new$List<Pair<T,U>> (Pair f, List<Pair> rest);
}

class List$env<Integer, String> {
    List<Pair> new$List<Pair<T,U>> (Pair f, List<Pair> rest) {
        return new List<Pair<Integer, String>> (f, r);
    }
}
```

Figure 3.5 : NEXTGEN translation of Zip static polymorphic method

the snippet environment must be "snippetized" in the enclosing context.

```
class List<T> {
    ...
    <U> List<Pair<T,U>> zip (List<U> r) {
          return new List<Pair<T,U>>( new Pair<T,U>(f, r.f),
                        r.<U>zip(r.r));
    }
}
class Client {
   public static void main(String[] args) {
     List<Integer> i = ...;
     List<String> s = ...;
     List<Pair<Integer, String>> p = i.zip(s);
   }
}
```

Figure 3.6 : Dynamic Polymorphic Method Source Code

3.10 NEXTGEN Translation of Dynamic Polymorphic Methods

In the case of dynamic polymorphic methods, the snippet environment generated for a call site must carry not only the bindings for the type parameters introduced in the method definition, but also the bindings of type parameters in the generic type of the receiver. Both sets of bindings are determined by static type checking. Figure 3.6 shows a dynamic version of the `zip` function defined in Figure 3.3.

At the call site `i.zip(s)`, the object `i` has static type `List<Integer>`. So the call site defines the same type parameter bindings as in the static case discussed earlier: `T` is bound to `Integer` and `U` is bound to `String`. NEXTGEN uses slightly different name mangling schemes for static and dynamic method snippet environments because they are not interchangeable. After name mangling, the call site becomes:

```
List<Pair<Integer, String>> p =
  i.zip(List$$DE<<Integer,String>>.ONLY, s);
```

and the method signature becomes:

```
<U> List<Pair<T,U>> zip (List$DE$ $env, List<U> r)
```

Naming issues aside, the most important aspect of this translation is how snippet environments cope with dynamic dispatch. The class of the receiver can vary during program

execution; any class type that is a subtype of the static type of the receiver is possible. Hence, the generated snippet environment must contain the snippets for *all* of the definitions of the method consistent with the static type of the receiver. In each class that belongs to the static type of the receiver, the method may have a distinct definition.

Consider a `List` subclass `RevList` that extends the `List` class in Figure 3.6, overriding the method `zip` to return a reversed list of zipped pairs:

```
class RevList<T> extends List<T> {
    ...
    <U> List<Pair<T,U>> zip (List<U> r) {  ... }
}
```

At the call site in the `Client` class, the object `i` can be an instance of `List<Integer>` or `RevList<Integer>`. As a result, the snippet environment passed at this site must contain all of the snippets for the definitions of `zip` in both `List` and `RevList`.

Since snippet environment classes only contain code, it is advantageous to combine environments with compatible type bindings. Obviously, distinct calls on the same polymorphic method with the same generic type bindings (for both the method type parameters and the receive type parameters) should use the same snippet environment. In addition, it conserves space and class loading time to use the same snippet environment for all polymorphic method calls with identical type bindings, regardless of the polymorphic method that is being called.

Type-dependent operations depending on class-level parameterization are snippetized in the related instantation class of the enclosing generic class.

3.10.1 Reflection in Pathological Cases

The snippet environment technique described in the preceding subsection may not work if a subclass introduces a new type parameter `S` that is not bound to a type parameter in the parent. If the statically inferred receiver type at a call site is (a type instantation of) the parent class, then the generated snippet environment will not contain a binding for the new

type parameter S, and therefore not be able to support type-dependent operations involving S. Consider the following:

```
class AssociativeList<S,T> extends List<T> {
    S x; T y; AssociativeList<S,T> r;
    PairList(S a, T b, AssociativeList<S,T> c)
          { x=a; y=b; r=c; }

    <U> List<Pair<T,U>> zip (List<U> r) {  ... }
}
```

Recall the call site generates the snippet enviornment `List$$DE<<Integer,String>>.ONLY`. Since a binding for the type parameter `S` does not appear in the static type of the receiver in the call site in the `Client` class, the snippet environment generated for this site is independent of `S`. In fact, different values of the receiver type may have different bindings for `S` and some (such as instances of the classes `List` and `RevList`) may have no binding for `S` at all. As a result, the code for the body of `zip` in the class `AssociativeList` must use reflection to determine what snippet environment instantiation should be loaded in order to resolve the proper meaning of snippets in this body. Note that the body of `zip` in `AssociativeList` must involve snipppets that depend on the bindings of type variables in the receiver type and the bindings of type variables in the method call type before this reflective mechanism is necessary.

The easiest way to implement this approach is to use "bridge methods". If a method definition `m` in class `C` needs to invoke a snippet requiring reflection, the NEXTGEN compiler generates two methods to implement `m`:

- a method `m` that takes a conventional snippet environment just like versions of `m` in superclasses that do not require reflection and
- an overloaded variant of `m` that takes an augmented snippet environment including the snippets that depend on new class type parameters not present in the context where

method `m` is introduced.*

The definition for `m` with the conventional snippet signature is bridge code that uses reflection to create the *correct* snippet environment and passes it to the variant version of `m` that takes an augmented snippet environment that includes the special snippets depending on new class type parameters. Note that this implementation only incurs the overhead of reflection when the receiver object requires it. A particular call site for the method `m` only uses reflection when (*i*) the receiver object *o* is in the scope of type parameters that are not in the scope of the static type of the receiver expression and (*ii*) the definition for `m` in class of *o* involves atomic operations that depend both on the bindings of new class type parameters and the bindings of method parameters.† Executions at a call site do not use reflection unless the receiver object requires it. In addition, if the static type of the receiver includes bindings for the new type parameters, then the variant of `m` that takes an augmented snippet environment is directly invoked at the call site because the augmented snippet environment can be determined statically by the compiler.

3.10.2 Incidence of Reflection in Practice

Although the body of Java code using generics is still comparatively small, *we have not yet encountered a single case where reflection is required* in any of the generic code bases that we have examined including `javac`, DrJava [5], and the Java libraries that have generified. Introducing new type variables in subclasses is uncommon and overriding a polymorphic method in a way that depends on both these new bindings and method type variable bindings

*The definition of `m` that does not override a definition of `m` in its superclass. In C#, such a method definition is labeled with the `virtual` modifier while all of the overriding definitions must be labeled with the `override` modifier. Java does not make this syntactic distinction.

†Snippet operations that depend only on the bindings of new type parameters can be snippetized like any other type-dependent operations that appear in the class in question. Snippet operations that depend only on the the bindings of polymorphic method parameters (statically determined at the call site) can be included in the conventional snippet environment for `m`.

is even more unusual. In practice, we do not believe that the reflection overhead incurred in pathological cases will have any measurable impact on the performance of real programs. The only way we have been able to detect this potential overhead is to write artificial programs specifically designed to exhibit the pathology.

3.11 Wildcards

Java 5.0 includes support for wildcards [37], a language construct that provides a type safe abstraction over different instantiations of parameterized classes. A wildcard is denoted by using "?" as an argument type for a generic class. The semantics for wildcards includes an optional bounding clause of the form

```
extends ClassOrInterfaceType | super ClassOrInterfaceType
```

Wildcards can be used to parameterize any generic type, excluding the top-level parameterization of a generic type instantiated by a *new* operation. In GJ and Java 5.0, references to bounded wildcard types (*e.g.* `C<? extends String>`), and partial wildcards (*e.g.*, `D<?, String>`) are illegal in executable contexts (*e.g.*, casts and `intanceof` operations).

Currently, NEXTGEN provides first-class support for unbounded wildcards. References to an unbounded wildcards, e.g. `C<?>`, in executable code is translated to the abstract base class `C` for the type. NEXTGEN does not support partial wildcard types, *e.g.*, `D<?, String>`, but it could be easily extended to support them by adding instantiation interfaces for all possible partial wildcard types to the generic interface hierarchy, but we doubt that the minor gain in functionality would be worth the added complexity.

First-class wildcard types bounded only by a super type (using *extends*), can be supported using synthetically generated interfaces to represent the wildcard type, similar to NEXTGEN interfaces discussed in 3.5. These interfaces differ than their non-variant counterparts in that a subtyping relationship exists between interface types if they are covariant subtypes. For example, if a class `B` extends a class `A`, the type `C<? extends B>` is a subtype `C<? extends A>`, which would be reflected in their corresponding interface types.

Support for first-class wildcards bounded by a subclass (using *super*) is a more challenging problem because the contra-variant relationship between wildcard types cannot be easily supported in the demand driven, dynamic class loading in NEXTGEN. To support *super* bounds, NEXTGEN must record the wildcard information in the generic type and use an introspective mechanism for dynamic type checking.

3.12 NEXTGEN Design Complications

As a putative future edition of Java, NEXTGEN creates a minor technical problem, namely the issue of accessing package private types in a cross-package instantiation, that must be addressed compatibility with Java 5.0.

3.12.1 Cross Package Instantiation

The NEXTGEN design presented does not specify where the instantiation classes for generic types and snippet environments are placed in the Java namespace. The most intuitive location would be in the same package.[‡] However, this placement can cause a cross-package instantiation problem when a private type is used in a snippet operation contained in a instantiation class that resides outside the package boundary[28].

The simplest solution to this instantiation problem is to automatically widen a private class to public visibility if it is passed as a type argument in the instantiation of a generic type residing in another package. This tradeoff for simplicity at the cost security has precedent in Java. When an inner class refers to private members of its enclosing class, `javac` automatically widens the visibility of the relevant members by generating getters and setters with package visibility. Although more secure implementations are possible, the Java designers decided to sacrifice some visibility for the sake of performance.

One solution proposed in [11] is to use an initialization protocol to pass a environment containing the necessary snippets to the constructor of the instantiation class, *i.e.*, template

[‡]A static inner class would be ideal, but they are translated into toplevel classes by the compiler.

class or snippet environment. For generic classes, this solution requires an initialization protocol for instantiation classes that may be tedious to manage. However, this is a minimal problem since the snippet environments used for polymorphic methods already perform a similar initialization scheme. Snippet environments are containers with minimal typing constraints: each environment instance implements a single typing interface. Thus, the actual location of the snippet environment is not important. So the NEXTGEN compiler can widen the private class to package visibility and then instantiate the snippet environment in the same package as the private class. This approach requires the same number of snippet environments as simply widening visibility from private to public. Therefore, NEXTGEN can address the cross package instantiation problem with minimal additional overhead.

Chapter 4

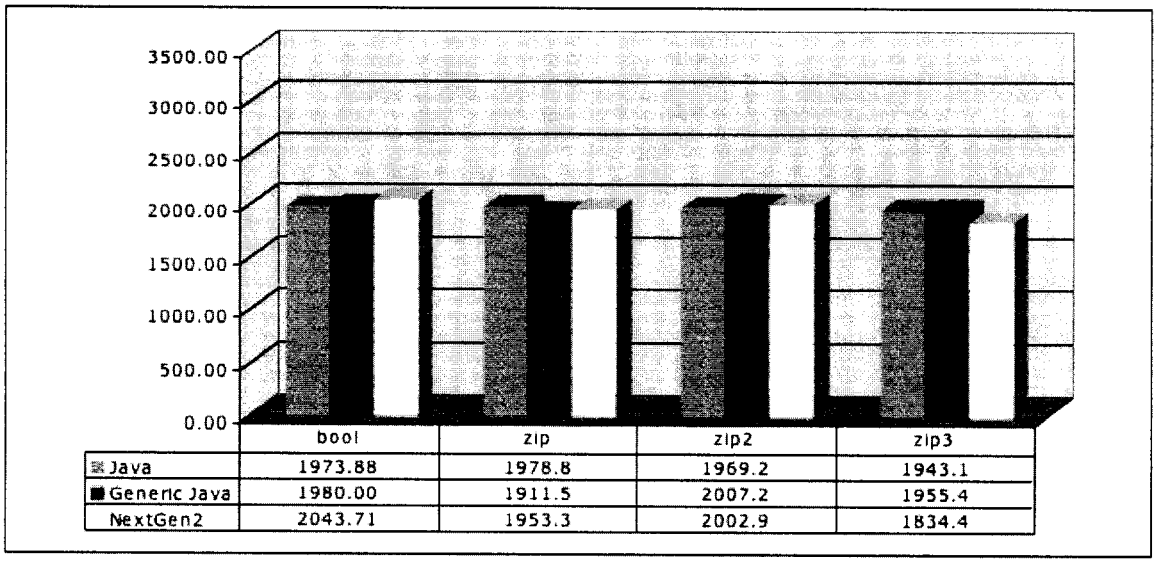
NEXTGEN Performance

The NEXTGEN benchmarks measure the overhead of supporting polymorphic method in the context of first-class genericty. Since no established benchmark for Generic Java currently exists, we had to develop our own specialized benchmark to analyze the performance of NEXTGEN. Each test was engineered in Generic Java. The benchmarks were then hand-translated into an equivalent non-generic Java source to provide a fair comparison with standard Java code. While the JSR14 compiler has an option to output non-generic source code, the output is not necessarily valid Java source code. The benchmarks consists of the following programs, all of which involve generics:

- **Bool:** A Boolean expression simplifier. This program parses in a large number of boolean expressions into an AST and simplifies them. The expressions are simplified through a series of passes by a generically typed visitor. This benchmark consists of 730 lines of code in 25 classes, 7 of these classes make heavy use of generics. While all the generically type visitor makes extensive use of polymorphic methods, it does not perform any type-dependent operations.

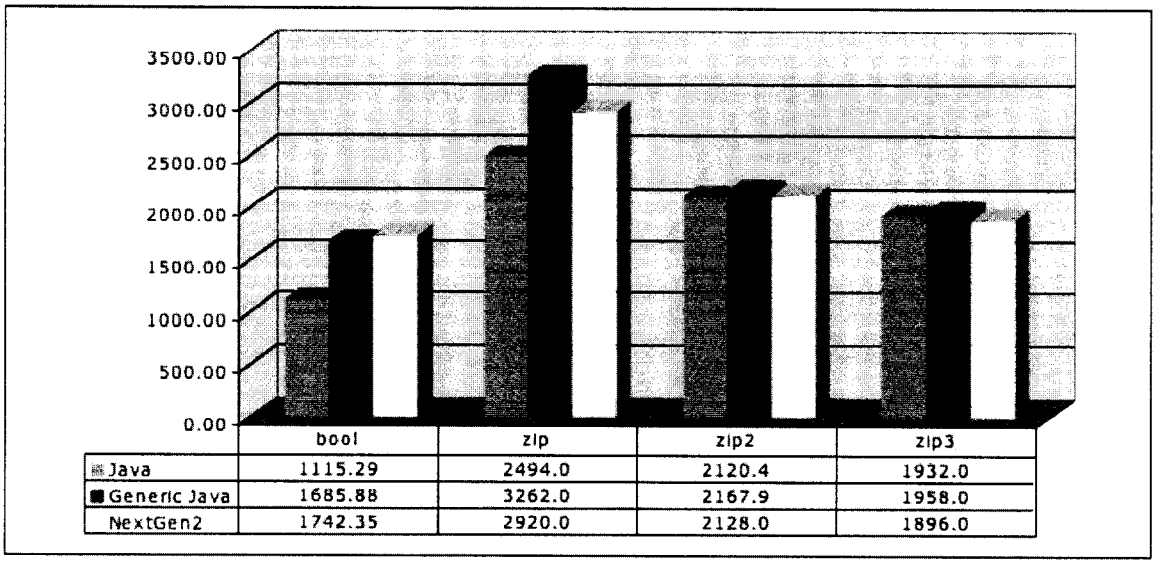
- **Zip1** Small program that repeatedly constructs small lists of 100 elements, and then zips them together. This case analyzes the initial overhead of calling a polymorphic method. This test consists of 80 lines of code in 3 classes, 2 of these make heavy use of generics and polymorphic methods.

- **Zip2:** Small program that repeatedly constructs lists of 1000 elements, and then zips them together. This case shifts focus more toward the performance of recursive methods. This test consists of 80 lines of code in 3 classes, 2 of these make heavy use of



	bool	zip	zip2	zip3
Java	1973.88	1978.8	1969.2	1943.1
Generic Java	1980.00	1911.5	2007.2	1955.4
NextGen2	2043.71	1953.3	2002.9	1834.4

Figure 4.1 : General Performance Results (ms)



	bool	zip	zip2	zip3
Java	1115.29	2494.0	2120.4	1932.0
Generic Java	1685.88	3262.0	2167.9	1958.0
NextGen2	1742.35	2920.0	2128.0	1896.0

Figure 4.2 : First Iteration of General Performance Results(ms)

generics and polymorphic methods.

- **Zip3:** Constructs a single pair of 1000 element lists, and then repeatedly zips them

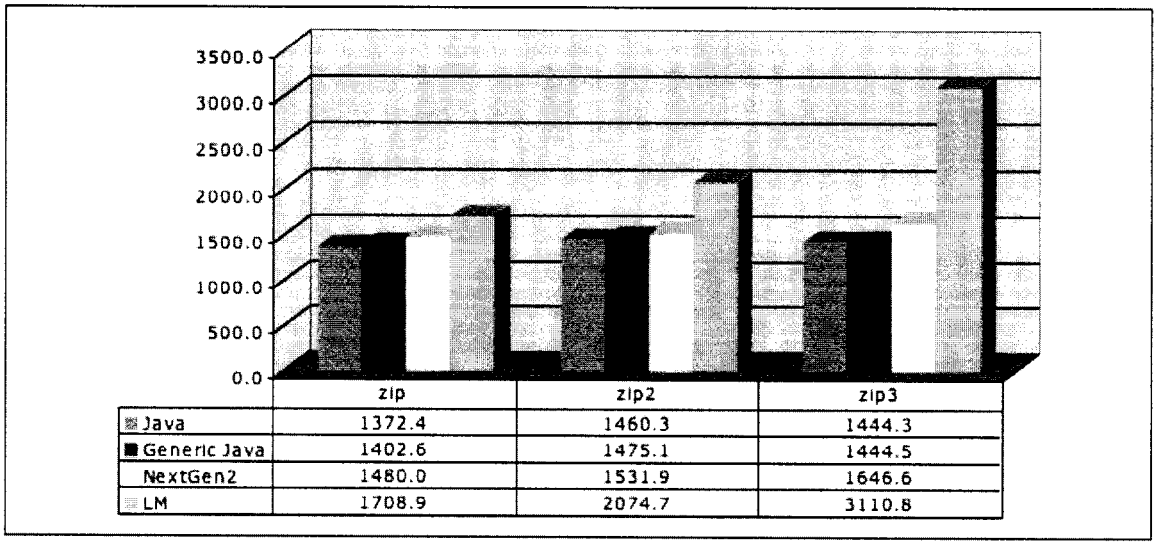
together. This case analyzes the potential of current Just-In-Time (JIT) optimzations. This test consists of 80 lines of code in 3 classes, 2 of these make heavy use of generics and polymorphic methods.

- **Reflect1:** Baseline test analyzing the performance of the case of pedagogical reflection. It constructs a list of pairs, and calls a functional operation `changeSecond` to change the type of the second element in each pair. This test consists of 100 lines of code in 4 classes, 3 of these make heavy use of generics and polymorphic methods.

- **Reflect2** Second test analyzing the performance of the case of pedagogical reflection. This test inserts a 5% probability of pedagogical reflection. It performs just like **Reflect1**: It constructs a list of pairs, and calls a functional operation `changeSecond` to change the type of the second element in each pair. Approximately 5% of the list are a subtype of pair that introduces an additional class-level type parameter. This test consists of 100 lines of code in 4 classes, 3 of these make heavy use of generics and polymorphic methods.

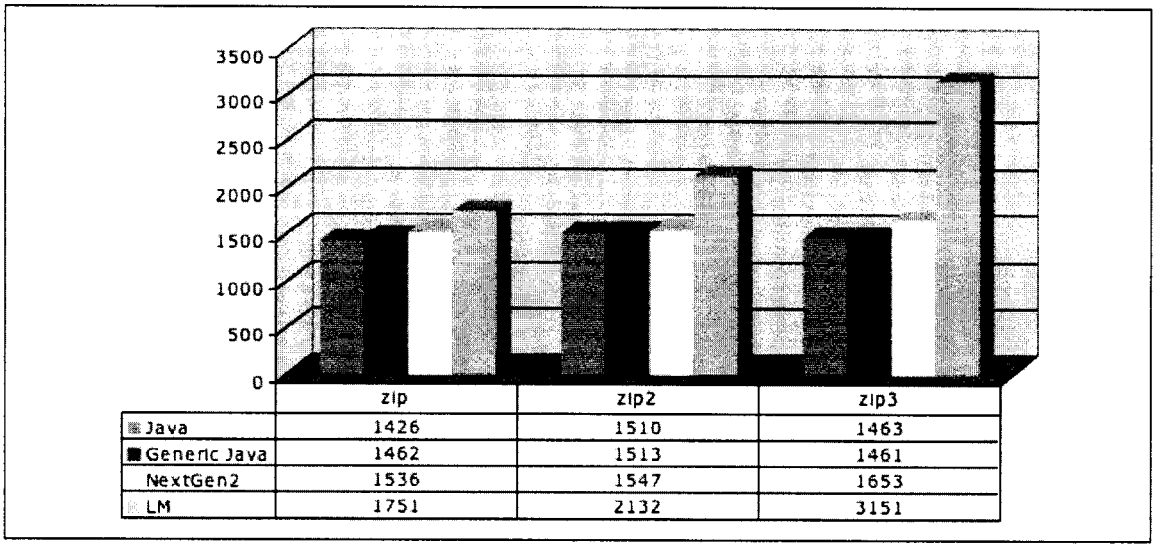
- **Reflect3** Third test analyzing the performance of the case of pedagogical reflection. This test inserts a 5% probability of pedagogical reflection. It performs just like **Reflect1**: It constructs a list of pairs, and calls a functional operation `changeSecond` to change the type of the second element in each pair. Approximately 10% of the list are a subtype of pair that introduces an additional class-level type parameter. This test consists of 100 lines of code in 4 classes, 3 of these make heavy use of generics and polymorphic methods.

- **JSR14 2.2** The task of compiling a compiler represents a very robust use of generics. The JSR14 compiler uses a series of Tree generic visitors to compile a Java source program to bytecode. The JSR14 compiler consists of approximately 25,784 lines of code in 225 classes. Nine classes make light use of generics, 59 classes make moderate use of generics, and 8 of these make heavy use of generic type and polymorphic methods.



	zip	zip2	zip3
Java	1372.4	1460.3	1444.3
Generic Java	1402.6	1475.1	1444.5
NextGen2	1480.0	1531.9	1646.6
LM	1708.9	2074.7	3110.8

Figure 4.3 : Performance of Polymorphic Method Recursion (ms)

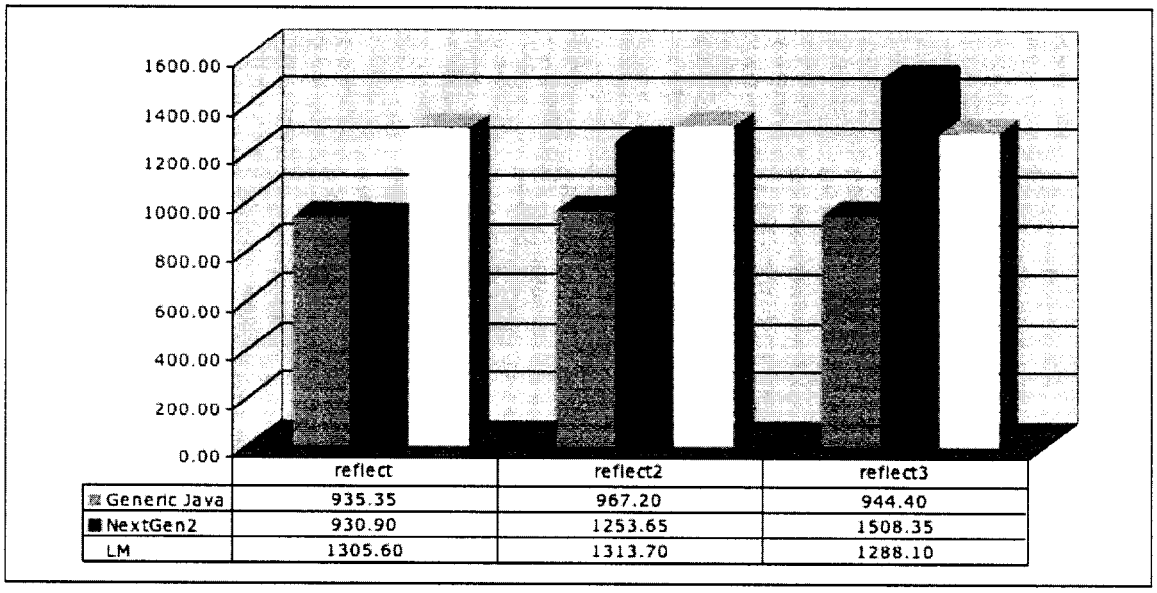


	zip	zip2	zip3
Java	1426	1510	1463
Generic Java	1462	1513	1461
NextGen2	1536	1547	1653
LM	1751	2132	3151

Figure 4.4 : First Iteration of Performance of Polymorphic Recursion (ms)

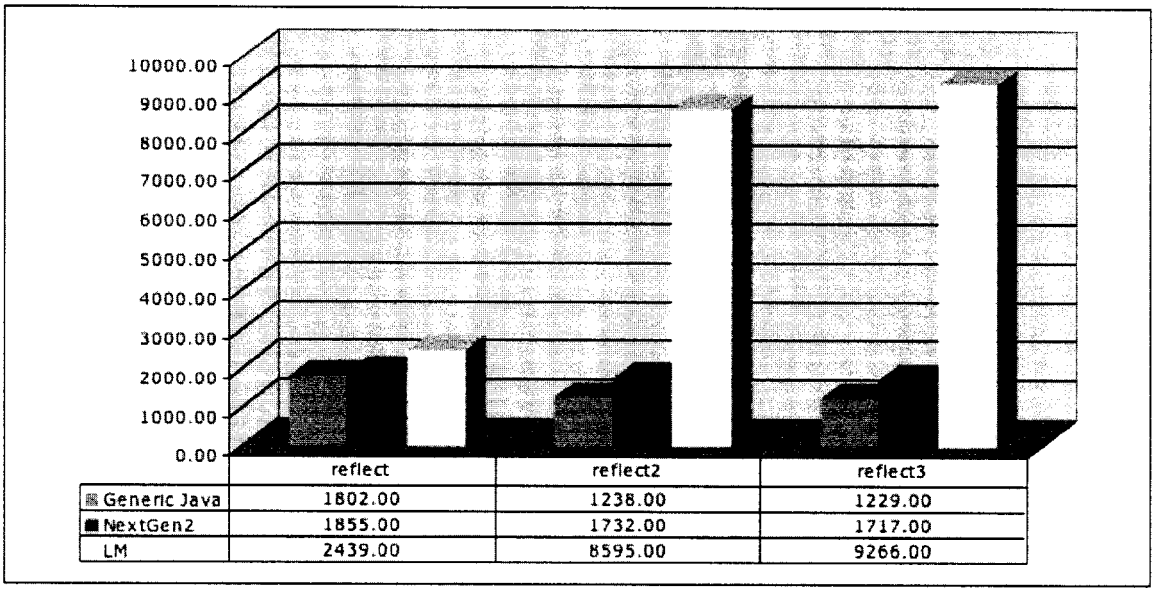
The benchmarks compared performance between the following platforms (not all were used in all cases):

- **Generic Java**: Generic source code compiled using Java 5.0



	reflect	reflect2	reflect3
Generic Java	935.35	967.20	944.40
NextGen2	930.90	1253.65	1508.35
LM	1305.60	1313.70	1288.10

Figure 4.5 : Performance of Pedagogical Reflection (ms)



	reflect	reflect2	reflect3
Generic Java	1802.00	1238.00	1229.00
NextGen2	1855.00	1732.00	1717.00
LM	2439.00	8595.00	9266.00

Figure 4.6 : First Iteration of Performance of Pedagogical Reflection (ms)

- **Java**: The equivalent non-generic source compiled using Java 5.0

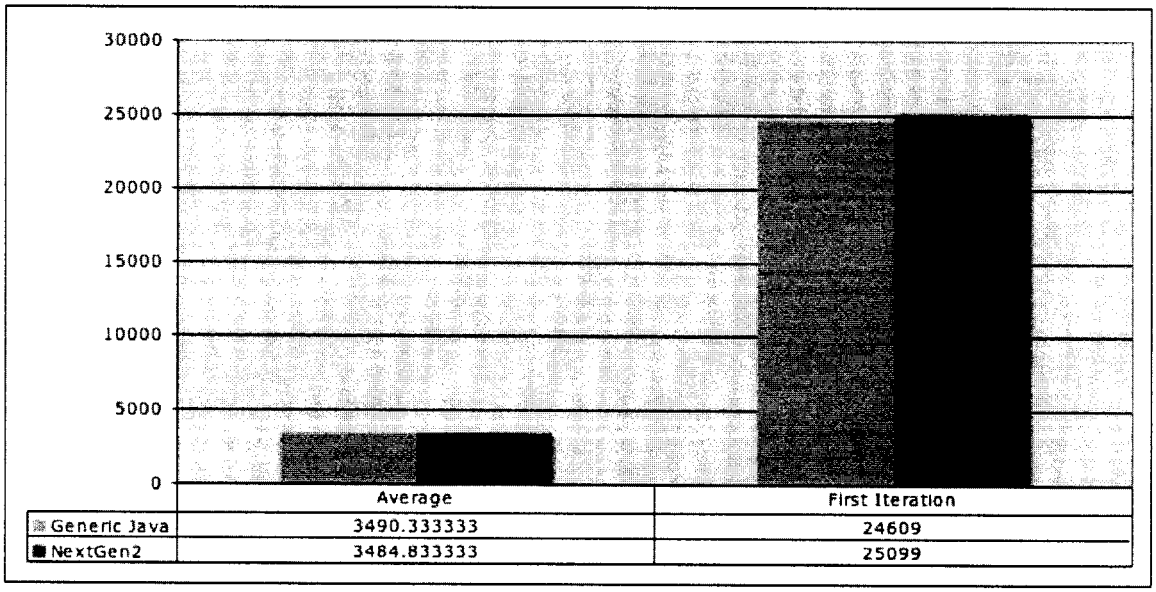


Figure 4.7 : JSR Performance Results(ms)

- NEXTGEN/NEXTGEN2: Generic Java source compiled using NEXTGEN
- **LM**: Generic Java source adapted for the LM architecture supporting first-class generics.[39, 38].

Each benchmark was executed twenty-one times using the Sun 1.4.1 server Java Virtual Machine (JVM) on a 2 Ghz, Intel Pentium 4 with 512 MB of RAM running Redhat Linux 8.0. The NEXTGEN benchmarks focus on the server JVM since the client JVM does not perform any Just-In-Time (JIT) optimizations. The expectation is that by design, the NEXTGEN code augmentations can be offset by specialized JIT optimizations.

The first iteration of each benchmark was dropped because it deviated significantly from the remaining 20 iterations. We believe this deviation results from the overhead of JVM startup and initial JIT compilation of the code, both of which are not directly relevant to what we are trying to measure. After dropping the first run, the variance among iterations for each benchmark was less than 10%. For thoroughness, this thesis also includes performance metrics of the first iteration of each test.

The slower performance of NEXTGEN reflects the overhead of polymorphic methods, and also the overhead of generic instantiation classes. This is to be expected since an implementation of polymorphic methods under first-class genericity requires a parametric representation of types. In general though, the majority of the operations in a program will not depend on run-time type information, and therefore, support for first-class genericity is not costly when amortized over a large number of instructions [39].

Figures 4.1-4.2 show that, given the current level of JIT optimizations, NEXTGEN performs competitive to Java and Generic Java. Surprisingly, the results for `zip3` show a 5% performance gain in the case of recursive polymorphic methods. An analysis of the metrics for zip1, zip2, and zip3 suggest this speedup results from JIT optimizations. In comparison to zip1 and zip2, the zip3 benchmark focuses only on the recursive invocation of zip; successive iterations reuse the same set of `Lists` to invoke `zip`. Since the trend in the NEXTGEN performance differs from that of GJ or Java, its safe to assume that the NEXTGEN translation provides a basis for new JIT optimizations.

Figures 4.3-4.4 compares NEXTGEN against LM using the tests zip1,zip2, and zip3. Because of the larger space requirements for classes in LM, the structure of the test zip3 was rewritten to improve garbage collection during execution. This change is reflected in the different results for zip3 shown in Figures 4.1-4.2 and 4.3-4.4.

Since these zip benchmarks use only new operations and polymorphic method invocations, they provide a fair comparison between NEXTGEN and LM.* LM new operations require an extra parameter to store run-time types, while NEXTGEN must load specialized instantiation classes. Polymorphic method invocation in LM requires passing a method descriptor (an integer index). This should be as fast, if not faster than passing a local final field in NEXTGEN.

In general, NEXTGEN performs at least 20% faster than LM. The overhead of the LM approach can be attributed to two factors. First, LM classes maintain a reference to a `type`

*Instanceof tests in LM require an O(n) worst case iteration, where n is the level of parameterization. NEXTGEN can perform instanceof tests in O(1) time.

`descriptor` object that stores generic type information. For a small collection class like `Pair` with only two fields, this increases object size by 50%. Second, the dynamic nature of LM preempts many JIT optimizations. During the execution of zip, a new `Pair` is created on each call. At each constructor invocation, LM must perform a lookup to retrieve the `type descriptor` for the new object, and thus determine its type parameterization:

```
((MD)td.MDs[0].elementAt(p)).friendTD[1]
```

Although this value never changes, LM must recompute it at each iteration. NEXTGEN on the other hand, relies on static class identifiers, allowing for potential JIT optimizations.

In the case of zip3, LM requires twice the time as GJ or NEXTGEN. A possible explanation for the extremely high overhead is that the higher space needs of LM resulted in some objects being stored in virtual memory and swapped to disk.

Figures 4.5-4.6 shows how the performance of NEXTGEN degrades in the pedagogical case. NEXTGEN performance degrades linearly since reflection code is executed in the bridge methods for a polymorphic method. Figure 4.5 implies that NEXTGEN produces an inefficient implementation when the probability of reflection is greater than 5%. On the average though, NEXTGEN is still more efficient than LM when the probability of reflection is less than 7-9%. An analysis of only the first run performance, shown in figure 4.6, show a perplexing result. LM performs 4-5 times slower than NEXTGEN and GJ in the tests reflect2 and reflect3. A conjecture is that constructor supercall chains are inefficiently handled in LM; each supercall stores a reference to the "farther" of the current type descriptor. Since the exact cause of this slowdown cannot be determined by this benchmark alone, a more in-depth analysis will be included in future work.

The JSR14 benchmark in figure 4.7 shows that a JSR14 compiler compiled with NEXTGEN performs competitive to a GJ version. This result affirms the belief that the cost of maintaining run-time type information, and thus supporting first-class genericity in Java, is not costly in large applications, since it is amortized over a large number of operations.

Chapter 5

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 rich 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 JVM implementation, we can reduce component parameterization to class parameterization.

Although we are focusing our attention on developing a component system for generic Java, essentially the same design issues arise developing a component system for any strongly-typed object-oriented language with nominal subtyping, most notably C#. All of our constructions adapt with minor adjustments to the context of C#.

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 run-time, 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 run-time only in the restricted sense that the members of the module are conventional Java classes and interfaces that are dynamically loaded at run-time. 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 provides concise names for signature and module instantiations. In addition, it provides a simple mechanism for linking mutually recursive modules.

We first present a more precise definition of signature and module instantiations, and then proceed to discuss signatures, modules, and bindings.

5.1 Signature and Module Instantiations

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 instantation:

```
SignatureType → QualifiedIdentifier < TypeArgs >
```

where

```
TypeArgs → TypeArg, TypeArgs
TypeArg → TypeVar | ModuleType
TypeVar → Identifier
```

Each `TypeArg` is either an imported module parameter (which is bound to a module) or a module instantiation. Note that all signature expressions are fully instantiated. There is no partial application of unlinked modules to module arguments.

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

```
ModuleType → QualifiedIdentifier < TypeArgs >
```

where each `TypeArg` is either an imported module parameter (which is bound to a module) or a module instantiation. Note that all module expressions are fully instantiated. There is no partial application of unlinked modules to module arguments.

The intuitive meaning of a module instantiation is literally the set of classes in the instantiated module. It is equivalent to a package in ordinary Java. Hence, there is no such thing as multiple instances (or copies) of a particular module instantiation.

5.2 Signatures

A *signature* is a syntactic entity consisting of a collection of class and interface prototypes that are used to constrain the "shape" of a matching module. More precisely, a signature has the form:

```
signature QualifiedIdentifier<ModTypeParameters> {extends SignatureTypes};
SigMember*;
```

where

```
ModTypeParameters → ModTypeParm | ModTypeParm, ModTypeParameters
ModTypeParm → TypeVar implements SignatureType
SignatureTypes → SignatureType | SignatureType, SignatureTypes
```

and { } enclose optional phrases. `SigMember` is either a class or interface prototype, a `bind` statement, or an `import` statement. The `SignatureType` in the `implements` clause of `ModuleTypeParam` serves as a bound for the preceding `TypeVar`. The `SignatureTypes` following the `extends` clause defines a set of signatures from which the signature inherits, akin to Java interfaces inheriting from other interfaces.

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 visiblity of the members of class prototypes can either be `public` or `protected`. `public` visibility is the default.

Figures 5.1 and 5.2 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`.

5.3 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 QualifiedIdentifier<ModTypeParameters> implements SignatureTypes;
ModuleMember*;
```

where `ModuleMember` is either a class or interface definition, an import statement, or a `bind` statement (which is defined in Section 5.4). Figure 5.3 shows a module `JamParser` that implements the signature `SParser`.

```
signature SSyntax;

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

Figure 5.1 : 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 5.2 : Signature for Jam parser

```
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 5.3 : JamParser Module Definition

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.

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

---

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

†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 5.4 : Module Instantiation

5.4 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 indentifier to a (linked) module instantiation. A module binding has the form:

`BindDecl` → `bind SignatureType Identifier = ModuleType;`

The `SignatureType` specifies the signature to associate with `Identifer`, which can be an exported signature of `ModuleType` or any super signature. In this way, the programmer can expose only the requisite part of a module with a larger export signature. Figure 5.4 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 contruction of program modules with recursive links, *e.g.* module `A` imports module `B` and vice-versa.

Chapter 6

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" [4]. Our adaptation of this principle is the following:

> 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 support wide latitude in refactoring component implementations while retaining compatiblity with the former signatures (APIs).

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 desing possibilities.

6.1 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

```
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 6.1 : Inheritance Across Component Boundaries

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 `DebugJamParser` in Figure 6.1, that provides a `Parser` with a more sophisticated logging facility. In the instantiated module:

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

the definition of `getLog()` in `DebugJamParser.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 [2]. 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. While mixins across component boundaries in CGEN are more restrictive than class-level mixins, CGEN employs 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 [2]. 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.

6.2 Cyclic Class Hierarchies

Generic Java forces the type 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 heirarchy.

In the presence of modules we must enforce the same constraint. However, our analysis of

the type hierarchy is less precise. The collection of classes is not completely known during the compilation of isolated modules. Some type-checking 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 of 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 allow a recursive module or mutually recursive (or dependent) modules to form cyclic class hierarchies. The following examples demonstrate some of the potential complications that can arise.

Example 1.

A class prototype extending an imported class in a recursive module could 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 recursive pattern is just like recursively defining a class or a function.

Example 2.

The cyclic hierarchy in Example 1, can be expanded to involve a set of mutually recursive signatures defining a set of cyclic classes.

```
signature S<B implements T<S<B>>>;
 interface C implements B.D { };

signature T<A implements S<T<A>>>;
 interface D implements A.C { };
```

Example 3.

By subtyping the declared parent type specified in a bounding signature, as discussed in Section 5.3, a cyclic class hierarchy can be concealed by its signature. The signature in Example 1 can be rewritten to:

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

which may conceal a cyclic hierarchy defined by the module `M<A>`:

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

Example 4.

The code in Example 3, can be expanded to involve a set of mutually recursive signatures and modules defining a set of cyclic classes.

```
signature S<B implements T<S<B>>>;
 class C { }

signature T<A implements S<T<A>>>;
 class D { };
```

```
module M<B implements T<S<B>>> implements S<B>;
 class C extends B.D { };

module N<A implements S<T<A>>> implements T<A>;
 class D extends A.C { };
```

Of course, the examples stated above can be easily extended to involve arbitrary intermediate subclasses and arbitrary numbers of signatures or modules.

CGEN can enforce the DAG constraint on type hierarchies with respect to the limited code present during compilation. For Examples 1 and 2 with dependent signatures, CGEN can easily perform an analysis of the containing classes to ensure the type hierarchies spanning the signatures form a DAG.

Since signatures conceal the cyclic classes in Examples 3 and 4, an analysis could only ensure that classes defined in their respective modules form a DAG. The detection of cyclic classes spanning module definitions must be deferred until the modules are linked together (using bind) at which time CGEN can instantiate the modules with their appropriate type arguments (module imports). This process fills in the missing super types of the mixin classes. CGEN can then determine if the modules produce a set of types that form a DAG. If a cyclic type hierarchy exists, CGEN can report this as an error resulting from improper linking, but this detection does not fully comply with the safe instantiation principle. Another module satisfying exactly the same signature could be used in the very same context without any problems.

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.

Chapter 7

Implementation Details

CGEN is an extension of the NEXTGEN compiler, a derivation of the Sun Java 5 compiler supporting first-class genericity, and a customized `classloader` to provide the framework necessary to propagate run-time type information. By building on the Java 5 code base, CGEN can immediately (1) Perform accurate type checking against the latest Java language specification, (2) Use support libraries in the sun `javac` compiler to minimize code redundancy and improve readability, and (3) Take advantage of a more systematic compiler API with fewer bugs and inconsistencies.

7.1 Code Maintenance

The cost of maintaining a full compiler is beyond the capabilities of the Rice JavaPLT research group. Understanding a compiler sophisticated as the `javac` compiler requires not only a wide breath of knowledge of the diverse stages of compilation, but also a profound understanding of the peculiarities of the underlying JVM implementation. Furthermore, it is infeasible to replicate the daily development of the `javac` compiler. There are continuous updates in the Javac compiler to fix bugs and to improve the interoperability between generic and non-generic code. Also, since the Java Programming language is under constant evolution, e.g., the recent addition of wildcards, and auto-boxing, it would be impossible to maintain a compatible compiler.

An optimal solution would decouple the CGEN extensions from the maintenance of the underly Java compiler. One option is to implement CGEN as a source-level preprocessor used before `javac` [6]. However, this approach is infeasible because CGEN name-mangling techniques are illegal in Generic Java identifiers. A second option is to implement CGEN as

a post-processor on Java bytecode. Unfortunately, most of the run-time type information has already been removed by the compiler, and the resulting bytecode relies on erased types. A third option is to perform only a minimal amount of direct modifications to the Java Compiler and then perform the remaining transformations on the bytecode level. In this implementation, the NEXTGEN compilation phase encodes the run-time type information needed for bytecode translation. This frees NEXTGEN from complete depending on the javac compiler, and the quarks and complexities of its data structures and APIs.

The CGEN compiler follows the third approach listed above. Details are provided in the following sections.

7.2 CGEN Compilation Model

From the user's point of view switching to CGEN requires a minimal adjustment from their current Java model: The command `ngc` replaces `javac`, and `nextgen` replaces `java`. The compilation of modules, just like the compilation of generic classes, generates extra class files for CGEN templates. CGEN supports separate compilation of unique modules, but a specific module must be compiled in its entirety.

7.3 Translation of Components

Support for components in CGEN builds on the existing NEXTGEN infrastructure supporting first-class genericity consisting of a specialized compiler and classloader. The existing parsing, type-checking and name resolution routines required minor adjustments to support signatures and modules definitions. The CGEN compiler contains additional type checking routines to verify a module definition against its declared signatures.

After type-checking, signatures and modules are "flattened" into regular Java code. The flattening of *signatures* is straightforward. Class-prototypes in a signature are flattened just like `static` inner classes. Signature-level parameterization is pushed to the class-level and the classes are moved to the top-level. The fully qualified name of the signature is used as the package for the collection of class prototypes. Since class prototypes only declare

specifications, they are compiled into `abstract` classes containing only method signatures without providing actual code. A *manifest* class file is generated to represent the signature. Java class attributes are used to store the specifications of the interfaces imported and exported by the signature as well as the class prototypes contained in the signature. The generated class files are used for static and dynamic type checking of modules- they are not used for any run-time representation in the JVM.

Modules are similarly "flattened". Module-level parameterization is pushed to the class-level and the classes in the module are moved to the top-level. Hence, the class files generated for a module are simply the class files for the classes in the module. Moreover, the references in these class files refer directly to classes—not to modules. The fully qualified name of the module is used as the package for the new top-level classes. A *manifest* file is also generated to store the import and export specifications of the module and a list of the prototypes contained in the module.

After "flattening", the new top-level classes are compiled into templates which are used at run-time to provide fully heterogeneous representations of each type-instantiated class. A template class file looks exactly like a conventional class file except that strings in the constant pool may contain embedded references to the type parameters for the class. These references specify an index of the form {0},{1}, *etc*, specifying which type parameter binding should be inserted when the class file is instantiated. The CGEN classloader replaces these references with the corresponding type arguments (represented as mangled strings) to generate specific generic class instantiations.

7.3.1 Module Parameterization

As stated earlier, module-level parameterization is pushed to the class-level. For example, a module `M<S>` providing a class `C<T>` would be encoded as `M.C <S,T>`. If we naively translate a module into the classes representated by the module we could have an explosion in the number of type parameters, *e.g.* `M.C<A,B,C,T>`. However this would cause produce unreasonably length identifiers, and possibly create infinite name for mutually-dependent

modules.

For mutually-dependent modules, we use the identifier `$thisMod` to denote a recursive reference to the enclosing module type.

For our translated class `M.C<S,T>`. the generated class file contains references in the constant pool, of the form {[#]-[ClassName]} denoted references to classes imported by other modules. By using this approach, instantiating modules is as efficient as instantiating a generic type.

`Bind` declarations are processed like `import` declarations. The local names and their bound types are stored in the type environment and used to resolve identifiers in the code. Each identifier is replaced by its real fully-qualified name.

7.4 Type Flattening

NEXTGEN performs type flattening after type checking. In this stage, NEXTGEN replaces parametric types with their NEXTGEN mangled representation, and snippetizes type-dependent operations. The basic type flattening scheme is as follows:

1. Recursively traverse the code.

2. Encode type-dependent operations into a snippet method call. NEXTGEN hashes the generated method names to ensure uniqueness.

3. Generate the corresponding snippet method in the related template class. For class level snippets, NEXTGEN stores the snippet in the class's template class. For polymorphic methods, NEXTGEN stores the snippets in the corresponding snippet environment.

4. When translating polymorphic methods, prepend the method parameters with the corresponding snippet environment interface.

5. On method invocations of polymoprhic methods, generate code to instantiate a `snippet environment` and pass it as a parameter into the method call.

After traversing the code, NEXTGEN stores the newly generated template classes for parametric classes and the snippet environments for polymorphic methods.

In the following sections we discuss the underlying encoding strategy for generic types and polymorphic methods and then show how we extend these encodings to provide support for modules.

7.4.1 Encoding Parametric Types

NEXTGEN must generate distinct class identifiers for the templates used to encode runtime type information. The class identifiers include the character sequence "$$", which by convention does not appear in Java source code. The use of two "$" characters distinguishes NEXTGEN classes from mangled inner classes which use a single "$". For a generic class `A<S,T>`, NEXTGEN creates the following two additional classes:

- `A<S,T>`, the light-weight template class
- `A<S,T>$`, its corresponding typing interface

where S, T are fully quantified class identifiers. All references to generic types are then encoded to valid java identifiers using the following translation scheme:

- Left angle bracket '<' to "$$L"
- Right angle bracket '>' to"$$R"
- Comma ',' to "$$C"
- Period (dot) '.' to "$$D"

So the class

```
Pair<java.lang.String, java.lang.Integer>
```

would be encoded as:

```
Pair$$Ljava$$Dlang$$DString$$Cjava$$Dlang$$DInteger$$R
```

The encoding for **snippet environments** builds on the encoding for generic types. A polymorphic method `m<T>` in class `A<S>` generates two classes: a **snippet environment** and its corresponding interface. For brevity, our earlier discussion referred to these two classes using the identifiers `[ClassIdentifier]$env` and `[ClassIdentifier]$env$`. The precise naming scheme is:

- `A≤S≥<bounds(T)>$$E<T>`, a **snippet environment**
- `A≤S≥<bounds(T)>$$E`, its corresponding typing interface

where `bounds(T)` are the full class identifiers for the upper bounds of the type parameters T. The indentifiers are then encoded using the following translation scheme:

- Left small angle bracket '≤' to "$$l"
- Right small angle bracket '≥' to "$$r"

and `$$E` can be either:

- "$$ES" denotes a static method environment
- "$$ED" denotes a dynamic method environment

Recall the dynamic **zip** example introduced in section 3.10. The the call site:

```
List<Pair<Integer, String>> p = i.<String>zip(s);
```

will produce the following snippet environment and interface:

```
List<java.lang.Integer>≤java.lang.Object≥$$ED<java.lang.String>
List<java.lang.Integer>≤java.lang.Object≥$$ED
```

which is encoded as:

```
List$$Ljava.lang.Integer$$R$$ljava.lang.Object$$r$$ED$$Ljava.lang.String$$R
List$$Ljava.lang.Integer$$R$$ljava.lang.Object$$r$$ED
```

As shown above, the NEXTGEN encoding scheme for polymorphic methods generates snippet environments based on the enclosing class and the type parameters in scope. Therefore, methods that have the same type parameters, and probably a related function, share a common snippet environment.

7.5 NEXTGEN Classloader

At run-time, the NEXTGEN classloader intercepts JVM requests to load a class with a mangled identifier. It then loads the corresponding template files to generate the requested class or interface. It searches the identifiers in the constant pool replacing each reference tag {n}, where n is an integer, with the corresponding type specified by the mangled class indentifer. The NEXTGEN classloader sticks closely to the original NEXTGEN classloader defined in [6].

Run-time support for modules incurs minimal additional overhead over the support for first-class generics because module-level parameterization is represented by generic class parameterization. References to imported modules are resolved like references to generic class type parameters.

Since CGEN instantiates modules heterogeneously, this results in the duplication of module code. However, since CGEN instantiates module classes on demand, code duplication occurs only on the classes referenced during program execution.

Chapter 8

Core CGEN

During the development of CGEN, we encountered many subtle issues in the design and implementation of the type system, prompting us to develop a formalization of CGEN. This chapter presents Core CGEN (CCG), a small, core language modeling the CGEN framework. It is based on the Featherweight GJ (FGJ) [17] and incorporates ideas from Core MixGen [2]. CCG adds the essential features to support components, but nothing more. Our analysis includes the type rules and operational semantics for CCG, as well as a proof of type safety. For the remainder of our discussion, 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 provides 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*) this allows different module implementations to provide different collections of fields.

The semantics and proof are similar to those for Featherweight GJ[18] and MixGen in [3], 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.

8.1 Syntax

The abstract syntax of CCG, shown in Table 8.1, 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 type variables.
- `R`, `S`, `T`, `U`, `V` range over all types.
- `x` ranges over method parameter names.
- `f` ranges over field names.

Syntax:

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

Table 8.1 : CCG Syntax

- 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, $\overline{\mathtt{T}}$ denotes a sequence of types $\mathtt{T}_0, ...\mathtt{T}_n$. As in CMG, we abuse this notation in different contexts. For example, $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$; denotes $\mathtt{T}_0\ \mathtt{f}_0; ...\mathtt{T}_n\ \mathtt{f}_n;$. Similiarly, the expression $\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}$ represents $\mathtt{X}_0 \lhd \mathtt{N}_0, ...\mathtt{X}_n \lhd \mathtt{N}_n$.

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

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 $\lhd$. Every class definition declares a super class using $\lhd$.

CCG requires explicit polymorphism on all parametric method invocations.

8.2 Valid Programs

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

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

A valid bind table `bds` must satisfy several constraints: (i) for every `X` in `bds`, `bds(X)` = `bind` $\mathbb{N}$ `X =` $\mathbb{N}$;, (ii), every type variable appearing in `bds` is in $dom($`bds`$)$. Using the set of binds `bind` $\overline{\mathbb{N}_0}$ $\overline{\texttt{X}}$ `=` $\overline{\mathbb{N}}$, an initial bound environment, mapping type variables to their upper bound, is defined as $\Delta_0 = \overline{\texttt{X}} \lhd \overline{\mathbb{N}}$. The bound environment Δ is discussed below in Section 8.4.

Program execution consists of evaluating `e` in the context of MT and a bound environment Δ_0 containing `bds`.

8.3 Valid Module Binds

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

$CT(\mathtt{D}\texttt{<}\overline{\mathtt{T}}\texttt{>.C}) = [\overline{\mathtt{Y}} \mapsto \mathit{bound}_{\Delta_0}(\overline{\mathtt{T}})]$

`class D<`$\overline{\mathtt{Y}}$`>.C<`$\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}$`> ◁ 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.

A valid class table must satisfy the following constriants: (i) for every fully-qualified class name `D<`$\overline{\mathtt{T}}$`>.C` appearing in $dom(CT)$, $CT(\mathtt{D}\texttt{<}\overline{\mathtt{T}}\texttt{>.C}) =$ `class D<`$\overline{\mathtt{T}}$`>.C...` (ii) `Object` $\notin dom(CT)$, (iii) every class name appearing in CT is in $dom(CT)$ and (iv) the set of class definitions must form a tree rooted at `Object`.

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.

8.4 Type Checking

The typing rules for expressions, method declarations, constructor declarations, class declarations, and module declarations are shown in Table 8.2. The typing rules in CCG includes two environments:

- A bounds environment Δ mapping type variables to their upper bounds. Syntactically, this is written as $\overline{\mathtt{X} \lhd \{\mathtt{N} \cup \mathbb{N}\}}$. 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 `V.C<`$\overline{\mathtt{T}}$`>` is the class type `C<`$\overline{\mathtt{T}}$`>` prefixed by the bound of the enclosing module `V`.

*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.

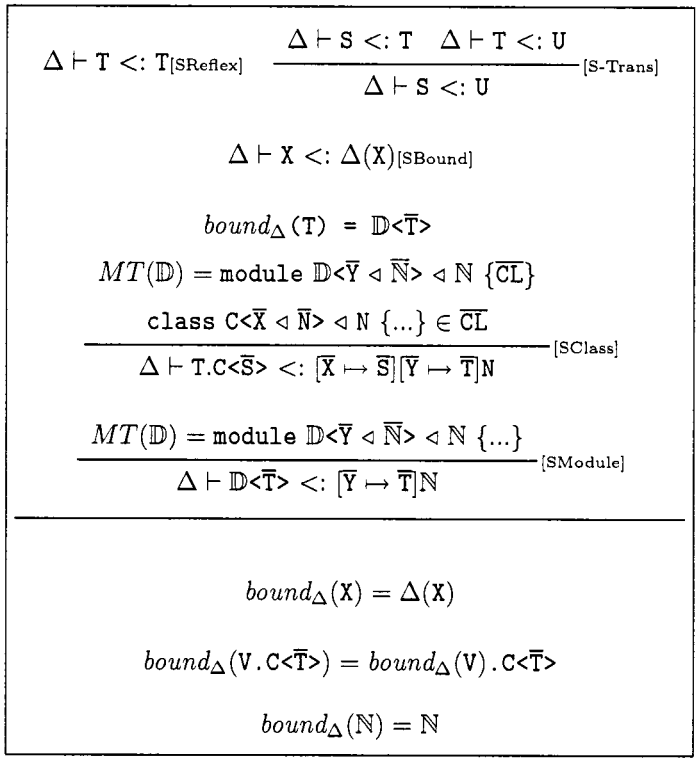


Table 8.2 : Subtyping and Type Bounds

- A type environement Γ mapping program variables to their static types. Syntatically, these mappings have the form $\overline{x} : \overline{T}$

CCG contains two disjoint sets of types: module types and class types. Class types are always qualified with their enclosing module instantiation. Type variables can be bound to either of these types. Figure 8.2 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.

8.5 Well-formed Types and Declarations

The rules for well-formed constructs appear in Tables 8.3 and 8.4. Class and module instantiations are well-formed in the environment Δ if all instantiations of type parameters are

subtypes of their formal types 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`.

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 consitutent elements are well-formed and none of the fields known statically to occur in ancestors are shadowed, every constructor has a distinct signature, and `CL` is a valid override of any class of the same name in the static type of the parent module for $\mathbb{D}$.

Module definitions are well-formed if their constituent elements are well-formed and each member class has a distinct name.

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 an empty type environment (Γ) and bound environment Δ_0.

8.6 Fields, Constructors and Methods

The auxiliary functions to lookup field names and values, method typing, method lookup, valid method overrides, constructor inclusion, and method inclusion appear in Tables 8.6, 8.7, and 8.8. These functions are passed 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

$$\Delta \vdash \texttt{Object ok}_{[\text{WFObject}]} \quad \Delta \vdash \texttt{Mod ok}_{[\text{WFMod}]} \quad \frac{\texttt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \texttt{X ok}}_{[\text{WFVar}]}$$

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

Table 8.3 : Well-formed Types

passed to *fieldVals* to disambiguate field names in the presence of accidental overrides.

Method types are determined by searching upward from a provided static 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 expresssions in the typing rules for method invocation. As explained later in Section 8.7, the annotated type of the receiver of an application of method `m` is reduced to a more specifc 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.

8.7 Expression Typing

The rules for expression typing are give in Table 8.5. 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

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

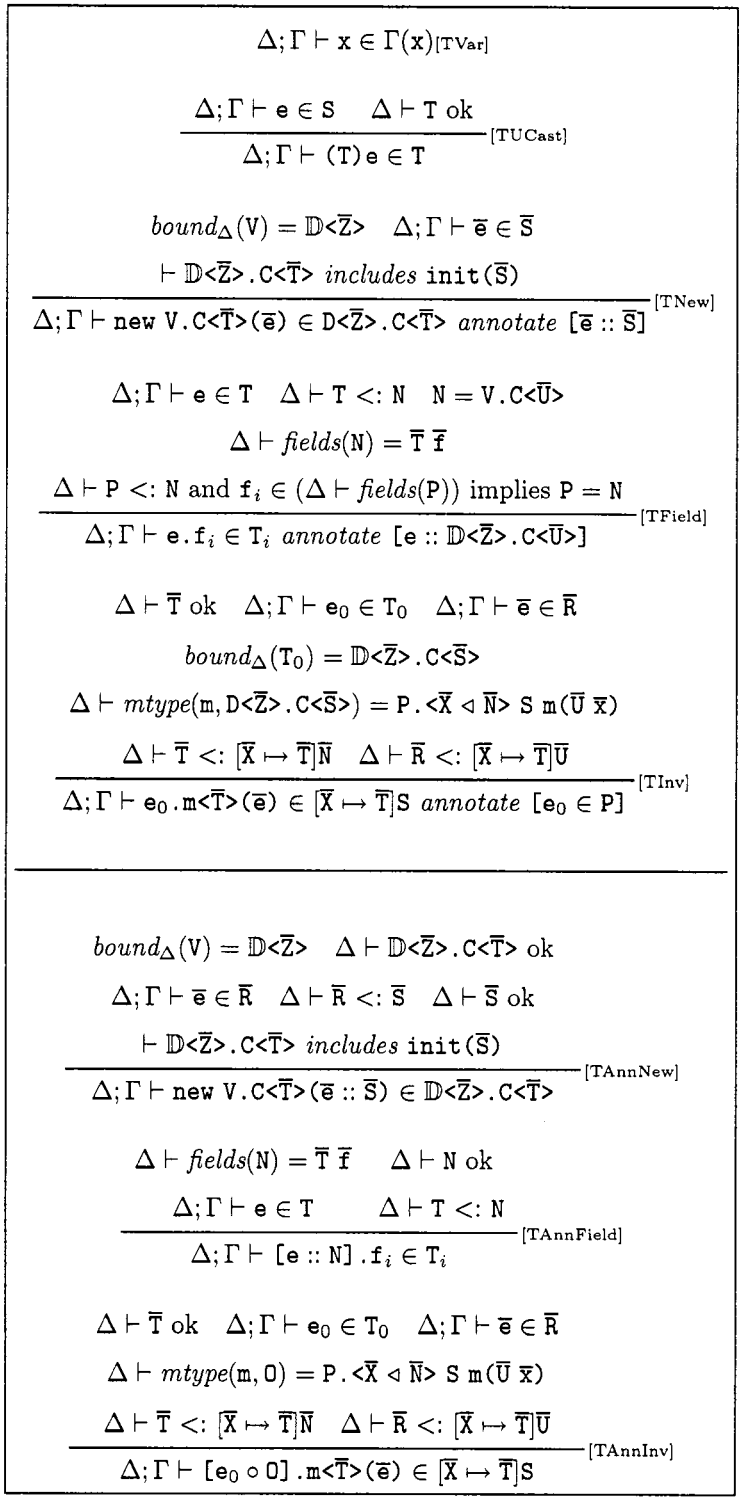
$$\frac{MT(\mathbb{D}) = \texttt{module}\ \mathbb{D}\texttt{<}\overline{\texttt{Y}} \lhd \overline{\mathbb{N}}\texttt{>} \lhd \mathbb{N}\ \{\overline{\texttt{CL}}\} \quad \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{R}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\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

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

$$\frac{\overline{\texttt{CL}}\ \text{ok in}\ \mathbb{D}\texttt{<}\overline{\texttt{Y}} \lhd \overline{\mathbb{N}}\texttt{>} \quad \overline{\mathbb{N}} <: \texttt{Mod} \quad \Delta = \overline{\texttt{Y}} \lhd \overline{\mathbb{N}} \quad \Delta \vdash \overline{\mathbb{N}}\ \text{ok} \quad \overline{\texttt{Y}} \lhd \overline{\mathbb{N}} \vdash \mathbb{N}\ \text{ok} \quad \texttt{CL}_i = \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{T}}\texttt{>} \lhd \texttt{N}\ \{...\}\ \text{and}\ \texttt{CL}_j = \texttt{class C<}\overline{\texttt{X}'} \lhd \overline{\texttt{T}'}\texttt{>} \lhd \texttt{N}'\ \{...\}\ \text{implies}\ i = j}{\texttt{module}\ \mathbb{D}\texttt{<}\overline{\texttt{Y}} \lhd \overline{\mathbb{N}}\texttt{>} \lhd \mathbb{N}\ \{\overline{\texttt{CL}}\}\ \text{ok}}\ [\text{TModule}]$$

$$\frac{\Delta_0 \vdash \mathbb{N}\ \text{ok} \quad \Delta_0 \vdash \mathbb{N} <: \mathbb{N}_0}{\texttt{bind}\ \mathbb{N}_0\ \texttt{X}\ \texttt{=}\ \mathbb{N};\ \text{ok}}\ [\text{TBind}]$$

Table 8.4 : Well-formed Declarations



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

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

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

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

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

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

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

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

Table 8.5 : Expression Typing

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

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

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

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

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

Table 8.6 : Fields

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 8.6.

Notice that receiver expressions of method invocations are annotated not with their static types per se, but instead with the closest supertype of the static type in which the called method is defined. The method found in that supertype is the only method of that name that is statically guaranteed to exist. During computation, the annotated type is reduced whenever possible, modeling the downward search semantics of hygienic mixin method overriding.

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

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

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

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

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

Table 8.7 : Constructors and Methods - 1/2

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 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

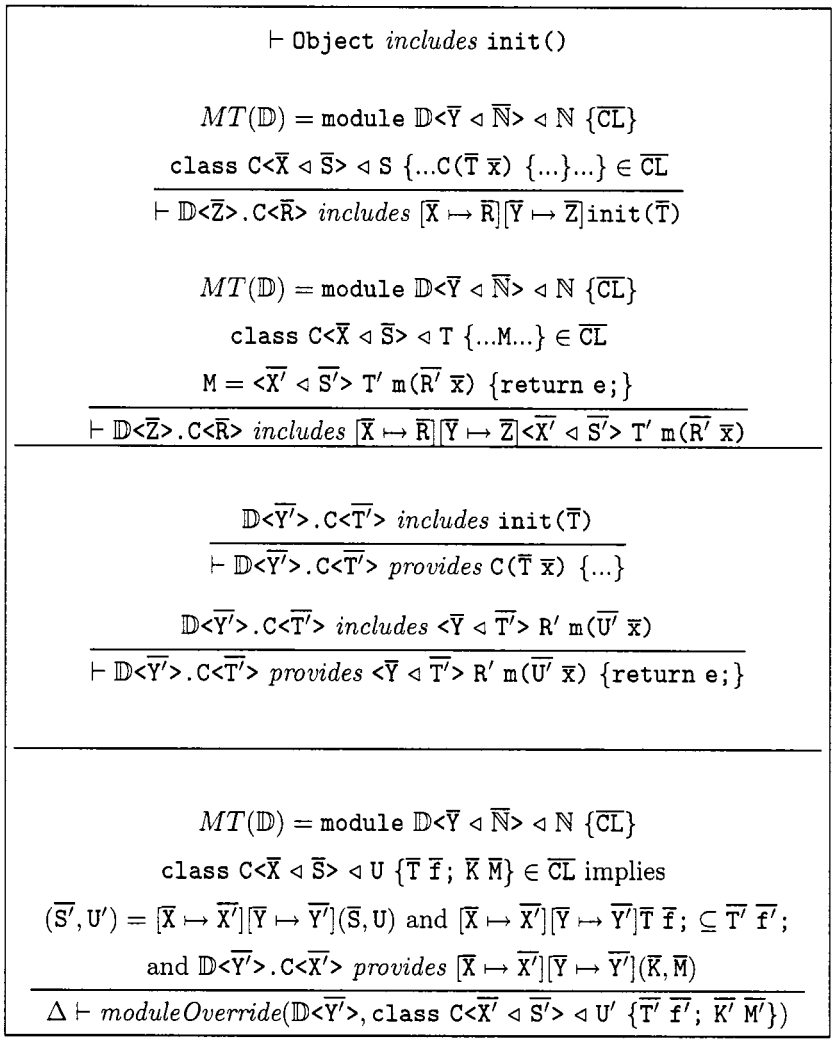


Table 8.8 : Constructors and Methods - 2/2

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 deteremined by the typing rules to change during computation. Just as type annotations play a crucial rule in peserving information in the computation rules, the must 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. "Stupid casts" can be detected either when a ground type is cast to an incompatible ground type. 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.

8.8 Computation

The rules for Computation are contained in Figure 8.9. Computation is specified by small-step semantics the static type of a receiver is used to resolve method appliations 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 in CGEN carry a Δ in computation representing the available bind declarations.

When computing the application of a method, the appropriate method body is found according to the mapping `mbody`. The application is then reduced to the body of the method, substituting all parameters with their instantiations, and `this` with the receiver. Because it is important that a method application is not reduced until the most specific matching type annotation of the receiver is found, two separate forms are used for type annotations. The original type annotation marks the receiver with an annotation of the form $\in$ T. This form of annotation is kept until no further reduction of the static type is possible. At that point, the form of the annotation is switched to :: T. Because the computation rules dictate that methods can be applied only on receivers whose annotations are of the latter form, we 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.

$$\frac{\Delta \vdash \mathit{mbody}(\mathtt{m}\langle\overline{\mathtt{V}}\rangle, \mathtt{N}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\Delta \vdash [\mathtt{new\ V.C}\langle\overline{\mathtt{S}}\rangle(\overline{\mathtt{e}} :: \mathtt{P}) :: \mathtt{N}]\,.\mathtt{m}\langle\overline{\mathtt{V}}\rangle(\overline{\mathtt{d}}) \rightarrow [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}][\mathtt{this} \mapsto \mathtt{new\ V.C}\langle\overline{\mathtt{S}}\rangle(\overline{\mathtt{e}} :: \overline{\mathtt{P}})]\mathtt{e}_0}\ [\text{RInv}]$$

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{e} \in \mathtt{N} \quad \Delta \vdash \mathtt{N} <: \mathtt{V.C}\langle\overline{\mathtt{U}}\rangle \quad \mathit{bound}_\Delta(\mathtt{V}) = \mathbb{D}\langle\overline{\mathtt{T}}\rangle \\ MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\langle\overline{\mathtt{Y}} \lhd \overline{\mathtt{N}}\rangle \lhd \mathbb{N}\ \{\overline{\mathtt{CL}}\} \\ \mathtt{class\ C}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{S}}\rangle \lhd \mathtt{T}\ \{...\} \in \overline{\mathtt{CL}} \\ \Delta \vdash \mathit{mtype}(\mathtt{m}, \mathtt{D}\langle\overline{\mathtt{Z}}\rangle\mathtt{.C}\langle\overline{\mathtt{U}}\rangle) = \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{Z}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{T})\end{array}}{\Delta \vdash [\mathtt{e} \in [\overline{\mathtt{Y}} \mapsto \overline{\mathtt{Z}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{T}]\,.\mathtt{m}\langle\overline{\mathtt{V}}\rangle(\overline{\mathtt{d}}) \rightarrow [\mathtt{e} \in \mathtt{D}\langle\overline{\mathtt{Z}}\rangle\mathtt{.C}\langle\overline{\mathtt{U}}\rangle]\,.\mathtt{m}\langle\overline{\mathtt{V}}\rangle(\overline{\mathtt{d}})}\ [\text{RInvSub}]$$

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

$$\frac{\Delta \vdash \mathtt{N} <: \mathtt{T}}{\Delta \vdash \mathtt{(T)new\ N}(\overline{\mathtt{e}} :: \overline{\mathtt{S}}) \rightarrow \mathtt{new\ N}(\overline{\mathtt{e}} :: \overline{\mathtt{S}})}\ [\text{RCast}] \qquad \frac{\Delta \vdash \mathit{fields}(\mathtt{R}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \quad \Delta \vdash \mathit{fieldVals}(\mathtt{new\ N}(\overline{\mathtt{e}}), \mathtt{R}) = \overline{\mathtt{e'}}}{\Delta \vdash [\mathtt{new\ N}(\overline{\mathtt{e}}) :: \mathtt{R}]\,.\mathtt{f}_i \rightarrow \mathtt{e}'_i}\ [\text{RField}]$$

$$\frac{\Delta \vdash \mathtt{e}_i \rightarrow \mathtt{e}_i{}'}{\Delta \vdash \mathtt{new\ T}(..., \mathtt{e}_i :: \mathtt{S}, ...) \rightarrow \mathtt{new\ T}(..., \mathtt{e}_i{}' :: \mathtt{S}, ...)}\ [\text{RCNewArg}]$$

$$\frac{\Delta \vdash \mathtt{e} \rightarrow \mathtt{e}'}{\Delta \vdash [\mathtt{e} \circ \mathtt{N}]\,.\mathtt{m}\langle\overline{\mathtt{V}}\rangle(\overline{\mathtt{d}}) \rightarrow [\mathtt{e}' \circ \mathtt{N}]\,.\mathtt{m}\langle\overline{\mathtt{V}}\rangle(\overline{\mathtt{d}})}\ [\text{RCInvRecv}]$$

$$\frac{\Delta \vdash \mathtt{e}_i \rightarrow \mathtt{e}_i{}'}{\Delta \vdash [\mathtt{e} \circ \mathtt{N}]\,.\mathtt{m}\langle\overline{\mathtt{V}}\rangle(...\mathtt{e}_i...) \rightarrow [\mathtt{e} \circ \mathtt{N}]\,.\mathtt{m}\langle\overline{\mathtt{V}}\rangle(...\mathtt{e}_i{}'...)}\ [\text{RCInvArg}]$$

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

Table 8.9 : Computation

8.9 Type Soundness

In this section we establish a proof of type soundness for Core CGEN. The proof of type soundness for CCG is based on the proofs for type soundness for FGJ and CMG.

Lemma 1 (Weakening). *Suppose* $\Delta + \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \vdash \overline{\mathtt{N}}$ *ok and* $\Delta \vdash \mathtt{U}$ *ok.*

1. *If* $\Delta \vdash \mathtt{S} <: \mathtt{T}$, *then* $\Delta + \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \vdash \mathtt{S} <: \mathtt{T}$

2. *If* $\Delta \vdash \mathtt{S}$ *ok, then* $\Delta + \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \vdash \mathtt{S}$ *ok.*

3. *If* $\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}$, *then* $\Delta; \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{e} \in \mathtt{T}$ *and* $\Delta + \overline{\mathtt{X}} \lhd \overline{\mathtt{N}}; \Gamma \vdash \mathtt{e} \in \mathtt{T}$.

Proof. Each of the above cases is proved by straightforward induction on the derivation of $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Delta \vdash \mathtt{S}$ ok and $\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}$. □

Lemma 2 (Type Substitution Preserves Fields). *For bound environment* Δ s.t. $\overline{\mathtt{X}} \notin dom(\Delta)$, *types* $\overline{\mathtt{U}}$ *and non-variable class type* $\mathtt{N}$, *if* $\Delta \vdash fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ *then* $\Delta \vdash fields([\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{N}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{T}}\ \overline{\mathtt{f}}$

Proof. Case analysis over the derivation of $\Delta \vdash fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$

Case $\Delta \vdash fields(\mathtt{Object}) = \bullet$: Trivial.

Case $\Delta \vdash fields(\mathtt{V.C<}\overline{\mathtt{S}}\mathtt{>}) = [\overline{\mathtt{R}} \mapsto \overline{\mathtt{S}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{Z}}]\overline{\mathtt{T}}\ \overline{\mathtt{f}}$: Assume $bound_\Delta(\mathtt{V}) = \mathbb{D}\mathtt{<}\overline{\mathtt{Z}}\mathtt{>}$ and $MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\ \{\overline{\mathtt{CL}}\}$ and $\mathtt{class\ C<}\overline{\mathtt{R}'} \lhd \overline{\mathtt{S}'}\mathtt{>} \lhd \mathtt{T}'\ \{...\} \in \overline{\mathtt{CL}}$. We must show that $\Delta \vdash fields([\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathbb{D}\mathtt{<}\overline{\mathtt{Z}}\mathtt{>.C<}\overline{\mathtt{S}}\mathtt{>}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}][\overline{\mathtt{R}} \mapsto \overline{\mathtt{S}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{Z}}]\overline{\mathtt{T}}\ \overline{\mathtt{f}}$. Since $\overline{\mathtt{X}} \notin dom(\Delta)$, we know $[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathbb{D}\mathtt{<}\overline{\mathtt{Z}}\mathtt{>.C<}\overline{\mathtt{S}}\mathtt{>} = \mathbb{D}\mathtt{<}[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{Z}}\mathtt{>.C<}[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{S}}\mathtt{>}$. Then we have $fields(\mathbb{D}\mathtt{<}[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{Z}}\mathtt{>.C<}[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{S}}\mathtt{>}) = [\overline{\mathtt{R}} \mapsto [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{S}}][\overline{\mathtt{Y}} \mapsto [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{Z}}]\overline{\mathtt{T}}\ \overline{\mathtt{f}} = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}][\overline{\mathtt{R}} \mapsto \overline{\mathtt{S}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{Z}}]\overline{\mathtt{T}}\ \overline{\mathtt{f}}$.

□

Lemma 3 (Type Substitution Preserves Method Types). *For bound environment* Δ s.t. $\overline{\mathtt{X}} \notin dom(\Delta)$, *types* $\overline{\mathtt{U}}$ *and non-variable class type* $\mathtt{N}$, *if* $\Delta \vdash mtype(\mathtt{m}, \mathtt{N}) = \mathtt{N.<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{R}}\mathtt{>\ T\ m(}\overline{\mathtt{U}'}\ \overline{\mathtt{x}}\mathtt{)}$ *then* $\Delta \vdash mtype(\mathtt{m}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{N}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}](\mathtt{N.<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{R}}\mathtt{>\ T\ m(}\overline{\mathtt{U}'}\ \overline{\mathtt{x}}\mathtt{)})$.

Proof. The premise $\Delta \vdash \mathit{mtype}(\mathtt{m}, \mathtt{N}) = \mathtt{N}.\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{R}}\mathtt{>}\ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})$ matches only one of the two rules that defines *mtype*. And this rule applies equally well to the substituted form similar to Lemma 2. □

Lemma 4 (Type Substitution Preserves Subtyping). *For bound environment* Δ s.t. $\overline{\mathtt{X}} \notin \mathit{dom}(\Delta)$, *types* $\overline{\mathtt{U}}$, *if* $\Delta + \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \vdash \mathtt{S} <: \mathtt{T}$ *and* $\Delta \vdash \overline{\mathtt{U}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{N}}$ *then* $\Delta \vdash [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{S} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{T}$.

Proof. By structural induction over the derivation of $\Delta + \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \vdash \mathtt{S} <: \mathtt{T}$

Case S-Reflex: Trivial.

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

Case S-Bound: $\mathtt{S} = \mathtt{X}$. if $\mathtt{X} \in \mathit{dom}(\Delta)$, then it's trivial. On the other hand if $\mathtt{S} = \mathtt{X}_i$, $\mathtt{T} = \mathtt{N}_i$, $\Delta + \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \vdash \mathtt{S} <: \mathtt{N}_i$. Then $[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{S} = \mathtt{U}_i$ and $[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{T} = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{N}_i$. But we're given that $\Delta \vdash \overline{\mathtt{U}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{N}}$, finishing the case.

Case S-Module: $\mathtt{S} = \mathbb{D}\mathtt{<}\overline{\mathtt{Z}}\mathtt{>}$, where $MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathbb{N}\ \{\overline{\mathtt{CL}}\}$ and $[\overline{\mathtt{Y}} \mapsto \overline{\mathtt{Z}}]\mathbb{N} = \mathtt{T}$. We must show that $\Delta \vdash [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathbb{D}\mathtt{<}\overline{\mathtt{Z}}\mathtt{>} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{Z}}]\mathbb{N}$. But notice that $[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathbb{D}\mathtt{<}\overline{\mathtt{Z}}\mathtt{>} = \mathbb{D}\mathtt{<}[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{Z}}\mathtt{>}$. Then by [S-Module], $\Delta \vdash \mathbb{D}\mathtt{<}[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{Z}}\mathtt{>} <: [\overline{\mathtt{Y}} \to [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{Z}}]\mathbb{N}$. But, $[\overline{\mathtt{Y}} \to [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{Z}}]\mathbb{N} = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{Z}}]\mathbb{N}$, finishing this case.

Case S-Class: Similar. $\mathtt{S} = \mathbb{D}\mathtt{<}\overline{\mathtt{Z}}\mathtt{>.C<}\overline{\mathtt{V}}\mathtt{>}$ where $MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathbb{N}\ \{\overline{\mathtt{CL}}\}$, $\mathtt{class\ C<}\overline{\mathtt{R}} \lhd \overline{\mathtt{S}}\mathtt{>} \lhd \mathtt{T}'\ \{...\} \in \overline{\mathtt{CL}}$, and $[\overline{\mathtt{R}} \mapsto \overline{\mathtt{V}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{Z}}]\mathtt{T}' = \mathtt{T}$. We must show that $\Delta \vdash [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathbb{D}\mathtt{<}\overline{\mathtt{Z}}\mathtt{>.C<}\overline{\mathtt{V}}\mathtt{>} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}][\overline{\mathtt{R}} \mapsto \overline{\mathtt{V}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{Z}}]\mathtt{T}'$. But notice that $[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathbb{D}\mathtt{<}\overline{\mathtt{Z}}\mathtt{>.C<}\overline{\mathtt{V}}\mathtt{>} = \mathbb{D}\mathtt{<}[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{Z}}\mathtt{>.C<}[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{V}}\mathtt{>}$. Next, by applying [S-Class], we can further reason $\Delta \vdash \mathbb{D}\mathtt{<}[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{Z}}\mathtt{>.C<}[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{V}}\mathtt{>} <: [\overline{\mathtt{R}} \to [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{V}}][\overline{\mathtt{Y}} \to [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{Z}}]\mathtt{T}'$. But, $[\overline{\mathtt{R}} \to [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{V}}][\overline{\mathtt{Y}} \to [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{Z}}]\mathtt{T}' = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}][\overline{\mathtt{R}} \mapsto \overline{\mathtt{V}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{Z}}]\mathtt{T}'$, finishing the case.

□

Lemma 5 (Type Substitution Preserves Well-formedness). *For bound environment* Δ *where* $\overline{\mathtt{X}} \notin \mathit{dom}(\Delta)$, *and* $\Delta \vdash \overline{\mathtt{U}}$ *ok, if* $\Delta + \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \vdash \mathtt{S}$ *ok,* $\Delta \vdash \overline{\mathtt{U}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{N}}$ *then* $\Delta \vdash [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{S}$ *ok.*

Proof. By structural induction over the derivation of $\Delta + \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \vdash \mathtt{S}$ ok

Case WF-Object: Trivial.

Case WF-Mod: Trivial.

Case WF-Var: $\Delta + \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \vdash \mathtt{S}$ ok. If $\mathtt{S} \in dom(\Delta)$ this is trivial. Let $\mathtt{S} = \mathtt{X}_i$. Then $[\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{S} = \mathtt{U}_i$. But we are give that $\Delta \vdash \overline{\mathtt{U}}$ ok.

Case WF-Module: $\Delta + \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \vdash \mathbb{D}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ ok. Immediate from Lemma 4 and the induction hypothesis.

Case WF-Class: $\Delta + \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \vdash \mathbb{D}\texttt{<}\overline{\mathtt{T}}\texttt{>.C<}\overline{\mathtt{S}}\texttt{>}$ ok. Immediate from Lemma 4 and the induction hypothesis.

□

8.10 Module Hierarchies

We now build on our formalized notion of ground expressions and types and address the potential for cyclic and infinite hierarchies in CCG. We begin our analysis first with modules, and then in Section 8.11 we examine classes. The following lemmas are used to ensure that the constraints placed on module tables prevents cyclic and infinite hierarchies.

Lemma 6 (Module Compactness). *For bound environment* Δ *and module type* $\mathbb{D}\texttt{<}\overline{\mathtt{N}}\texttt{>}$ s.t. $\Delta \vdash \mathbb{D}\texttt{<}\overline{\mathtt{N}}\texttt{>}$ *ok, there is a finite chain of module types* $\mathtt{P}_0, ..., \mathtt{P}_N$ *s.t. for all* i *s.t.* $1 \leq i \leq N$, $\Delta \vdash \mathtt{P}_{i-1} <: \mathtt{P}_i$ *and* $\mathtt{P}_N = \mathtt{Mod}$.

Proof. A well-formed module table allows only one source for module parent types: modules can specify a specific module in its extends clause. Thus, this condition is required directly on all module instantiations for all well-formed module tables. □

Lemma 7 (Antisymmetry). *For module types* $\mathbb{C}\texttt{<}\overline{\mathtt{N}}\texttt{>}$, $\mathbb{D}\texttt{<}\overline{\mathtt{P}}\texttt{>}$, s.t. $\Delta \vdash \mathbb{C}\texttt{<}\overline{\mathtt{N}}\texttt{>}$ *ok, and* $\Delta \vdash \mathbb{D}\texttt{<}\overline{\mathtt{P}}\texttt{>}$ *ok, if* $\Delta \vdash \mathbb{C}\texttt{<}\overline{\mathtt{N}}\texttt{>} <: \mathbb{D}\texttt{<}\overline{\mathtt{P}}\texttt{>}$ *then either* $\Delta \not\vdash \mathbb{D}\texttt{<}\overline{\mathtt{P}}\texttt{>} <: \mathbb{C}\texttt{<}\overline{\mathtt{N}}\texttt{>}$ *or* $\mathbb{C}\texttt{<}\overline{\mathtt{N}}\texttt{>} = \mathbb{D}\texttt{<}\overline{\mathtt{P}}\texttt{>}$.

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

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

Case S-Class: Impossible since $\mathbb{C}\langle\overline{\mathtt{N}}\rangle$ is a module type.

Case S-Module: Then $\mathbb{D}\langle\overline{\mathtt{P}}\rangle$ is a parent of $\mathbb{C}\langle\overline{\mathtt{T}}\rangle$. Therefore, the constraints on module heirarchy also require $\Delta \nvdash \mathbb{D}\langle\overline{\mathtt{P}}\rangle <: \mathtt{C}\langle\overline{\mathtt{N}}\rangle$.

Case S-Trans: In this case, there exists some $\mathtt{T}$ s.t. $\Delta \vdash \mathbb{C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{T}$ and $\Delta \vdash \mathtt{T} <: \mathbb{D}\langle\overline{\mathtt{P}}\rangle$. If $\mathbb{C}\langle\overline{\mathtt{N}}\rangle = \mathbb{D}\langle\overline{\mathtt{P}}\rangle$, we are done, so assume $\mathbb{C}\langle\overline{\mathtt{N}}\rangle \neq \mathbb{D}\langle\overline{\mathtt{P}}\rangle$ By the induction hypothesis, either $\mathbb{C}\langle\overline{\mathtt{N}}\rangle = \mathtt{T}$ or $\Delta \nvdash \mathtt{T} <: \mathbb{C}\langle\overline{\mathtt{N}}\rangle$. But if $\mathbb{C}\langle\overline{\mathtt{N}}\rangle = \mathtt{T}$, then $\Delta \vdash \mathbb{C}\langle\overline{\mathtt{N}}\rangle <: \mathbb{D}\langle\overline{\mathtt{P}}\rangle$ was already derived as a premise to [S-Trans], and then by the induction hypothesis, $\Delta \nvdash \mathbb{D}\langle\overline{\mathtt{P}}\rangle <: \mathbb{N}\langle\overline{\mathtt{N}}\rangle$. Finally, consider the case that $\mathbb{C}\langle\overline{\mathtt{N}}\rangle \neq \mathtt{T}$. Then we can show that by contradiction, if $\Delta \vdash \mathbb{D}\langle\overline{\mathtt{P}}\rangle <: \mathbb{C}\langle\overline{\mathtt{N}}\rangle$. implies that $\Delta \vdash \mathtt{T} <: \mathbb{C}\langle\overline{\mathtt{N}}\rangle$ which contradicts the induction hypothesis. So $\Delta \nvdash \mathbb{D}\langle\overline{\mathtt{P}}\rangle <: \mathbb{C}\langle\overline{\mathtt{N}}\rangle$.

□

Lemma 8 (Uniqueness). *For module type* $\mathbb{C}\langle\overline{\mathtt{N}}\rangle$ s.t. $\Delta \vdash \mathbb{C}\langle\overline{\mathtt{N}}\rangle$ *ok, there is exactly one type* $\mathtt{P} \neq \mathbb{C}\langle\overline{\mathtt{N}}\rangle$ *(i.e., the declared parent instantiation) s.t. both of the following conditions hold:*

1. $\Delta \vdash \mathbb{C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{P}$
2. *If* $\Delta \vdash \mathbb{C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}$, $\mathbb{C}\langle\overline{\mathtt{N}}\rangle \neq \mathtt{O}$, *and* $\Delta \vdash \mathtt{O} <: \mathtt{P}$ *then* $\mathtt{O} = \mathtt{P}$.

Proof. Let $\mathtt{P}$ be the declared parent instantiation of $\mathbb{C}\langle\overline{\mathtt{N}}\rangle$. Suppose for a contradiction that there was a type $\mathtt{O} \neq \mathtt{P}$ s.t. $\Delta \vdash \mathbb{C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}$, and $\Delta \vdash \mathtt{O} <: \mathtt{P}$. Then there is some finite derivation of $\Delta \vdash \mathbb{C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}$. Consider a shortest derivation of $\Delta \vdash \mathbb{C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}$, i.e., a derivation employing no more rule applications than any other derivation. Such a derivation can't conclude with [S-Reflex] because $\mathbb{C}\langle\overline{\mathtt{N}}\rangle \neq \mathtt{O}$, nor [S-Class] because $\mathbb{C}\langle\overline{\mathtt{N}}\rangle$ is a module type. Also, it can't conclude with [S-Module] because $\mathtt{P} \neq \mathtt{O}$. Thus it must conclude with [S-Trans]. Then there is some type $\mathtt{O}'$ s.t. $\Delta \vdash \mathbb{C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}'$ and $\Delta \vdash \mathtt{O}' <: \mathtt{O}$.

Similarly, a shortest derivation of $\Delta \vdash \mathbb{C}\texttt{<}\overline{\texttt{N}}\texttt{>} <: \texttt{O}'$ can't conclude with [S-Reflex]; otherwise our derivation of $\Delta \vdash \mathbb{C}\texttt{<}\overline{\texttt{N}}\texttt{>} <: \texttt{O}$ is not the shortest derivation, nor [S-Class] since $\mathbb{C}\texttt{<}\overline{\texttt{N}}\texttt{>}$ is a module type. Our derivation of $\Delta \vdash \mathbb{C}\texttt{<}\overline{\texttt{N}}\texttt{>} <: \texttt{O}'$ can't conclude with [S-Module]; otherwise $\texttt{O}' = \texttt{P}$, which is impossible by Lemma 7. Thus, a shortest derivation of $\Delta \vdash \mathbb{C}\texttt{<}\overline{\texttt{N}}\texttt{>} <: \texttt{O}'$ must conclude with [S-Trans]. Continuing in this fashion, we can show that at each step in our derivation of $\Delta \vdash \mathbb{C}\texttt{<}\overline{\texttt{N}}\texttt{>} <: \texttt{O}$, the rule [S-Trans] must be employed, requiring yet another step in the derivation. Thus, no finite length derivation could conclude with $\Delta \vdash \mathbb{C}\texttt{<}\overline{\texttt{N}}\texttt{>} <: \texttt{O}$, so $\Delta \not\vdash \mathbb{C}\texttt{<}\overline{\texttt{N}}\texttt{>} <: \texttt{O}$.$\otimes$ Therefore, type O does not exist. □

8.11 Class Hierarchies

The presence of recursive modules in CGEN allows for the possibility of cyclic class hierarchies. We must ensure that the constraints placed on modules and bind declarations prevents cyclic class hierarchies from from forming. We can guarantee the sanity of CCG class hierarchies with the following three lemmas:

Lemma 9 (Compactness). *For bound environment* Δ *and class type* $\texttt{T.C}\texttt{<}\overline{\texttt{N}}\texttt{>}$ s.t. $\Delta \vdash \texttt{T.C}\texttt{<}\overline{\texttt{N}}\texttt{>}$ *ok, there is a finite chain of class types* $\texttt{P}_0, ..., \texttt{P}_N$ *s.t. for all* i *s.t.* $1 \leq i \leq N$, $\Delta \vdash \texttt{P}_{i-1} <: \texttt{P}_i$ *and* $\texttt{P}_N = \texttt{Object}$.

Proof. There are two sources for parent types for classes in a well-formed module table: (1) from either a locally defined class or (2) from a class introduced by a module import. We consider the later case to be a mixin type because its parent type is determined during module linking (instantiation).

In the case of non-mixin instantiations, the condition is required directly in the subclassing chain. In the case of mixin instantiations, we can show this condition holds through by contradiction. Assume there exists a mixin instantiation N where this required condition does not hold. This implies that the parent must be a mixin instantiation; otherwise the lemma would obviously be satisfied by the parent instantiation, and by [S-Class] for N as well. Lets call the parent type to be N′. By recursively following our argument, we can show

that the parent type of $\mathtt{N}'$ should likewise be a mixin instantiation $\mathtt{N}''$, and so on. Recall that for an instantiated module $\mathbb{D}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}$, the imported modules $\overline{\mathtt{T}}$ are determined with respect to a bound environment Δ. Since modules are linked through bind declarations, this implies the available bind declarations link a set of modules to create a cyclic class hierarchy . However, this is prohibited by the restrictions on bind declarations. $\bigotimes$ Thus, the condition holds for all mixin instantiations as well as non-mixin class instantiations. □

Lemma 10 (Antisymmetry). *For class types* $\mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$, $\mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}$ s.t. $\Delta \vdash \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$ *ok,* $\Delta \vdash \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}$ *ok, if* $\Delta \vdash \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>} <: \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}$ *then either* $\Delta \nvdash \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>} <: \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$ *or* $\mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>} = \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}$.

Proof. By structural induction on the derivation of $\Delta \vdash \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>} <: \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}$.

Case S-Reflex: Then $\mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>} = \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}$.

Case S-Module: Impossible since $\mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$ is a class type.

Case S-Class: Then $\mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}$ is the parent of $\mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$.

Case S-Trans: In this case, there exists some V s.t. $\Delta \vdash \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>} <: \mathtt{V}$ and $\Delta \vdash \mathtt{V} <: \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}$. If $\mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>} = \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}$, we are done, so assume $\mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>} \neq \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}$ By the induction hypothesis, either $\mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>} = \mathtt{V}$ or $\Delta \nvdash \mathtt{V} <: \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$. But if $\mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>} = \mathtt{V}$, then $\Delta \vdash \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>} <: \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>}$ was already derived as a premise to [S-Trans], and then by the induction hypothesis, $\Delta \nvdash \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>} <: \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$. Finally, consider the case that $\mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>} \neq \mathtt{V}$. Then we can show that by contradiction, if $\Delta \vdash \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>} <: \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$. implies that $\Delta \vdash \mathtt{V} <: \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$ which contradicts the induction hypothesis. So $\Delta \nvdash \mathtt{U.D}\mathtt{<}\overline{\mathtt{P}}\mathtt{>} <: \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$.

□

Lemma 11 (Uniqueness). *For a given class type* $\mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$ s.t. $\Delta \vdash \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$ *ok, there is exactly one type* $\mathtt{P} \neq \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>}$ *(i.e., the declared parent instantiation) s.t. both of the following conditions hold:*

1. $\Delta \vdash \mathtt{T.C}\mathtt{<}\overline{\mathtt{N}}\mathtt{>} <: \mathtt{P}$

2. *If* $\Delta \vdash \mathtt{T.C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}$, $\mathtt{T.C}\langle\overline{\mathtt{N}}\rangle \neq \mathtt{O}$, *and* $\Delta \vdash \mathtt{O} <: \mathtt{P}$ *then* $\mathtt{O} = \mathtt{P}$.

Proof. Let P be the declared parent instantiation of $\mathtt{T.C}\langle\overline{\mathtt{N}}\rangle$. Suppose for a contradiction that there was a type $\mathtt{O} \neq \mathtt{P}$ s.t. $\Delta \vdash \mathtt{T.C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}$, and $\Delta \vdash \mathtt{O} <: \mathtt{P}$. Then there is some finite derivation of $\Delta \vdash \mathtt{T.C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}$. Consider a shortest derivation of $\Delta \vdash \mathtt{T.C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}$, i.e., a derivation employing no more rule applications than any other derivation. Such a derivation can't conclude with [S-Reflex] because $\mathtt{T.C}\langle\overline{\mathtt{N}}\rangle \neq \mathtt{O}$. Also, it can't conclude with [S-Class] because $\mathtt{P} \neq \mathtt{O}$. It cannot conclude with [S-Module] since $\mathtt{T.C}\langle\overline{\mathtt{N}}\rangle$ is a class type. Thus it must conclude with [S-Trans]. Then there is some type $\mathtt{O}'$ s.t. $\Delta \vdash \mathtt{T.C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}'$ and $\Delta \vdash \mathtt{O}' <: \mathtt{O}$. Similarly, a shortest derivation of $\vdash \mathtt{T.C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}'$ can't conclude with [S-Reflex]; otherwise our derivation of $\Delta \vdash \mathtt{T.C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}$ is not the shortest derivation. Also, our derivation of $\Delta \vdash \mathtt{T.C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}'$ can't conclude with [S-Class]; otherwise $\mathtt{O}' = \mathtt{P}$, which is impossible by Lemma 10. Thus, a shortest derivation of $\Delta \vdash \mathtt{T.C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}'$ must conclude with [S-Trans]. Continuing in this fashion, we can show that at each step in our derivation of $\Delta \vdash \mathtt{T.C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}$, the rule [S-Trans] must be employed, requiring yet another step in the derivation. Thus, no finite length derivation could conclude with $\Delta \vdash \mathtt{T.C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}$, so $\Delta \nvdash \mathtt{T.C}\langle\overline{\mathtt{N}}\rangle <: \mathtt{O}$. Therefore, type O does not exist. □

8.12 Preservation

With these lemmas in hand, we now proceed to show that substitution preserves typing in CGEN.

Lemma 12 (Type Substitution Preserves Typing). *For annotated types* $\overline{\mathtt{U}}$ s.t. $\Delta \vdash \overline{\mathtt{U}}$ *ok, if* $\Delta + \overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}; \Gamma \vdash \mathtt{e} \in \mathtt{S}$, $\Delta \vdash \overline{\mathtt{U}} \triangleleft [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\overline{\mathtt{N}}$ *then* $\Delta; [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\Gamma \vdash [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{e} \in [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{S}$.

Proof. By structural induction over the derivation $\Delta + \overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}; \Gamma \vdash \mathtt{e} \in \mathtt{S}$.

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

Case T-Cast: $\mathtt{e} = (\mathtt{S})\mathtt{e}'$ where $\Delta; \Gamma \vdash \mathtt{e}' \in \mathtt{T}$. By Lemma 5, $\Delta \vdash [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{S}$ ok. By the induction hypothesis $\Delta; [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\Gamma \vdash [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{e}' \in [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{T}$. Then by [T-Cast], $\Delta; [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\Gamma \vdash [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{e} \in [\overline{\mathtt{X}} \mapsto \overline{\mathtt{U}}]\mathtt{S}$.

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

Case T-Ann-Invk: $\mathtt{e} = \mathtt{[e_0 \circ O]\ .m\langle\overline{T}\rangle(\overline{e})} \in [\overline{X} \mapsto \overline{T}]S$. All the antecedents in [T-Ann-Invk] except for the method substitution type apply by straightforward application of the induction hypothesis, and supporting substitution lemmas. So we need to show that:

$\Delta \vdash \mathit{mtype}(\mathtt{m}, [\overline{X} \mapsto \overline{U}]\mathtt{O}) = [\overline{X} \mapsto \overline{U}]\mathtt{P}.\langle\overline{X} \lhd \overline{R}\rangle\ \mathtt{T\ m}(\overline{U'}\ \overline{x})$.

There are two cases:

Subcase $\mathtt{O} = \mathtt{P}$: The by Lemma 3, $\Delta \vdash \mathit{mtype}(\mathtt{m}, [\overline{X} \mapsto \overline{U}]\mathtt{O}) = [\overline{X} \mapsto \overline{U}]\mathtt{O}.\langle\overline{X} \lhd \overline{R}\rangle\ \mathtt{T\ m}(\overline{U'}\ \overline{x})$.

Subcase $\mathtt{O} \neq \mathtt{P}$: Because the annotated type of the receiver in the original invocation expression from which e was reduced was determined by [T-Invk], it must have matched P. The only reduction of the original expression which could have modified the annotated type is [R-Inv-Sub]. But the antecedents of [R-Inv-Sub] ensure that an annotated type S is reduced to T only if $\Delta \vdash \mathit{mtype}(\mathtt{m}, \mathtt{S}) = \mathit{mtype}(\mathtt{m}, \mathtt{T})$. Therefore, $\Delta \vdash \mathit{mtype}(\mathtt{m}, \mathtt{O}) = \mathit{mtype}(\mathtt{m}, \mathtt{P})$ and the case is finished by Lemma 3.

□

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

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

Case T-Var: $\mathtt{e} = \mathtt{x}_i$. Then $\vdash [\overline{x} \mapsto \overline{e}]\mathtt{e} = \mathtt{e}_i$. Since $\Delta \vdash e_i \in \mathtt{R}_i$, we have $\mathtt{S'} = \mathtt{R}_i$.

Case T-Cast: $\mathtt{e} = (\mathtt{S})\mathtt{e'}$. By the induction hypothesis, $\vdash [\overline{x} \mapsto \overline{e}]\mathtt{e'}$ is well-typed, so $\Delta \vdash [\overline{x} \mapsto \overline{e}]\mathtt{e} \in \mathtt{S}$.

Case T-Ann-New: $\mathtt{e} = \mathtt{new\ S}(\overline{\mathtt{e'}} :: \overline{\mathtt{R}})$. But $[\overline{\mathtt{x}} \mapsto \overline{\mathtt{e}}]\mathtt{new\ S}(\overline{\mathtt{e'}} :: \overline{\mathtt{R}}) = \mathtt{new\ S}([\overline{\mathtt{x}} \mapsto \overline{\mathtt{e}}]\overline{\mathtt{e'}} :: \overline{\mathtt{R}})$. By the induction hypothesis, $\Delta \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{e}}]\overline{\mathtt{e'}} \in \overline{\mathtt{R'}}$ where $\Delta \vdash \overline{\mathtt{R'}} <: \overline{\mathtt{R}}$. So, by [T-Ann-Field], $\Delta \vdash \mathtt{new\ S}([\overline{\mathtt{x}} \mapsto \overline{\mathtt{e}}]\overline{\mathtt{e'}} :: \overline{\mathtt{R}}) \in \mathtt{S}$.

Case T-Ann-Field: $\mathtt{e} = \mathtt{[e'} :: \mathtt{N]\,.f}$. Term substitution does not effect the annotation N or field f. From the induction hypothesis we know $\Delta \vdash \mathtt{e'} \in \mathtt{N'}$ where $\Delta \vdash \mathtt{N'} <: \mathtt{N}$. So by [T-Ann-Field], $\Delta \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{e}}]\mathtt{e} \in \mathtt{S}$.

Case T-Ann-Invk: $\mathtt{e} = \mathtt{[e_0 \circ O]\,.m<\overline{T}>(\overline{e'})}$. By the induction hypothesis, $[\overline{\mathtt{x}} \mapsto \overline{\mathtt{e}}]\mathtt{e}_0$ is well-typed, as well as $[\overline{\mathtt{x}} \mapsto \overline{\mathtt{e}}]\overline{\mathtt{e'}}$. The other premises of [T-Ann-Invk] are not affected by term substitution, and the static type of the invocation is determined solely by m and the annotated type of the receiver, neither of which are modified by term substitution. So, by [T-Ann-Invk], $\Delta \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{e}}]\mathtt{e} \in \mathtt{S}$.

□

With these lemmas in hand, we are now in a position to establish a subject reduction 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* $\Delta \vdash \mathtt{S} <: \mathtt{T}$.

Proof. By structural induction over the derivation of $\Delta \vdash \mathtt{e} \rightarrow \mathtt{e'}$.

Case R-Cast: $\mathtt{e} = \mathtt{(O)new\ N}(\overline{\mathtt{e}} :: \overline{\mathtt{S}})$. By [T-Cast], $\Delta \vdash \mathtt{e} \in \mathtt{O}$. By [R-Cast], $\Delta \vdash \mathtt{N} <: \mathtt{O}$. Finally, by [T-Ann-New], $\Delta \vdash \mathtt{new\ N}(\overline{\mathtt{e}} :: \overline{\mathtt{S}}) \in \mathtt{N}$, which finishes the case.

Case R-Field: $\mathtt{e} = [\mathtt{new\ T_0.C<\overline{T}>}(\overline{\mathtt{e}}) :: \mathbb{D}'\mathtt{<\overline{V'}>.C'<\overline{T'}>}]\,.\mathtt{f}_i$. By [R-Field], $\mathtt{e'} = \Delta \vdash \mathit{fieldVals}(\mathtt{new\ T_0.C<\overline{T}>}(\overline{\mathtt{e}} :: \overline{\mathtt{R}}), \mathbb{D}'\mathtt{<\overline{V'}>.C'<\overline{T'}>})_i$. Let $\Delta \vdash \mathit{fields}(\mathbb{D}'\mathtt{<\overline{V'}>.C'<\overline{T'}>}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}}$ and $\mathit{bound}_\Delta(\mathtt{T_0}) = \mathbb{D}\mathtt{<\overline{V}>}$. By [T-Ann-Field], $\Delta \vdash \mathtt{e} \in \mathtt{S}_i$. Let

$MT(\mathbb{D}) = \mathtt{module}\ \mathbb{D}\mathtt{<\overline{Y} \lhd \overline{N}> \lhd N}\ \{\overline{\mathtt{CL}}\}$

$\mathtt{class\ C<\overline{X} \lhd \overline{N}> \lhd N}\ \{...\mathtt{C}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\{...\}...\} \in \overline{\mathtt{CL}}$,

$MT(\mathbb{D}') = \texttt{module}\ \mathbb{D}'\texttt{<}\overline{\texttt{Y}'} \lhd \overline{\mathbb{N}'}\texttt{>} \lhd \mathbb{N}'\ \{\overline{\texttt{CL}'}\}$
$\texttt{class}\ \texttt{C}'\texttt{<}\overline{\texttt{X}'} \lhd \overline{\texttt{N}'}\texttt{>} \lhd \texttt{N}'\ \{...\} \in \overline{\texttt{CL}'}$,

We show by induction over the derivation of $\texttt{e}' = \Delta \vdash \mathit{fieldVals}(\texttt{new C<}\overline{\texttt{U}}\texttt{>(}\overline{\texttt{e}}\texttt{),D<}\overline{\texttt{V}}\texttt{>})_i$ that $\Delta \vdash \texttt{S} <: \texttt{S}_i$. There are two possibilities:

Subcase $\mathbb{D}\texttt{<}\overline{\texttt{V}}\texttt{>.C<}\overline{\texttt{T}}\texttt{>} = \mathbb{D}'\texttt{<}\overline{\texttt{V}'}\texttt{>.C}'\texttt{<}\overline{\texttt{V}'}\texttt{>}$: Because we are given that $\Delta \vdash \texttt{e} \rightarrow \texttt{e}'$, it must be the case that $\Delta \vdash \mathit{fieldVals}(\texttt{new}\ \texttt{T}_0\texttt{.C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{),}\mathbb{D}\texttt{<}\overline{\texttt{V}}\texttt{>.C<}\overline{\texttt{T}}\texttt{>})_i = [\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}][\overline{\texttt{x}} \mapsto \overline{\texttt{e}}]\texttt{e}'_i$ where $\overline{\texttt{e}'}$ are the expressions assigned to the fields of class C in the matching constructor. By [R-Constructor], we know that $\Delta + \overline{\texttt{V}} \lhd \overline{\texttt{Y}} + \overline{\texttt{X}} \lhd \overline{\texttt{N}}; \overline{\texttt{x}} : \overline{\texttt{T}} \vdash \texttt{e}'_i \in \texttt{S}'_i$ where $\texttt{S}_i = [\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}]\texttt{S}'_i$. Since $\Delta \vdash \overline{\texttt{T}}$ ok, we know by Lemma 12 and 13 that $\Delta \vdash [\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}][\overline{\texttt{x}} \mapsto \overline{\texttt{e}}]\texttt{e}'_i \in [\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}]\texttt{S}'_i$.

Subcase $\mathbb{D}\texttt{<}\overline{\texttt{V}}\texttt{>.C<}\overline{\texttt{T}}\texttt{>} \neq \mathbb{D}'\texttt{<}\overline{\texttt{V}'}\texttt{>.C}'\texttt{<}\overline{\texttt{V}'}\texttt{>}$: Then $\Delta \vdash \mathit{fieldVals}(\texttt{new}\ \texttt{T}_0\texttt{.C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{),}\mathbb{D}'\texttt{<}\overline{\texttt{V}'}\texttt{>.C}'\texttt{<}\overline{\texttt{T}'}\texttt{>}) = \mathit{fieldVals}(\texttt{new}\ [\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}]\texttt{N}([\overline{\texttt{x}} \mapsto \overline{\texttt{e}}]\overline{\texttt{e}''})$, where $\texttt{e}''$ are the arguments passed in the super-constructor call within the matching constructor of C. But by the induction hypothesis, $\Delta \vdash \mathit{fieldVals}((\texttt{new}\ [\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}]\texttt{N}([\overline{\texttt{x}} \mapsto \overline{\texttt{e}}]\overline{\texttt{e}''})) \in \texttt{S}'$ where $\Delta \vdash \texttt{S}'_i <: \texttt{S}_i$, finishing the case.

Case R-Invk: $\texttt{e} = [\texttt{new}\ \texttt{T}_0\texttt{.C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} :: \texttt{P}]\texttt{.m<}\overline{\texttt{U}}\texttt{>(}\overline{\texttt{d}}\texttt{)}$, $\Delta \vdash \mathit{mbody}(\texttt{m<}\overline{\texttt{U}}\texttt{>,P}) = (\overline{\texttt{x}}, \texttt{e}_0)$. Let

$\Delta \vdash \mathit{mtype}(\texttt{m,P}) = \mathbb{D}\texttt{<}\overline{\texttt{V}}\texttt{>.D <}\overline{\texttt{R}}\texttt{>.}[\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}]\texttt{<}\overline{\texttt{X}'} \lhd \overline{\texttt{T}'}\texttt{>}\ \texttt{S}\ \texttt{m(}\overline{\texttt{U}'}\ \overline{\texttt{x}}\texttt{)}$
$MT(\mathbb{D}) = \texttt{module}\ \mathbb{D}\texttt{<}\overline{\texttt{Y}} \lhd \overline{\mathbb{N}}\texttt{>} \lhd \mathbb{N}\ \{\overline{\texttt{CL}}\}$
$\texttt{class}\ \texttt{D<}\overline{\texttt{X}} \lhd \overline{\texttt{S}}\texttt{>} \lhd \texttt{T}\ \{...\} \in \overline{\texttt{CL}}$

By [T-Ann-Inv], $\Delta \vdash \texttt{e} \in [\overline{\texttt{X}'} \mapsto \overline{\texttt{U}}][\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}]\texttt{S}$. Let $\Delta_1 = \Delta + \overline{\texttt{Y}} \lhd \overline{\mathbb{N}} + \overline{\texttt{X}} \lhd \overline{\texttt{S}} + \overline{\texttt{X}'} \lhd \overline{\texttt{T}'}$, and $\Gamma = \overline{\texttt{x}} : \overline{\texttt{U}} + \texttt{this} : \mathbb{D}\texttt{<}\overline{\texttt{V}}\texttt{>.D <}\overline{\texttt{R}}\texttt{>}$. By [T-Method], $\Delta_1; \Gamma \vdash \texttt{e}_0 \in \texttt{S}'$ where $\Delta_1 \vdash \texttt{S}' <: \texttt{S}$. By Lemma 12, $\Delta; \Gamma \vdash [\overline{\texttt{X}'} \mapsto \overline{\texttt{U}}][\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}]\texttt{e}_0 \in [\overline{\texttt{X}'} \mapsto \overline{\texttt{U}}][\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}]\texttt{S}'$ and by Lemma 4, $\Delta \vdash [\overline{\texttt{X}'} \mapsto \overline{\texttt{U}}][\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}]\texttt{S}' <: [\overline{\texttt{X}'} \mapsto \overline{\texttt{U}}][\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}]\texttt{S}$. Also, by [T-Ann-Invk], $\Delta \vdash \overline{\texttt{e}} \in \overline{\texttt{U}''}$ where $\Delta \vdash \overline{\texttt{U}''} <: [\overline{\texttt{X}'} \mapsto \overline{\texttt{U}}][\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}]\overline{\texttt{U}'}$. Then by Lemma 13, $\Delta \vdash [\overline{\texttt{x}} \mapsto \overline{\texttt{e}}][\texttt{this} \rightarrow \texttt{new}\ \texttt{T}_0\texttt{.C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}][\overline{\texttt{X}'} \mapsto \overline{\texttt{U}}][\overline{\texttt{X}} \mapsto \overline{\texttt{T}}][\overline{\texttt{Y}} \mapsto \overline{\texttt{V}}]\texttt{e}_0 \in \texttt{S}''$,

where $\Delta \vdash \mathtt{S}'' <: [\overline{\mathtt{X}'} \mapsto \overline{\mathtt{U}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{V}}]\mathtt{S}'$. Finally, by the transitivity of subtyping, $\Delta \vdash \mathtt{S}'' <: [\overline{\mathtt{X}'} \mapsto \overline{\mathtt{U}}][\overline{\mathtt{X}} \mapsto \overline{\mathtt{T}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{V}}]\mathtt{S}$, finishing this case.

Case R-Inv-Sub: $\mathtt{e} = [\mathtt{e}'' \in [\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{V}}]\mathtt{O}].\mathtt{m}\texttt{<}\overline{\mathtt{R}}\texttt{>}(\overline{\mathtt{d}})$. $\mathtt{e}' = [\mathtt{e}'' \in \mathbb{D}\texttt{<}\overline{\mathtt{V}}\texttt{>}.\mathtt{C}\texttt{<}\overline{\mathtt{R}}\texttt{>}].\mathtt{m}\texttt{<}\overline{\mathtt{R}}\texttt{>}(\overline{\mathtt{d}})$. Let $\Delta \vdash \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{V}}]\mathtt{O}) = \mathtt{P}.\texttt{<}\overline{\mathtt{X}'} \lhd \overline{\mathtt{T}'}\texttt{>}\ \mathtt{R}\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})$. By [T-Ann-Invk], $\Delta \vdash \mathtt{e} \in [\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{V}}][\overline{\mathtt{X}'} \mapsto \overline{\mathtt{T}'}]\mathtt{R}$. But by [R-Inv-Sub], $\Delta \vdash \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}][\overline{\mathtt{Y}} \mapsto \overline{\mathtt{V}}]\mathtt{O}) = \mathit{mtype}(\mathtt{m}, \mathbb{D}\texttt{<}\overline{\mathtt{V}}\texttt{>}.\mathtt{C}\texttt{<}\overline{\mathtt{R}}\texttt{>})$, finishing the case.

Case R-Stop: $\mathtt{e} = [\mathtt{e}'' \in [\overline{\mathtt{X}} \mapsto \overline{\mathtt{V}}]\mathtt{O}].\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}(\overline{\mathtt{d}})$. This rule alters the type annotation for the receiver. However, since [T-Ann-Invk] works on either form of annotation, the type of the expression is preserved.

Case RC-Cast: $\mathtt{e} = (\mathtt{S})\mathtt{e}_0$, $\mathtt{e}' = (\mathtt{S})\mathtt{e}'_0$. By the induction hypothesis, $\mathtt{e}'$ is well-typed, so the type of e'_0 is S by [T-Cast].

Case RC-Field: $\mathtt{e} = (\mathtt{e} :: \mathtt{N}).\mathtt{f}_i$. Because [T-Ann-Field] determines the field type based solely on the annotated static type (which is not altered by [RC-Field]), the type of $\mathtt{e}'$ is identical to that of e.

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

Case RC-Inv-Recv, RC-Inv-Arg: $\mathtt{e} = [\mathtt{e}_0 \in \mathtt{N}].\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}(\overline{\mathtt{d}})$. Let $\mathtt{e}_i$ be the reduced subexpression in e, and let $\mathtt{e}_i$ be reduced to $\mathtt{e}'_i$ in $\mathtt{e}'$. In both of theses cases, the induction hypothesis ensures that $\mathtt{e}'_i$ satisfies the required properties of $\mathtt{e}_i$ in [T-Ann-Invk]. But since the type determined by [T-Ann-Invk] depends solely on m and N, and neither m nor N is altered by these reductions, the static type is preserved.

□

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

Lemma 14 (Class Locations of Method Type Signatures). *For non-variable type* $\mathtt{O}$ *and environment* Δ *where* $\Delta \vdash \mathtt{O}$ *ok, if* $mtype(\mathtt{m}, \mathtt{N}) = \mathtt{O}.\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{T}}\mathtt{>}\ \mathtt{R}\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})$ *then for any type* $\mathtt{O}'$ *s.t.* $\Delta \vdash \mathtt{N} <: \mathtt{O}' <: \mathtt{O}$, *if* $\Delta \vdash mtype(\mathtt{m}, \mathtt{N}) = mtype(\mathtt{m}, \mathtt{O}')$ *then* $\mathtt{O}' = \mathtt{O}$, *i.e.,* $\mathtt{O}$ *is the closest superclass containing* $\mathtt{m}$ *with a matching method signature.*

Proof. Trivial induction on the derivation of $mtype(\mathtt{m}, \mathtt{N}) = \mathtt{O}.\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{T}}\mathtt{>}\ \mathtt{R}\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})$ □

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

1. *If* $\Delta; \Gamma \vdash \mathtt{e}.\mathtt{f}_i \in \mathtt{T}$ *then* $\Delta; \Gamma \vdash [\mathtt{e} :: \mathtt{N}].\mathtt{f}_i \in \mathtt{T}$.
2. *If* $\Delta; \Gamma \vdash \mathtt{new\ R}(\overline{\mathtt{e}}) \in \mathtt{T}$ *then* $\Delta; \Gamma \vdash \mathtt{new\ R}(\overline{\mathtt{e}} :: \overline{\mathtt{N}}) \in \mathtt{T}$.
3. *If* $\Delta; \Gamma \vdash \mathtt{e}.\mathtt{m}\mathtt{<}\overline{\mathtt{V}}\mathtt{>}(\overline{\mathtt{e}}) \in \mathtt{T}$ *then* $\Delta; \Gamma \vdash [\mathtt{e} \circ \mathtt{N}].\mathtt{m}\mathtt{<}\overline{\mathtt{V}}\mathtt{>}(\overline{\mathtt{e}}) \in \mathtt{T}$.

Proof. By analysis over the typing rules that generate annotations.

1. $\Delta; \Gamma \vdash \mathtt{e}.\mathtt{f}_i \in \mathtt{T}$. The only distinction between the antecedents of [T-Field] and [T-Ann-Field] is that the type $\mathtt{N}$, which is the type used for field accesses, is explicitly determined using the receiver in [T-Field]. By using Lemma 11, we have the condition that there is no proper subtype $\mathtt{P}$ of $\mathtt{N}$ s.t., $\Delta \vdash fields(\mathtt{P})$ includes $\mathtt{f}_i$, thus ensuring $\mathtt{N}$ is unique. This unique type is used to annotate the receiver, and therefore the same type referred to in [T-Ann-Field]. Thus, $\Delta; \Gamma \vdash [\mathtt{e} :: \mathtt{N}].\mathtt{f}_i \in \mathtt{T}$. Because none of the argument expressions have been reduced, their static types will match the annotated types exactly and $\Delta \vdash \overline{\mathtt{e}} \in \overline{\mathtt{N}}$. The other antecedents of [T-New] match antecedents of [T-Ann-New] exactly.

2. $\Delta;\Gamma \vdash \texttt{new R}(\overline{\texttt{e}}) \in \texttt{T}$. Because none of the argument expressions have been reduced, their static types will match the annotated types exactly with $\Delta \vdash \overline{\texttt{e}} \in \overline{\texttt{N}}$. The other antecedents of [T-New] match antecedents of [T-Ann-New] exactly.

3. $\Delta;\Gamma \vdash \texttt{e.m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{e}}\texttt{)}$. Again, no reduction has occurred, so by Lemma 14 the annotated type $\texttt{O}$ will match the closest supertype of the bound of the the static type $\texttt{T}_0$ of the receiver that contains $\texttt{m}$. Then $\Delta \vdash \mathit{mtype}(\texttt{m}, \texttt{O}) = \mathit{mtype}(\texttt{m}, \mathit{bound}_\Delta(\texttt{T}_0))$ and the case is finished by [T-Ann-Invk].

□

8.13 Progress

Lemma 15 (Field Values). *For bound environment* Δ, *non-variable type* $\texttt{N}$, *If* $\Delta \vdash \mathit{fields}(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}}$ *and* $\Delta \vdash \texttt{new P}(\overline{\texttt{e}} :: \overline{\texttt{R}}) \in \texttt{P}$ *where* $\Delta \vdash \texttt{P} <: \texttt{N}$ *then* $\Delta \vdash \mathit{fieldVals}(\texttt{new P}(\overline{\texttt{e}} :: \overline{\texttt{R}}), \texttt{N}) = \overline{\texttt{e}'}$ *where* $|\overline{\texttt{e}'}| = |\overline{\texttt{f}}|$.

Proof. Induction over the derivation of $\mathit{fields}(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}}$.

Case N=Object: The definition of *includes* specifies that `Object` includes only the zero-ary constructor. Since the rules of subtyping specify that `Object` is a subtype of only itself, $\Delta \vdash \mathit{fields}(\texttt{Object}) = \mathit{fieldVals}(\texttt{new Object}(), \texttt{Object}) = \bullet$.

Case N= $\texttt{T.C<}\overline{\texttt{R}}\texttt{>}$, $\mathit{fields}(\texttt{N}) = [\overline{\texttt{X}} \mapsto \overline{\texttt{R}}]\overline{\texttt{T}}\ \overline{\texttt{f}}$**:** We proceed by structural induction on the derivation of $\Delta \vdash \texttt{P} <: \texttt{N}$.

Subcase S-Reflex: By [T-Constructor], every valid constructor in a class must initialize all fields $\overline{\texttt{f}}$ with expressions $\overline{\texttt{e}}$.

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

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

Subcase S-Class: $\texttt{N}$ is the instantiated parent class of $\texttt{P}$. In this case, only one definition of *fieldVals* applies. So we know that $\Delta \vdash \mathit{fieldVals}(\texttt{new P}(\overline{\texttt{e}} :: \overline{\texttt{S}}), \texttt{N}) =$

fieldVals($\texttt{new N}(\texttt{e}''), \texttt{N}$) = $\overline{\texttt{e}'}$ where $\overline{\texttt{e}''}$ is a subset of $\overline{\texttt{e}}$ used in the super call. By reasoning analogous to case [S-Reflect], we know $|\overline{\texttt{e}'}| = |\overline{\texttt{f}}|$ (note that we cannot employ the induction hypothesiss directly since the induction is over the derivation of $\Delta \vdash \texttt{P} <: \texttt{N}$, not the derivation of *fieldVals*).

Subcase S-Module: Impossible since this theorem applies only to class types.

□

Lemma 16 (Method Bodies). *If* $\Delta \vdash \mathit{mtype}(\texttt{m}, \texttt{N}) = \texttt{P.<}\overline{\texttt{X}} \lhd \overline{\texttt{T}}\texttt{> R m(}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)}$ *and* $\Delta \vdash \overline{\texttt{V}} <: [\overline{\texttt{X}} \mapsto \overline{\texttt{V}}]\overline{\texttt{T}}$ *then there exists some* e *s.t.,* $\Delta \vdash \mathit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{N}) = (\overline{\texttt{x}}, \texttt{e})$.

Proof. Trivial induction over the derivation of $\Delta \vdash \mathit{mtype}(\texttt{m}, \texttt{N}) = \texttt{P.<}\overline{\texttt{X}} \lhd \overline{\texttt{T}}\texttt{> R m(}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)}$. □

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* $\texttt{new T.C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}$ *where* $\mathit{bound}_\Delta(\texttt{T}) = \mathbb{D}\texttt{<}\overline{\texttt{V}}\texttt{>}$ *and all* $\overline{\texttt{e}}$ *are values.*

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

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 3 (Progress). *For program* $(MT, \texttt{bds}, \texttt{e})$ *s.t.* $\Delta_0 \vdash \texttt{e} \in \texttt{R}$, *if* $\Delta_0 \vdash \texttt{e} \xrightarrow{*} \texttt{e}'$ *then either* e′ *is a value,* e′ *contains a bad cast, or there exists* e″ *s.t.* $\Delta_0 \vdash \texttt{e}' \rightarrow \texttt{e}''$.

Proof. Because $\Delta_0 \vdash \texttt{e} \xrightarrow{*} \texttt{e}'$ we know that e′ is well-typed. We proceed by structural induction over the form of e′.

Case $\texttt{e}' = [\texttt{new N(}\overline{\texttt{e}} :: \overline{\texttt{S}}\texttt{)} :: \texttt{P}]\texttt{.f}$: We know that by [T-Ann-Field] that $\Delta_0 \vdash \mathit{fields}(\texttt{P}) = \overline{\texttt{T}}\ \overline{\texttt{f}}$ where $\texttt{f} = \texttt{f}_i$. By Lemma 15, $\Delta_0 \vdash \mathit{fieldVals}(\texttt{new N(}\overline{\texttt{e}}\texttt{)}, \texttt{P}) = \overline{\texttt{e}''}$ and $|\overline{\texttt{e}''}| = |\overline{\texttt{f}}|$. Then by [R-Field], $\Delta \vdash \texttt{e}' \rightarrow \texttt{e}''_i$.

Case $\mathtt{e}' = [\mathtt{d} :: \mathtt{P}].\mathtt{f}$, $\mathtt{d}$ is not a new expression: By [T-Ann-Invk], $\mathtt{d}$ is well-typed, so by the induction hypothesis, either $\mathtt{d}$ is a value, $\mathtt{d}$ contains a bad cast, or there exists a $\mathtt{d}'$ s.t., $\Delta_0 \vdash \mathtt{d} \rightarrow \mathtt{d}'$. But since $\mathtt{d}$ is not a new expression, it can't be a value. If it contains a bad cast, then so does $\mathtt{e}'$ and we are done. And if $\Delta_0 \vdash \mathtt{d} \rightarrow \mathtt{d}'$ then by [RC-Field], $\Delta_0 \vdash [\mathtt{d} :: \mathtt{P}].\mathtt{f} \rightarrow [\mathtt{d}' :: \mathtt{P}].\mathtt{f}$.

Case $\mathtt{e}' = [\mathtt{d} \circ \mathtt{P}].\mathtt{m}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{e}})$, $\mathtt{d}$ is not a new expression: Analogous to the case above. Case $\mathtt{e}' = [\mathtt{new}\ \mathtt{N}(\mathtt{e}) :: \mathtt{P}].\mathtt{m}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{e}})$. By [T-Ann-Invk], $\Delta_0 \vdash \mathit{mtype}(\mathtt{m}, \mathtt{P}) = \mathtt{0}.\texttt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{T}}\texttt{>}\ \mathtt{R}\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})$ Then by Lemma 16, $\Delta_0 \vdash \mathit{mbody}(\mathtt{m}, \mathtt{P}) = (\mathtt{x}, \mathtt{e}'')$, and by [R-Invk], $\Delta_0 \vdash \mathtt{e}' \rightarrow \mathtt{e}''$.

Case $\mathtt{e}' = [\mathtt{new}\ \mathtt{N}(\mathtt{e}) \in \mathtt{P}].\mathtt{m}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{e}})$: By [T-Ann-Invk] and [T-Ann-New], $\Delta_0 \vdash \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}}) \in \mathtt{N}$. By Theorem 2, $\Delta_0 \vdash \mathtt{N} <: \mathtt{P}$. If $\Delta_0 \vdash \mathit{mtype}(\mathtt{m}, \mathtt{N}) = \mathit{mtype}(\mathtt{m}, \mathtt{P})$ then $\Delta_0 \vdash \mathtt{e}' \rightarrow [\mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}}) \in \mathtt{N}].\mathtt{m}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{e}})$ by [R-Invk-Sub]. Otherwise $\Delta_0 \vdash \mathtt{e}' \rightarrow [\mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}}) :: \mathtt{P}].\mathtt{m}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{e}})$ by [R-Invk-Stop], finishing the case.

Case $\mathtt{e}' = \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}} :: \overline{\mathtt{T}})$: Then either $\mathtt{e}'$ is a value and we are finished, or there is some $\mathtt{e}_i$ in $\overline{\mathtt{e}}$ that is not a value. Then by the induction hypothesis, either $\mathtt{e}_i$ contains a bad cast (and then so does $\mathtt{e}'$), or there exists some $\mathtt{e}_i'$ s.t. $\Delta_0 \vdash \mathtt{e}_i \rightarrow \mathtt{e}_i'$. Then by [RC-New-Arg], $\Delta_0 \vdash \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}} :: \overline{\mathtt{T}}) \rightarrow \mathtt{new}\ \mathtt{N}(\mathtt{e}_0 :: \mathtt{T}_0, ...\mathtt{e}_i' :: \mathtt{T}_i, ...\mathtt{e}_N :: \mathtt{T}_N)$, finishing the case.

Case $\mathtt{e}' = (\mathtt{N})\mathtt{e}''$: Because $\mathtt{e}'$ is well typed, we know there is some $\mathtt{P}$ s.t., $\Delta_0 \vdash \mathtt{e}'' \in \mathtt{P}$. If $\Delta_0 \not\vdash \mathtt{P} <: \mathtt{N}$ then $\mathtt{e}'$ is a bad cast and we are done. Otherwise $\Delta_0 \vdash \mathtt{e} \rightarrow \mathtt{e}''$ by [R-Cast].

□

8.14 Type Soundness

Theorem 4 (Type Soundness). *For program* $(MT, \mathtt{bds}, \mathtt{e})$ *s.t.* $\Delta_0 \vdash \mathtt{e} \in \mathtt{T}$, *evaluation of* $(MT, \mathtt{bds}, \mathtt{e})$ *yields one of the following results:*

1. $\Delta_0 \vdash \mathtt{e} \xrightarrow{*} \mathtt{v}$ *where* $\mathtt{v}$ *is a value of type* $\mathtt{S}$ *and* $\Delta_0 \vdash \mathtt{S} <: \mathtt{T}$.

2. $\Delta_0 \vdash \mathtt{e} \xrightarrow{*} \mathtt{e}'$ *where* $\mathtt{e}'$ *contains a bad cast,*

3. *Evaluation never terminates, i.e., for every* $\mathtt{e}'$ *s.t.* $\Delta_0 \vdash \mathtt{e} \xrightarrow{*} \mathtt{e}'$ *there exists* $\mathtt{e}''$ *s.t.* $\Delta_0 \vdash \mathtt{e}' \rightarrow \mathtt{e}''$.

Proof. Immediate from Theorems 1, 2, and 3. □

Chapter 9

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 [8] is a statically typed extension to Java that supports mixin modules. Mixin modules are a generalization of mixin classes as introduced in [9]. 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* by *hiding* and *renaming* operations, but the use of structural matching of signatures doesn't meld seamlessly in the nominally typing of Java.

SMARTJAVAMOD [7] 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

*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.

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 [20] 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 that 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 [29, 30] 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). Scala *modules* are singleton classes that do not specify a static signature, which prevents them from creating independent state. But Scala does not support first-class generics, preventing modules from being parameterized by type.[†] Consequently, Scala components rely on hard-coded references to other modules and classes, making it more difficult to develop modules

[†]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.

independently of one another.

Fortress [24] 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 [32, 12, 13]. Neither of the two widely used dialects of ML–Standard ML [26] and OCaml [19]–support recursive modules.

One issue that has arisen in putative recursive extensions to the ML module system is the *double vision* problem[13], 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 6.2.

A potential solution to the ML recursive module problem is the module system proposed in [31], which is based on the same module architecture for PLT Scheme discussed in [15]. 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 have been submitted the Java Community Process, namely JSR-277 and JSR-294. *Java Specification Request 277: Java Module System*[23] 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*[25] 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.

ComponentJ [34] is a language extension for Java that views components as service providers. Components import and export methods, but not types. Components in ComponentJ are first-class values implemented using objects, implying that there is no inheritance across components.

Our use of modules signatures containing class prototypes is most closely related to the notion of virtual classes in GBETA [14]. In GBETA, class members may be virtual classes, akin to virtual methods in Java. Virtual classes are class valued attributes of an object that can overridden and extended in subclasses. An overriding class must support all of the members of the class that it overrides. In GBETA virtual classes are accessed relative to an instance of the enclosing class, which means that such classes must be configured as singletons to model components. In CGEN, a module instance is unique for each distinct tuple of type parameters. Virtual classes support extensibility in code development, but classes containing virtual class members typically contain explicit dependencies on other classes, so they are

not true components.

Chapter 10

Conclusion

A general component system is essential in decomposing applications into independent and composable units of compiled code. As the Component NEXTGEN architecture demonstrates, a component system can be formulated for a nominally typed object-oriented language supporting first-class generic types simply by adding appropriate annotations and syntactic sugar.

10.1 Future Work

Our development of the current CGEN framework suggests several directions for further exploration. We briefly discuss the most immediate ones below.

10.1.1 Performance of Components

Support for components in CGEN introduces additional overhead to instantiate module classes and generates more forwarding methods and interface-based method dispatches into the implementation. We do not believe the overhead to be significant because modules are translated into generic classes, and the extra forwarding methods are typically inlined by JIT compilers. However, instead of relying simply on the past performance metrics of NEXTGEN, we would like to better substantiate our claims using a set of benchmarks specifically targeting components. The overhead of CGEN modules can be compared against other component systems in Java, as well as legacy packages providing the same functionality. Since not all alternatives support module mixin classes, we would prohibit classes from extending module imports— the CGEN implementation applies uniformly to all modules

and their containing classes.*

Of course, since modules in CGEN can be composable by third parties, our benchmarks would need to provide metrics comparing both packages with hard-coded references†, as well as packages employing the object patterns discussed in Chapter 2. These comparisons would help analyze the overhead associated with the heterogenous translation of components used in CGEN. In the future, we could see if the overhead could be reduced by employing more sophisticated code sharing techniques across module instantiations.

10.1.2 First-class Modules

Currently in CGEN, module instantiations can be only created using final, second-class bind declarations that can appear only at the top-level of a program. A logical extension of our current design would be to allow support first-class modules, allowing the run-time behavior of a program to change the instantiated (linked) modules referred to by bind variables. We would allow bind declarations to appear in non-top-level contexts, e.g. class and method bodies, and allow bind variables to be mutable and be used as arguments to method invocations. Bind declarations, as specified in Chapter 5, however, can not support mutually dependent/recursive bind declarations in method bodies, e.g.,

```
void m() {
   bind S1<Y> X = M1<Y>;
   bind S2<X> Y = M1<X>;
   ...
}
```

because expressions can only reference types declared earlier in the method body— this is similar the treatment of anonymous inner classes in Java. A possible solution is to allow bind

*Instantiating a module mixin class in CGEN is identical to instantiating a non-mixin class, since the super type of a class is simply a reference in the constant pool

†Code containing hard-coded references would not be as modular and may require weaker type bounds or additional run-time coercions

declaration to declare multiple module instantations. The above example could be rewritten as

```
void m() {
   bind S1<Y> X = M1<Y>, S2<X> Y = M1<X>;
   ...
}
```

First-class modules can be implemented by introducing light-weight generic classes to represent each module. An instantiated module could then be represented by the singleton instantiation of the representative generic class. To pass modules as arguments to method invocations, run-time support for signatures is necessary to provide explicit types to bound the classes provided by first-class modules, since all types in Java must be accessed using hard-coded, fully-qualified references.

The Reflection API would require extensions to (1) provide reification of modules and signatures, (2) pre-process CGEN-encoded identifiers of hygienic methods and module classes, and (3) filter out CGEN-generated forwarding methods. If CGEN was adopted as an extension to the Java platform, these extensions could be seamlessly incorporated into the Reflection API.

10.1.3 Module Bundling

A logical extension of the CGEN framework would be to support the construction of modules that encapsulate other modules. A developer could specify a set of concrete module instantiations to use in in the body of a module. In other words, bind declarations could appear inside module declarations. To preserve the notion of components as independent units of compiled code, CGEN would need to bundle the explicitly linked modules with the enclosing module. This could be done using a *.jar* file or using some functionality of JSR277[23].

Appendix A

The CGEN Implementation Code Structure

This Appendix discusses the high-level architecture and code layout of the CGEN compiler. The CGEN framework is an extension of the NEXTGEN compiler, an extension of the Java 5 compiler provided under the Sun Shared Source Code license. In developing the CGEN compiler we have tried to preserve the original `javac` compiler as much as possible, in an effort to ensure that the original `javac` code and the NEXTGEN branch are as similiar as possible. This facilitates future portability to the Java 6.0 code base when it stabilizes.

The Java 5 compiler, as well as the CGEN compilers derivied from it, are written in Generic Java. As a result there is more precise type checking in the source of these compilers than would be possible with ordinary Java.

Throughout this Appendix, it is assumed that the reader has a familiarity with Subversion, JUnit, Ant, and Unix. You must have Java 5.0, Ant, and Junit installed on your computer. A tutorial on SVN is available at: http://subversion.tigris.org/. A tutorial on JUnit is available at http://www.junit.org/ A tutorial on Ant is available at http://ant.apache.org/.

A.1 The CGEN CVS Repository

The CGEN source code is maintained in SVN repository in the `javaplt` cs.net account. To checkout the CGEN source code, you need to execute the following command:

```
svn checkout svn+ssh://javaplt@[server]/home/javaplt/
   .svnroot/nextgen2/trunk nextgen2
```

where `[server]` can be any linux or unix server managed by IT, e.g., finland, greenland, or iceland.

This `svn checkout` will create a directory called `nextgen2` in the current directory with the latest version of CGEN. After checking out a copy of CGEN, execute a `cd nextgen2` and then type the following command to set up the CGEN environment:

```
$ source etc/bashrc
```

After setting up the enviroment, type in the following command:

```
$ ant all
```

Doing so will run all the Ant targets in the project.

A.2 The CGEN Project Directory Structure

The `nextgen2` project directory contains the following files and subdirectories:

- `bin` This directory contains a set of script files to run the CGEN compiler and run-time environment.
- `build.xml`. This file contains the XML source code for all Ant targets associated with the CGEN project.
- `built`. This directory is not present on a clean checkout. It is created by Ant to store the class files for all source files associated with CGEN.
- `etc` This directory contains configuration files for CGEN
- `jars` Contains the jar files for the compiler and classloader, constructed with target `ant jar`.
- `lib` This directory contains the third party libraries used by CGEN.
- `nextgen2` This directory contains the files distributed with the CGEN binary.
- `src/` This directory contains all the java source code for the CGEN compiler and related utilities and tests.

A.3 Ant Targets in the CGEN Project

The CGEN project contains a `build.xml` file located in the root directory. The ant targets can be run using the command `ant [cmd]` where `[cmd]` can be

- `all` Compiles all source code and runs unit tests
- `compile` Compiles the CGEN compiler and class loader
- `compile-benchmarks` Compiles the CGEN benchmarks
- `compile-compiler` Compiles only the CGEN compiler
- `compile-loader` Compiles only the CGEN class loader
- `compile-tests` Compiles the CGEN test suite
- `compile-unittests` Compiles a set of unit tests for CGEN
- `info` Outputs Java version, ant version, and Java class path
- `quick_info` Outputs Java version and ant version
- `simpletests` Executes the CGEN test suite
- `snapshot` Archives a snapshot of the current CGEN source code
- `jar` Generate a new distribution jar
- `clean` Delete generated class files
- `clean-tests` Delete only the generated test and benchmark files

A.4 Releasing a New Version of CGEN

Currently the CGEN project is hosted on `japan.cs.rice.edu`. To post a new release of CGEN, upload a distribution jar to the `/home/javaplt/public_html/nextgen/jars/`.

A.5 CGEN Package Design

The CGEN project is divided into three main sections: compiler, byte code processor, and classloader. A set of utility classes located in `edu.rice.cs.nextgen2.util` are used across these three sections.

A.5.1 Compiler Package Design

The source code for the CGEN compiler is located in `edu.rice.cs.nextgen2.compiler`. The code is further divided in the following sub-packages:

- `code` Contains code for many of the data structures, including types and symbols, utilized by the different phases of compilation.
- `comp` Contains the code for computing most phases of compilation, including symbol table entry, type checking ,data flow checking, and generic type erasure.
- `flatten`. Contains the core NEXTGEN code. This code is responsible for converting type-dependent operations into snippet calls. No comparable package exited in the Java 5.0 `javac` compiler.
- `jvm` Contains the code related to reading and generating Java class files.
- `main` Contains the main interfaces for using the NEXTGEN compiler.
- `parser` Contains the code used for scanning and parsing Java source code.
- `resources` Contains the resource files used to display output for the user.
- `tree` Contains code for the abstract syntax trees, and their related factories and visitors, utilized by the NEXTGEN compiler.
- `util` Contains many general-purpose data structures, including parametric collection classes used by the NEXTGEN compiler. Although the Java collections library contains many similar classes, the implementation of the locally provided classes are more adapted to the NEXTGEN design.

A.5.2 Byte Code Processor Package Design

The byte code processor is used to propagate snippet methods between related generic class templates and also polymorphic method template classes. The classes used in the CGEN byte code processor are located in the following package: `edu.rice.cs.nextgen2.bytecode`. The main entry point is the class `SnippetProcessor`.

A.5.3 Class loader Package Design

The main classes used in the CGEN class loader are located in the following package: `edu.rice.cs.nextgen2.classloader`. The main entry point is the class `Runner`.

Bibliography

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 49–65, Atlanta, GA, 1997.

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

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

[4] 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`.

[5] E. Allen, R. Cartwright, and B. Stoler. DrJava: A lightweight pedagogic environment for Java. In *SIGCSE*, 2002.

[6] 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.

[7] 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.

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

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

[10] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *OOPSLA*, pages 183–200, Vancouver, BC, 1998.

[11] Robert Cartwright and Guy L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *OOPSLA*, pages 201–215. ACM, 1998.

[12] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *PLDI*, pages 50–63, New York, NY, USA, 1999. ACM Press.

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

[14] E. Ernst, K. Ostermann, and W. Cook. A virtual class calculus. In *ACM POPL*, 2006.

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

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

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

[18] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *OOPSLA*, 23(3):396–450, 2001.

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

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

[21] Sun Microsystems. *JavaBeans API Specification*, graham hamilton edition, Aug 1997.

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

[23] Sun Microsystems. JSR 277: Java Module System, 2005.

[24] Sun Microsystems. The Fortress language specification, Sept 2006.

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

[26] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

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

[28] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[29] Martin Odersky. Programming in Scala, 2006. Draft.

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

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

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

[33] James Sasitorn and Robert Cartwright. Efficient first-class generics on stock Java virtual machines. In *SAC*, 2006.

[34] João Costa Seco and Luís Caires. A basic model of typed components. *Lecture Notes in Computer Science*, 1850:108–128, 2000.

[35] Clemens Szyperski. *Component Software.* Addison-Wesley, 1998.

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

[37] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the java programming language. In *SAC*, pages 1289–1296, New York, NY, USA, 2004. ACM Press.

[38] Mirko Viroli. Parametric polymorphism in Java: an efficient implementation for parametric methods. In *SAC*, pages 610–619, 2001.

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